Performance comparisons of various runs algorithms

Performance comparisons of various runs algorithms

By

Robert Fuller, 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 (2012) McMaster University
(Computing and Software) Hamilton, Ontario

TITLE:	Performance comparisons of various runs algorithms
AUTHOR:	Robert Fuller, B.Sc. (McMaster University)
SUPERVISORS:	Dr. Frantisek Franek and Dr. William F. Smyth
NUMBER OF PAGES:	x, 48.

Abstract

This thesis discusses and describes empirical comparisons of execution times of three programs for computing runs in strings. Since two of the programs were thought to be of $O(n \log n)$ algorithms (`crochB` and `crochB7`) and the third is an implementation of a linear algorithm (`runFinder`), it was expected that for larger strings `runFinder()` will strongly outperform the other two programs in the processing of long strings. The aim of this study is thus manifold. We establish the upper limits of lengths of strings for which the performances of `crochB` and `crochB7` are faster or comparable to the performance of `runFinder`; we also investigate what kind of penalty in performance `crochB7` incurs for the memory saving implementation; furthermore, we wish to explore the relative trade-offs of using one technique (represented through the programs with which experimentation was gone about) over another: within what context would it be advantageous to utilize one program over another of those that are being investigated.

The motivation for this work is the continuation of work of Franek, Jiang, Smyth, Weng, and Xiao, who implemented a space efficient version of Crochemore's repetition algorithm [6], and then extended it to compute runs [4, 5]. The three programs tested are:

1. `crochB` – a direct `C++` implementation of the extension of Crochemore's algorithm for runs by Franek, Jiang, and Weng without any space savings techniques;
2. `crochB7` – a space efficient version of `crochB` by the same authors,
3. `runFinder` – an efficient `C++` implementation by Hideo Bannai from the

Department of Informatics at Kyushu University in Japan. His implementation utilizes the linear-time strategy of computing the suffix array of the string; using the suffix array it then computes the LCP array; using the suffix and LCP arrays it computes the Lempel-Ziv factorization; from the Lempel-Ziv factorization all leftmost runs are computed using Main's algorithm; and the rest of the runs are computed using Kolpakov-Kucherov's algorithm.

In this thesis, the three programs are discussed, the experimental setup for the performance measurements is described, the measurements are presented and a brief analysis of the results follows. It will be shown that although an expectation of $O(n\ log\ n)$ performance can be expected in the case of processing of one category of investigated data by the latest version of the implementation of the Crochemore program, in some circumstances (discussed), a performance expectation of order n^2, and in others one between this and one of order $n\ log\ n$ will be encountered.

Acknowledgments

I wish to first thank my thesis supervisors, Drs. Franya Franek and Bill Smyth, for their support, encouragement, and the invaluable wisdom and experience they offered toward the completion of this thesis. I would also like to acknowledge the kind correspondence and gracious supplement of source code, explanations, and example testing code provided by Dr. Hideo Bannai of the Department of Informatics at Kyushu University, Japan.

Contents

Notation and Definitions

In this preliminary section we discuss the notions and notation pertinent to the understanding of the work presented in this thesis.

Let x be a string over an alphabet $\mathcal{A}$. This means that x is a sequence of symbols from $\mathcal{A}$. The length of x, denoted by $|x|$, is the number of symbols in x. The empty string, i.e., a string of length 0 is denoted by ε. The notation $x[i]$ denotes the i-th symbol of x, and we speak of i-th position in string x. The notation $x[i..j]$ denotes the substring of x consisting of symbols from position i to position j inclusively. An important notion of regularity in strings is the *period*: a string $x = x[1..n]$ has a period p if $x[i] = x[i+p]$ for any $1 \leq i \leq n-p$.

Consider a string u. We denote by uu the concatenation of two copies of u. We refer to uu as a *square of period* $|u|$; this is also denoted by u^2. The expression uuu may also be abbreviated as u^3 and is referred to as a *cube of period* $|u|$. The symbol u^k refers to concatenation of k copies of u and is refereed to as a k *repetition of period* $|u|$. Since a string x has at least one period, but can have more, the notation $x = u^r$ where $|u|$ is the smallest period of x and where $r = |x| \ / \ |u|$ is referred to as the *normal form* of x.

Let $x = x[1..n]$. Then the substring $x[1..i]$, for any $i \leq n$, is said to be a *prefix* of x; if $i < n$ we speak of a *proper prefix* of x. The empty string ε is said to be a *trivial prefix* of x. Similarly, $x[i..n]$ for any $1 \leq n$ is said to be a *suffix* of x; if $i > 1$ we speak of a *proper suffix* of x. The empty string ε is said to be a *trivial suffix* of x.

Let $x = uv^rw$ for some substrings u, w and a non-empty substring v and and integer $r \geq 2$. Then we speak of an occurrence of an r repetition of v in x, or simply of an r repetition of v in x. A *primitively rooted* repetition vv is such that its *generator* v is *primitive*, i.e. v is not a repetition itself. A repetition $x = uv^r w$ in x is said to be *maximal* if it cannot be extended to the left – i.e. there is no u_1 so that $x = u_1 v^{r+1} w$, and cannot be extended to the right – i.e. there is not w_1 so that $x = uv^{r+1} w_1$. A repetition $x = uv^r w$ can be encoded as a triple (s, p, r) where s indicates the starting position of the repetition, p the period of the repetition, and r the exponent of the repetition. The end of the repetition can be easily computed as $e = s + rp - 1$. Alternatively, it can be encoded also as (s, p, e), where s again refers to the starting position of the repetition, p again refers to the period of the repetition, and e the ending position of the repetition. The exponent can be easily computed using $r = (e - s)\ /\ p$. It is clear [1] that knowing explicitly all maximal primitively rooted repetitions gives us the implicit knowledge of all repetitions as every repetition is a part of a maximal primitively rooted one.

A more concise and succinct notion still is that of a *run* in a string x. In a sense, it is a maximal fractional primitively rooted repetition. To be more precise, (s, p, r, t) encodes a run if (s, p, r) encodes a maximal primitively rooted repetition that cannot be shifted left – i.e. either $s = 1$ or there is no r repetition of period p starting at position $s - 1$, and $(s + t, p, r)$ encodes a maximal primitively rooted repetition that cannot be shifted right – i.e. there is no r repetition of period p starting at position $s + t = 1$. We can encode a run also as a triple (s, p, e), where s indicates the starting position of the run, p its period, and e its ending position. Clearly, $r = (e - s)\ /\ p$ and $t = (e - s)\ \%\ p$.

Chapter 1

Introduction

Strings, or linear sequences of symbols, represent one of the most fundamental objects in discrete mathematics and theoretical computer science. In the the study of combinatorial and computational properties of strings, several important notions arise. Of the most fundamental is the notion if the ***period*** of a string. Since strings are practically structureless objects, any structural knowledge that can be derived about them can be highly beneficial to their processing. If a string posses significant periodicity, its processing can be simplified and/or sped up. For instance, data compression highly relies on the knowledge of the periodicity of the string to be compressed. Many algorithms needed for sequencing of DNA or proteins rely on the knowledge of the periodicity of the string representing the DNA or the protein.

It is thus imperative to have highly time and space efficient algorithms for determining the periodicity of a possibly large string. As indicated in the preliminary section, computing a list of all runs in a string provides a rich information of the periodicity of a string. Please, see [16] for details and the references concerning the history of algorithms for computing runs in a string.

In this context, the concept of "calculation" refers to outputting or recording of locations of occurrences of all runs within a string.

While the methods associated with the linear-time calculation of runs and repetitions have been established theoretically, the efficiency of their various implementations must be considered, and compared, with that of alternative methods and their implementations. Some consideration is given to whether actual performance values, with respect to time consumption, are in keeping with what is expected. It is a goal of this thesis to determine this. This motivation is provided in part by the desire to determine in what context which of the tested programs would be most practical for use. That is, is there a point (as would be suggested by the functions representing the theoretical run time of each of the programs) with respect to processed string length at which the performance of one program surpasses that of the others. It is hypothesized that there is, and the investigations within this thesis will seek to both prove this and relay the point at which this occurs. An additional aim of this study is to determine what kind of penalty in performance `crochB7` pays in comparison to `crochB` for the memory saving implementation.

In this thesis we present comparisons of `C++` implementations pf three algorithms for computing runs. The two of them, `crochB` and `crochB7` rely on the refinement process due to Crochemore [1]. The third implementation – `runFinder` – due to Hideo Bannai from the Department of Informatics at Kyushu University in Japan utilizes the linear-time strategy of computing the suffix array of the string, using the suffix array it then computes the LCP array, using the suffix and LCP arrays it computes the Lempel-Ziv factorization, from the Lempel-Ziv factorization all leftmost runs are computed using Main's algorithm, and the rest of the runs are computed using Kolpakov-Kucherov's algorithm, see [16].

The original Crochemore's so-called *partitioning algorithm* [1] for computing maximal repetitions in strings, the first of many $O(n \log n)$ repetitions algorithms, was implemented in a space efficient manner by Franek, Smyth, and Xiao [6] and latter extended to compute runs by Franek and Jiang [3]. However, this extension while of the same complexity $O(n \log n)$ as the underlying partitioning algorithm, was not very memory efficient as it required additional $O(n \log n)$ memory for cumulation of computed squares and their consolidation into runs. Franek, Jiang and Weng in [4, 5] modified the algorithm [3] to compute runs, or maximal primitively rooted repetitions, or primitively rooted distinct squares when the types rather than occurrences of squares are counted, without any need for extra memory and preserving the original complexity $O(n \log n)$. The program `crochB` is based on this algorithm and it is a straightforward implementation in `C++` without any memory saving techniques. Thus `crochB` has an $O(n \log n)$ time complexity and its memory requirement is $19n$ integers, where n is the length of the input string. The program `crochB7` is a modification of `crochB` using virtualization and multiplexing of memory to lower the memory requirement to $13n$ integers, while implementing the same algorithm and thus preserving the original complexity. Though both `crochB` and `crochB7` can compute maximal repetitions or distinct squares, the investigation in this thesis was limited to the computation of runs as `runFinder` computes only runs.

The method by which performance measurement would be observed was an important consideration of the investigation around which this thesis is centred. It would be ideal to remove from consideration any factors that might not have a direct relation to the comparable tasks at hand, between the programs observed. As such, the primary technical mechanism utilized was through the facilities offered in the `time.h` library of the `C++` programming language. In particular, a mechanism is offered that allows for the reporting of an approximation of the number of clock ticks elapsed since the initial call of the program in which the reporting function (`clock()`) is called. Furthermore, it was originally desired to exclude such operations as memory allocation from observation (primarily due to the unpredictability of operating system-based functionality with respect to actual time utilized). This exclusion is done as a property of the clock-tick method of time observation: as those time-unpredictable function are handled outside of the running program, their utilized clock cycles are part of those whose values are to be reported by `clock()`.

Observed data that were reported consisted of the calculated actual time required for the processing of a string of a particular length. Since the facilities offered for the calculation of running time provide an *approximation* of clock cycles utilized, shorter strings presented the challenge of appearing to utilize no cycles for their processing. This was overcome by the repeated running of the processing portion of the programs' code for smaller strings (the number of iterations being dependant on the shortness of the input string) and then averaging of the result with respect to the number of iterations of the processing that were performed upon a given string.

Overview

Within this thesis is presented a brief introduction to the essential aspects of operation for each of the programs considered in testing (Chapters 2 – 4). Important particulars of the methods and processes utilized for testing are discussed (Chapter 5), with reference to and discussion about several unexpected aspects of observations made (Chapter 6). Those areas that would seem particularly interesting for future investigation are mentioned and briefly discussed along with conclusions about the experiments performed and their results (Chapter 7). Relevant data and graphs are supplied, appended to this main document.

Chapter 2

A brief description of `crochB` program

It was estimated by M. Crochemore himself that an implementation of the partitioning algorithm [1] for computing maximal repetitions in a string would require $20n$ integers of memory due to a significant overhead in data structures needed for the partition refinement process. In 2003, Franek, Smyth, and Xiao [6] implemented in `C++` the algorithm using various memory saving techniques like *array virtualization* – when an array's storage is distributed across other arrays utilizing their unused fragments of storage while keeping a constant-time access, or *memory multiplexing* – when two stacks or queues are made to share the same storage as it can be shown that as one grows, the other shrinks and vice-versa, and so they would never corrupt the storage of the other. These techniques allowed to lower the overall memory requirements to either $14n$ or $15n$ integers – depending which version is considered. It will be explained later in this chapter how the refining process is used to identify all primitively rooted squares in a string. Any maximal repetition (s, p, r) can be viewed as a consolidation of $(r-2)$ shifts of the leading square $(s, p, 2)$ of the repetition.

For instance, $..aabaabaabaab..$ consists of 7 primitively rooted squares:

$..\underline{aabaab}aabaab..$,

$..a\underline{abaaba}abaab..$,

$..aa\underline{baabaa}baab..$,

$..aab\underline{aabaab}aab..$,

$..aaba\underline{abaaba}ab..$,

$..aabaa\underline{baabaa}b..$,

$..aabaab\underline{aabaab}...$

Thus, the squares are consolidated to maximal repetitions. Due to the design of the underlying Crochemore's repetition algorithm, these squares are determined in the natural order (based on the starting position) and thus they can be consolidated on the run and they do not have to be first collected and the consolidated.

The implementation [6] was used in 2009 as a basis by Franek and Jiang [2, 3] and the program was embellished to compute runs. Any run (s, p, r, t) can be viewed as consisting of $t + 1$ shifts of a maximal repetition (s, p, r, t). For instance, $..aabaabaabaabaa..$ consists of 3 maximal repetitions:

$..\underline{aabaabaabaab}aa..$,

$..a\underline{abaabaabaaba}a..$,

$..aa\underline{baabaabaabaa}...$

The implementation [2, 3] thus relies on the underlying [6] and its reporting of the found repetitions. Since the repetitions are not reported in any discernible order, the repetitions must first be collected, and then consolidated into runs. The data structure required to collect the repetitions and to consolidate them into runs was implemented in $3n \log n$ integers of memory. Though the run-time complexity $O(n \log n)$ of the underlying partitioning algorithm was preserved, the significant memory overhead rendered the implementation impractical.

In 2011, Franek, Jiang and Weng in [4, 5] modified the algorithm [6] to compute runs, or maximal primitively rooted repetitions, or primitively rooted distinct squares when the types rather than occurrences of squares are counted, without any need for extra memory and preserving the original complexity $O(n \log n)$. It consolidates the squares into runs on the go while utilizing some of the auxiliary data structures needed for the refinement process – these data structures are not needed when the the refinement process of the level p is completed and when the found squares of the period p are reported and consolidated into runs.

The program `crochB` is based on this algorithm and it is a straightforward implementation in `C++` without any memory saving techniques. Thus `crochB` exhibits $O(n \log n)$ time and its memory requirement is $19n$ integers, where n is the length of the input string.

The underlying partitioning algorithm relies on the computation of classes of equivalence, each containing indexing positions of the input string. The members of the same class are starting positions of identical substrings of a given length. If two substrings $x[i..i+k]$ and $x[j..j+k]$ of a string x are equal, then this is denoted by writing $i \approx_k j$ and means that the positions i and j are equivalent. The algorithm computes the classes of equivalence $\approx_k$ for successive k, referred to as the levels of refinement.

Consider for instance a string $x = abaababaabaabab\$$. For technical reasons, we imagine that the string is terminated by a sentinel symbol \$, that is considered smaller (with respect to an alphabetical ordering) than any other symbol of the

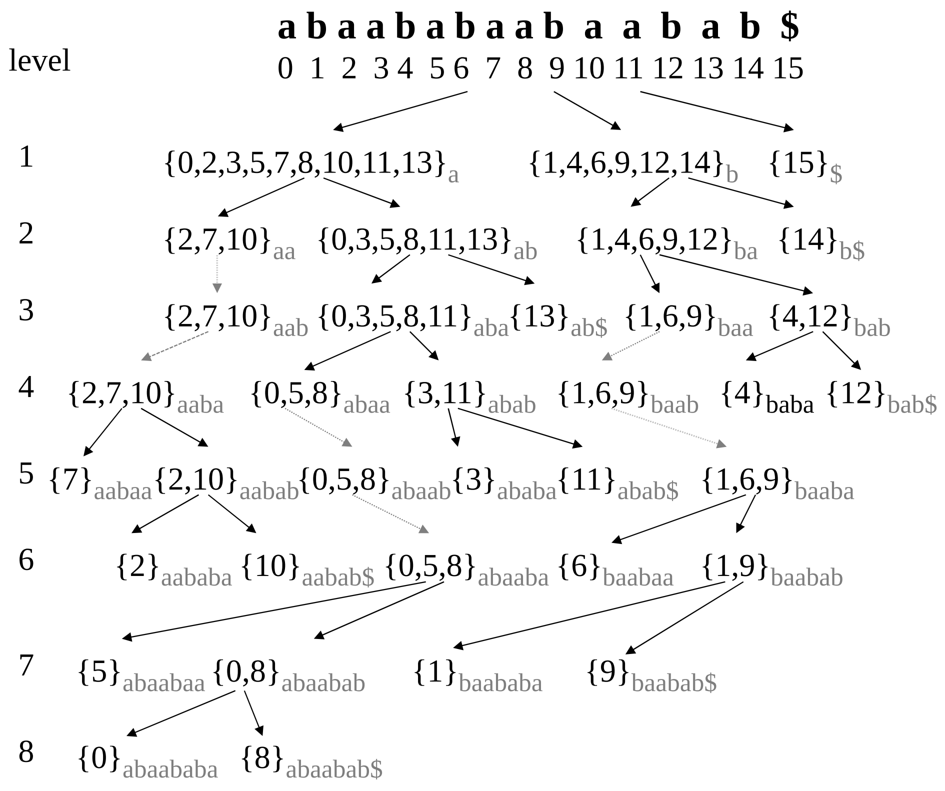
a b a a b a b a a b a a b a b $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
level
1 {0,2,3,5,7,8,10,11,13}a {1,4,6,9,12,14}b {15}$
2 {2,7,10}aa {0,3,5,8,11,13}ab {1,4,6,9,12}ba {14}b$
3 {2,7,10}aab {0,3,5,8,11}aba {13}ab$ {1,6,9}baa {4,12}bab
4 {2,7,10}aaba {0,5,8}abaa {3,11}abab {1,6,9}baab {4}baba {12}bab$
5 {7}aabaa {2,10}aabab {0,5,8}abaab {3}ababa {11}abab$ {1,6,9}baaba
6 {2}aababa {10}aabab$ {0,5,8}abaaba {6}baabaa {1,9}baabab
7 {5}abaabaa {0,8}abaabab {1}baababa {9}baabab$
8 {0}abaababa {8}abaabab$

Figure 2.1: The classes of equivalence in Crochemore's partitioning

string – very similar to the idea of `C` strings terminated by the `NULL` character. The classes on each level of refinement are depicted in Figure 2.1.

Classes indicated on level k are classes of $\approx_k$ equivalence. Thus, a class on level k consists of positions in the string x at which identical substrings of length k of x start. For instance, the class $\{0, 5, 8\}$ on level 6 indicates that $x[0..6] = x[5..1] = x[8..14] = abaaba$ (indicated by the little gray subscript of the class in the diagram). Please, note, that the indexing of strings starts with 0 as it is more natural for `C/C++` based implementations. When a class splits into several classes by the refinement, this is indicated by black solid arrows in the diagram; if a class remains untouched by the refinement, this is indicated by a dotted gray arrow.

Even though it is possible to determine the classes directly from the string, the processing to do so would be $O(n^2)$: scan the string from left to right to determine the classes of the 1st level, scan the string again to determine the classes of the 2nd level, etc. To achieve $O(n \log n)$ complexity, a smart refinement of level $k+1$ using already computed so-called small classes of level k is performed [1]. An additional important feature of the refinement is the fact that the indices in the classes are kept in the natural order, though the same is not true for the classes themselves: for instance in Figure 2.1, the indices in class the $\{0, 5, 8\}$ on level 6 would be recorded in that order, but we cannot rely on this class being computed before the class $\{6\}$.

The process of refinement finishes when all classes are refined to singletons. In Figure 2.1, once a class shrinks to a singleton, it does not appear in subsequent levels, but if it were to, each level would thus be a partitioning of $\{0, 1, \ldots, n-1\}$ into classes. This is why the algorithm is often referred to as a partitioning algorithm.

The refinement process and the resulting classes would be of no use if the primitively rooted squares could not be obtained from them. Without regard to complexity, having all non-singleton classes of level k suffices to determine all primitively rooted squares of period k is rather straightforward: just scan every class and if two consecutive indices differ exactly by k, the smaller index is a starting position of a primitively rooted square of period k. For example, consider the class $\{0, 3, 5, 8, 11\}$ on level 3: thus $0, 3$ indicate a square *abaaba* at position 0; $5, 8$ indicate a square *abaaba* at position 5; and $8, 11$ indicate a square *abaaba* at position 8. This naïve approach is not feasible as it would again inflate the run-time complexity to $O(n^2)$.

Thus an auxiliary array $Gap[k]$ is maintained for each level k that supplies information relating to indices that are of distance k from their predecessors. More precisely $Gap[k] = i$ if and only if $i - k$ and i belong to the same class on level k and are consecutive indices in the class. The *Gap* array can be modified and maintained during the refinement process without destroying the run-time complexity.

The program `crochB` relies on the *Gap* array and directly traces the runs through the squares determined from the *Gap* array. Using other auxiliary structures that are needed only for the refinement to remember already consolidated parts of a run so it is not re-consolidated again. The tracing of maximal repetitions is achieved in the same way. Finally, for computation of the number of primitively rooted distinct squares, it is sufficient to consider only one square per a class, i.e. once a class has been used through the traversal of the *Gap* array, it is "marked" and if another square from the same class is to be determined, it is ignored. In this fashion, the program can as easily determine the runs as the maximal repetitions or the number of distinct squares.

The implementation of `crochB` is similar to [6], however without any of the memory-saving techniques discussed in the next chapter, leading to $O(n \log n)$ run-time complexity and $19n$ integers storage requirement.

Chapter 3

A brief description of the `crochB7` program

The program referred to by `crochB7` is the result of several progressive refinements to the `crochB` program. As the name of this latest version suggests, there were 6 revisions completed between the first and latest program. The trend of the major changes between revisions was to move away from some stored data toward a calculation of certain values (as will be discussed below) to obtain the same results. As a result of this, of course some increase of runtime was expected. As will be seen in some of the testing data, this has occurred between `crochB` and `crochB7`, more notably during the processing of larger and / or more complex input data.

Below is given some detail as to the progression from `crochB` to `crochB7` in the modifications made from one revision to the next.

3.1 Progression from `crochB` to `crochB7`

The original version of `crochB` utilized several arrays for its calculations. Some of these remained through future refinements of the program, but some were replaced by functions to perform the same tasks, but with a reduced requirement for memory. Below are details of the used arrays and the functions that subsequently replaced some of them. Other refinements were made, and are detailed below as well.

The majority of information used in the calculations done by Crochemore's algorithm is stored in arrays, indexed by position within the input string. They are primarily used to emulate linked lists in an efficient way.

The first two arrays are *CStart* and *CEnd*. These give the start and end positions of classes, each indicated by a given index of the arrays.

Next are *CNext* and *CPrev*, used to emulate forward and backward pointers. They respectively point to the next and previous elements of a given class.

An array $CMember$ is used to indicate the membership of an element, indicated by index, within a class, indicated by stored value (*i.e.*, $CMember[i] = j$ indicates that i is a member of class j).

CSize stores the sizes of the classes being utilized. Finally, $CEmpty$ indicates empty classes. All of the above arrays are of type `integer`. Other arrays are used to deal with families.

The arrays *FStart*, *FNext*, *FPrev*, and $FMember$ serve the same function as with those of similar names (the above "C"-prefixed arrays) in the context of classes. They also are of type `integer`.

Further arrays are used for the process of refinement. $Refine$ indicates which elements (through index) should be moved to which classes (through stored value). *RStack* is used as a stack, keeping track of those positions of $Refine$ that are occupied. *Sel* is used to queue the members of all of the small[1]

[1]A "small" class is one whose members together number less than one of the classes that has been determined to be the "large" class. This in turn is a class containing a number of

classes. Finally, *Sc* points to the last member of each of the small classes. As with all of the previously described arrays, those just mentioned are of type `integer`.

Finally, several arrays are used to implement the gap function. *Gap* indicates the first element of a gap list for a given index. *GMember* indicates that a given index value belongs to the stored value's gap list. That is, $GMember[i] = j$ implies that i belongs to the j gap list. *GNext* and *GPrev* are used to emulate forward and reverse links in a list of gap values. Once again, all arrays just discussed are of type `integer`.

During the construction of the first program refinement (`crochB1`), several of the arrays discussed above were replaced by functions, to reduce memory usage.

The function *GMember*, used in the same way as was the array of the same name, returns `null` if the passed argument is either a member of no class, or is the first member of a class. Otherwise, the class of which it is a member is returned, by returning $CPrev[i]$, where i is the passed argument. The introduction of function *GMember* to replace the array of the same name reduced the memory usage of the program from $19n$ to $18n$ integers.

Next, in `crochB3`, further improvement to memory usage was made through the introduction of function $FMember$, again to replace the functionality of the array of the same name. The memory requirement was reduced through this to $17n$ integers. The definition of $FMember$ is such that if a given pointer is null for argument i, it returns the value $FStart[i]$; if $i \leq$ the stack pointer, $FNext[FPrev[FStart[i]]]$ is returned; $FStart[i]$ is returned otherwise. Finally,

elements that is not less than that of any other class among its siblings.

FEnd is defined such that $FEnd(i) = FPrev[FStart[i]]$.

Further refinement was made with the introduction of `crochB4` by way of causing the arrays $CEmpty$ and *CStart* to reside within the same memory segment as one another. This reduced the memory need further to $16n$ integers.

In the next version of the program (`crochB5`), things are arranged such that information previously stored in *CEnd* and *CSize* could be derived from *CStart*, *CNext*, and *CPrev*. As a result of this, the space requirement was reduced to $14n$ integers. After the modification, the information that was "moved" became accessible through the following facts: $CEnd(i) = CPrev[CStart[i]]$; $CSize(i) = CNext[CPrev[CStart[i]]]$.

In the revision represented by `crochB6`, through the reduction of the possible length of an input string from `UNSIGNED LONG MAX` to `LONG MAX`, guarantee was given that adequate space be available such that $CMember$ could be virtualized over the arrays *GNext*, *Gap*, and *GPrev*. The elimination of $CMember$ reduced the memory size requirement of the program was reduced to $13n$ integers.

The most recent revision of the modified Crochemore's program, rather than representing changes in memory requirements (which currently remain at $13n$ integer), instead was the result of refinements of the code used to perform the program's tasks. This latest version is known as `crochB7`, and is one of the three programs utilized for testing in the context of this writeup.

Chapter 4

A brief description of runFinder

In this chapter, there is presented a brief description of the underlying algorithm utilized by the runFinder program. It is comprised of several algorithms, each of which (excepting the final) supplies the next stage with input toward the goal of calculating runs within the initial input of a string.

4.1 Linear-time Runs Calculation

The utilized portion of the Run Finder program employes the well-known linear time method for finding runs within a string on a general alphabet. The general procedure for this can be summarized to the following steps.

1. Compute the input string's suffix array (hence *SA*)
2. Use SA and compute the longest common prefix (array) (hence *LCP*)
3. Use SA along with LCP to compute the Lempel-Ziv factorization (hence *LZ*) of the input string
4. Use Main's algorithm with LZ as input to compute all left-most runs within the original string

5. Use Kolpakov and Kucherov's algorithm with LZ as input to compute the remainder of the runs within the original string.

Below, some particulars of each step will be expanded upon.

4.1.1 Computation of the Suffix Array

The structure of the suffix array (which was described by U. Manber and E. W. Meyers in 1993[15]) is such that it can keep track of the beginning position of each of the suffixes of a given string. The array is sorted in such a way that the suffixes indicated by it are in order. That is, if indices 1 and 3 are stored in that order within SA, the suffix indicated by index 1 is lexicographically less than that indicated by index 3. It is the achievement of this ordering that is the bulk of the work toward the task of constructing SA of a given string, and this can be done in $\Theta(n)$ time, with respect to the number of strings to be sorted.

It was not until 2003 that the initial algorithms for computing SA appeared that were also in linear time, which of course is a requirement toward run calculation in linear time. The first of these were documented in [10, 11, 9], with those documented in [11] and [9] proving to be feasible for practical use.

4.1.2 Longest Common Prefix Array

The second component involved in the linear-time calculation of runs of a string under discussion is the Longest Common Prefix array. Using the symbol $lcp(i, j)$, let its function be defined at points i and j to be the longest common prefix of the suffixes whose starting positions are respectively located at positions i and j in a given string $x[1 \dots n]$. The LCP array is computed such that for a position i, $LCP[i] = lcp(SA[i-1], SA[i])$ (where SA is the suffix array of the string under consideration).

It is clear that were LCP to be computed in the obvious way, this procedure would be of $O(n^2)$ time. However, as is the requirement of the overall task at hand, methods for finding LCP of a given input string exist that are of linear time.

4.1.3 Lempel-Ziv Factorization

The Lempel-Ziv Factorization of a string was introduced in 1978 by Abraham Lempel and Jacob Zivfor purposes toward data compression. However, it has also been found to yield information regarding periodicity within a string.

Once factorization has occurred, a string x may be viewed as a concatenation of its factors $u_1, u_2, \ldots, u_n$ such that the following holds. If $x = u_1 \ldots u_k$, then each u_i from $u_1, \ldots, u_{k-1}$, and possibly including u_k, u_i is the longest prefix of $u_i \ldots u_k$ that also occurs earlier in x, or is one character. As an example, consider the following string: *abbaabbabaaabab*, which has the LZ factorization $a, b, b, a, abb, baa, ab, ab$[13].

A method by which LZ may be computed was introduced in a paper by Crochemore, Ilie, and Smyth[13] (see pseudo-code listing "Algorithm 2"). This method involves the computation of an array called the *longest previous factor* array (hence *LPF*). As its name might suggest, LPF keeps track of information regarding the longest previous (Lempel-Ziv) factor in a string with respect to a given position within it. It is then one step from the computation of LPF to LZ. Additional to LPF, information that will be useful to a later step in the overall process of finding runs is kept in a structure called *PrevO*. This array will keep track of the start position of the previous occurrence of a factor with respect to a given position, indicated by the array's index. The computation of both PrevO and LPF is showing in pseudocode listing "Algorithm 1."

LPF is defined as an array with respect to string x such that

$$LPF[i] = max\{l|x[i \ldots i+l-1] \textit{ is a factor of } x[0 \ldots i+l-2]\} \cup \{0\}) \quad (4.1.1)$$

Algorithm 1: 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 2: Lempel-Ziv factorization[17]

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

4.1.4 Main's Algorithm

The next step in the calculation of runs within a string is implemented through an algorithm by Main [14] that calculates leftmost runs of the string (see pseudo-code listing "Algorithm 3"). This is done using LZ as input. To discuss this further, some new concepts are introduced, below.

Considering the LZ factorization of a string x, let those runs that are within one factor be said to be in *Class 1*. Let those runs that cross a boundary between factors be said to be in *Class 2*.

Main's algorithm considers a string, composed of blocks of factors such that the string $x = w_1 \dots w_2$, where each w_i is a block of LZ factors. The algorithm works by considering for the value h between 2 and k, the substring t_h, which is defined to be the substring preceding the block w_h and is of length $2|w_{h-1}\, w_h|$, at most[1]. Within the substring $t_h\, w_h$ can be calculated its right- and left-max runs (those that are adjacent to the boarder of two factors). These values are then investigated, and if it can be seen that a given run of this type extends beyond the border of a neighbouring factor, it is a run of Class 2. Once this procedure is completed with respect to a string, all Class 2, and thus all leftmost runs, are found. The process of doing this over the whole string x

[1]The bound $2|w_{h-1}\, w_h|$ is taken from a paper by Main: [14]

being of $\Theta(n)$ running time[14].

Algorithm 3: Main's algorithm[17]

```
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,(LZE[i-1]+p),l) 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, LZE[i-1]+l+p, l) to LinkedList[LZE[i-1]+p]
    end if
  end for
end for
```

4.1.5 Kolpakov and Kucherov's Algorithm

The final step in the process of finding all runs within a string involves the utilization of an algorithm by Kolpakov and Kucherov[12]. The algorithm is fairly straight forward:

- Begin observation of a LZ factor that has a previous neighbour
- For each Class 2 run occurring within it, look at its previous occurrences

- If a given previous occurrence falls within the block, it is a Class 1 run.

Since the union of the collections of Class 1 and 2 runs comprises all runs within a string, the collection of runs is now complete. The pseudo-code for the above procedure is listed in “Algorithm 4”.

Algorithm 4: Kolpakov and Kucherov’s algorithm[17]

```
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 $j \in LinkedList[i - delta]$ 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
```

Chapter 5

The experimental setup

The mechanism and relevant particulars of the experiments performed are presented. Included are discussions of the methodology utilized, including considerations related to some of its particulars; test data are referenced; and the platform on which the experimentation took place is listed.

5.1 Test Mechanism

In order to ascertain a reasonable and fair measurement of performance of the programs in question, a method was needed to determine the running time of each. It would be desired to exclude from measurement those operations called for by the relevant parts of the program being test that themselves had no direct bearing (other than, for example, the provision of resources such as memory) on the actual work being performed by the program with respect to runs calculation. In addition, given that the programs were run within a multitasking environment, a method that would exclude much of the direct influence (at least with regard to working clock cycle consumption) of programs external to those being tested would be most desirable. Given these two key motivators, the method provided through the facilities given by the `time.h` library of the `C++` programming language were utilized. Specifically, those facilities that provide access to an approximation of clock cycles utilized by the program up to a specified point were included in the testing framework.

The mentioned library presents the `clock()` function, which reports an approximation of clock cycles utilized form the beginning of execution of the program in which it is called until the calling of that function. Its return type is `long int`, and utilizing the difference of two values returned by calling `cloc()` at different times can provide a measure of consumed clock cycles over a given time period. These values in conjunction with the constant `CLOCKS_PER_SEC` were used to calculated milliseconds of actual time represented by the number of clock cycles utilized. That is, were the program running in an environment within which only its relevant operations were carried out, it would take the amount of time calculated to perform its task.

5.2 Test Data

The programs were made to run each on the same sets of data, chosen beforehand. These sets were constituted by collections of data representing:

1. Samples of English language texts
2. Strings representing DNA segments
3. Fibonacci string segments
4. Random strings

Each of the data sets above was represented by collections of strings of specific and varied lengths. It was chosen to supply to the programs strings of the above sorts of lengths beginning at length 50 and extending through length 140 000. The results are represented graphically as split into two groups: big and small (about the 7500-string mark).

5.3 Test Particulars – Repeated Test Runs

As a result of the manner through which the runtime information is presented, through the `time.h clock()` function), shorter runtimes are not reported before a certain point. That is, if an execution of code utilizes few enough clock cycles, its activity will not register (as a result of the nature of the approximation gone about in the supplying of the reported utilized clock cycles). This problem was encountered during the processing of shorter strings during the investigations for this thesis. A method by which this issue might be worked with is through the repeated execution of the relevant run-calculating code for each of the shorter strings. The clock cycles used for `all` of these executions together is reported, and this value is then divided by the number of executions gone about on a given string. Thus, the resultant value represents an approximation of the number of clock cycles utilized for that string whose runs had just been calculated.

5.4 Test Platform

- All programs were compiled using Microsoft Visual C++ 2010 Express compiler

- All experiments were ran on a stand-alone PC with Intel(R) Core(TM)2 Duo CPU E7400 @ 2.80GHz processor
- The machine had 2.00 GB memory (1.87 GB usable)
- Under the operating system Windows 7 Professional

Chapter 6

Results

The input data files used in the experiments are listed below. Filenames of input and result data are listed below. In the case of input data files, the nature of the data begin's the file's name, followed by the length of the string contained within the file, and then the file type extension (*txt*). With respect to results data, the name of the program under observation begins the filename, followed by the word *results*, then the category of input data used to produce the results, followed by the size (in characters) of that input data, and finally the file type extension (*txt*).

- dna.50.txt, $\cdots$, dna.140000.txt
- english.50.txt, $\cdots$, english.140000.txt
- fibo.50.txt, $\cdots$, fibo.140000.txt
- rand.50.txt, $\cdots$, rand.140000.txt

and the files with the results for `crochB`

- crochB.results.dna.50.txt, $\cdots$, crochB.results.dna.140000.txt

- crochB.results.englis.50.txt, $\cdots$, crochB.results.english.140000.txt
- crochB.results.fibo.50.txt, $\cdots$, crochB.results.fibo.140000.txt
- crochB.results.rand.50.txt, $\cdots$, crochB.results.rand.140000.tx

and for `crochB7`

- crochB7.results.dna.50.txt, $\cdots$, crochB7.results.dna.140000.txt
- crochB7.results.englis.50.txt, $\cdots$, crochB7.results.english.140000.txt
- crochB7.results.fibo.50.txt, $\cdots$, crochB7.results.fibo.140000.txt
- crochB7.results.rand.50.txt, $\cdots$, crochB7.results.rand.140000.txt

and for `runFinder`

- rF.results.dna.50.txt, $\cdots$, rF.results.dna.140000.txt
- rF.results.englis.50.txt, $\cdots$, rF.results.english.140000.txt
- rF.results.fibo.50.txt, $\cdots$, rF.results.fibo.140000.txt
- rF.results.rand.50.txt, $\cdots$, rF.results.rand.140000.txt

These files can be found at `http://www.cas.mcmaster.ca/~fullerrc/`

For consideration of the results of the tests performed, several key aspects were looked at. The primary objectives of this thesis were centred about observing the point at which it would become advantageous to use one program over another for runs calculation (with respect to input string length); investigating the question of the behaviour of the programs in operation in comparison to the theoretical bounds on their performance; and relating their theoretical and actual performances mathematically through the observation of a constant between these two values.

Some differences were observed between the performance of a given program when processing certain of the utilized datasets (please see below for particulars). Specifically, Fibonacci strings seemed to clearly show the Crochemore programs to consume more time than had been expected: of the order of n^2 time with respect to input string length. This is a consistent issue through the string types considered (with only one exception in the case of the processing of Random string: see Table 6.5). It is possible that it is the highly periodic nature of the Fibonacci strings that causes this behaviour to be clearly seen. Please see below for a more detailed discussion.

6.1 Overall Trends

Though particulars of the performance results of each of the programs and how they relate to one another vary between datasets, there are some trends that appear mostly consistent[1].

For shorter strings, as defined by the division of tables in Tables 6.6 – 6.13, the Crochemore programs initially outperform runFinder. However, within certain ranges of string lengths (the particular point varies per dataset, see below for more detail), runFinder's performance exceeds that of the Crochemore programs, and the trend is not reversed within the domain of observed data. The observations from longer strings continues this trend: runFinder begins[2]

[1]With the exception of the results based on the processing of the Fibonacci strings used.

[2]"Begins" with respect to the domain of observation restricted to long strings, but the observations are actually continued from short strings.

Dataset	crochB / runFinder	crochB7 / runFinder
DNA	2000 – 2500	1000 – 1500
English	2000 – 2500	2000 – 2500
Fibonacci	150 – 200	250 – 300
Random	2000 – 2500	1000 – 1500

Table 6.1: Performance Intersection with runFinder

outperforming the Crochemore programs, and the trend does not reverse.

6.2 Intersection of Performance

As mentioned above, the point at which runFinder begins to outperform the Crochemore programs varies per dataset. For this reason, the listing of the points of intersection of the runtimes for each program are given in Table 6.1, divided by dataset.

As can be seen above, the point at which runFinder begins to outperform the Crochemore programs is somewhat consistent, but shows some variance between datasets.

6.3 Curve-Fitting

Because of the existence of certain expectations with respect to the performance of the programs under investigation, it was possible to examine some modelling of these expectations in comparison with actual performance. The methods by which these observations were gone about revolved around use of the gnuplot [8] program and its data analysis capabilities to perform curve-fitting and graphing. A tutorial [7] was utilized to provide information regarding particulars of the fitting process as it related to the gnuplot program. Considering the

Program	Function	Constant	Asymptotic Standard Error
crochB	*nlogn*	0.000146017	+/- 4.444e-06 (3.044%)
crochB	n^2	1.2256e-08	+/- 1.385e-10 (1.13%)
crochB7	*nlogn*	0.000133007	+/- 3.524e-06 (2.65%)
crochB7	n^2	1.1081e-08	+/- 1.731e-10 (1.562%)
rF	n	0.00037505	+/- 1.281e-06 (0.3415%)

Table 6.2: DNA

Program	Function	Constant	Asymptotic Standard Error
crochB	*nlogn*	0.000143196	+/- 3.892e-06 (2.718%)
crochB	n^2	1.089e-08	+/- 1.388e-10 (1.274%)
crochB7	*nlogn*	0.000125915	+/- 3.197e-06 (2.539%)
crochB7	n^2	9.56447e-09	+/- 1.053e-10 (1.101%)
rF	n	0.000299246	+/- 5.699e-06 (1.904%)

Table 6.3: English

polynomial behaviour of crochB (and crochB7) that becomes visible during the processing of the Fibonacci strings used for testing, the fitting of $O(n^2)$ functions was attempted with respect to processing time result data for the remainder of the datasets. The results indicated that the best fit for nearly all those data not clearly reflecting performance of either $O(n^2)$ or $O(n\ log\ n)$ for crochB and crochB7 lies between $O(n^2)$ and $O(n\ log\ n)$ (please see Tables 6.2 – 6.5 for comparison of fittings, as well as graphs given in Tables 6.14 – 6.25), with the exception being seen through the processing of Random strings (Table 6.5), where crochB7's performance remains $O(n\ log\ n)$.

6.4 Graphs

Graphical representations of the results data are presented.Graphs were divided between "short" and "long" strings. The motivation for this division was to

Program	Function	Constant	Asymptotic Standard Error
crochB	$nlogn$	0.0830115	+/- 0.003101 (3.735%)
crochB	n^2	8.80837e-06	+/- 1.332e-08 (0.1512%)
crochB7	$nlogn$	0.0363081	+/- 0.001448 (3.989%)
crochB7	n^2	3.86534e-06	+/- 1.99e-08 (0.5147%)
rF	n	0.000510317	+/- 1.678e-06 (0.3287%)

Table 6.4: Fibonacci

Program	Function	Constant	Asymptotic Standard Error
crochB	$nlogn$	0.000147729	+/- 3.549e-06 (2.402%)
crochB	n^2	1.5339e-08	+/- 2.55e-10 (1.662%)
crochB7	$nlogn$	0.000136875	+/- 1.896e-06 (1.386%)
crochB7	n^2	1.39936e-08	+/- 3.595e-10 (2.569%)
rF	n	0.000447371	+/- 1.72e-06 (0.3845%)

Table 6.5: Random

provide adequate views of each of the representations with respect to relative string length. Given that the step size between tested string lengths varied (from 50 to 1000), and performance from one end of the string length range to the other varied significantly, combining the results' graphs would have caused the more finely grained aspects of the short strings' graphs to be overshadowed by those of the large strings' graphs. The combination is therefore avoided when observing for significant points (intersections of performance values of programs). In other figures (Tables 6.14 – 6.25), the combination is made and used in the calculation of constants relating theoretical runtime bounds and observed data, and then in the graphical representation of these both.

As a further note, distinct (but minor) discontinuities can be observed at points $x = 10$ and $x = 6$ (respectively) of the short and long string graphs, in the case of each of the datasets. This is a result of a change in step size within the window of observation presented through those particular graphs.

DNA chart (time units are in miliseconds)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
length	50	100	150	200	250	300	350	400	450	500	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500	6000	6500	7000	7500
crochB	0.0	0.0	0.1	0.1	0.1	0.1	0.1	0.2	0.2	0.2	0.5	0.9	1.2	1.6	1.9	2.4	3.1	3.6	3.9	4.7	5.0	6.0	7.2	8.1
crochB7	0.0	0.0	0.1	0.1	0.1	0.2	0.2	0.2	0.3	0.3	0.6	1.0	1.4	2.2	2.2	2.8	3.2	3.5	4.1	4.9	5.9	6.0	6.6	10.2
runFinder	0.3	0.4	0.4	0.4	0.4	0.5	0.4	0.5	0.5	0.5	0.7	1.1	1.4	1.7	1.6	2.0	2.2	2.3	2.6	2.4	3.0	3.0	2.9	3.7

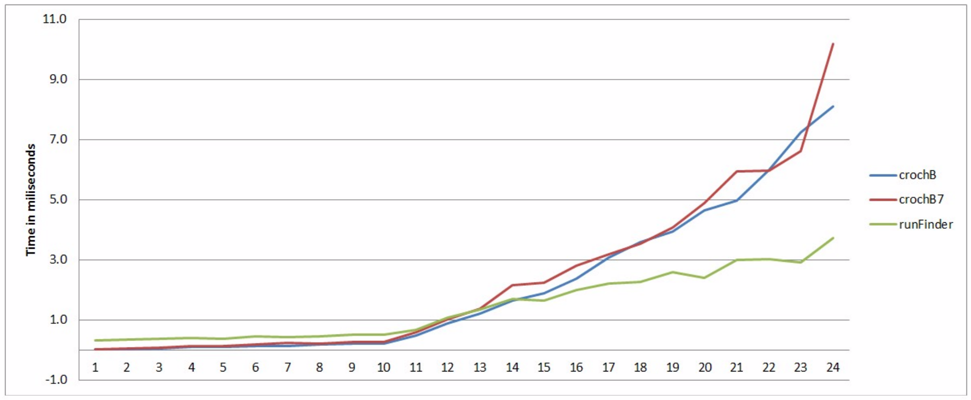

Table 6.6: DNA Data – Short Strings

Although it does take away from some of the (relative) smoothness of the graphs the points at which this occurred are not significant with respect to important events (intersections, notable points of increase in runtimes) of the result data. As such, the discontinuities are left in, with this note explaining them.

6.5 Comparison With Theoretical Runtimes

The consideration of the performance of the observed programs in comparison with that which was theoretically expected yielded for the most part little deviation. However, as mentioned, particularly in the case of strings that displayed high periodicity (Random and Fibonacci), crochB and crochB7 performed of order n^2 (please see Tables 6.23, 6.24, 6.20, and 6.21). Additionally, similar behaviour (performing between $O(n^2)$ and $O(n\ log\ n)$) was seen in crochB7 all but Random strings, where $O(n\ log\ n)$ performance was more clearly observable.

DNA chart (time units are in miliseconds)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
length	7500	8000	8500	9000	9500	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000	110000	120000	130000	140000
crochB	8.1	7.7	8.5	9.0	8.2	9.3	19.9	31.0	41.6	52.4	64.2	76.0	91.3	106.3	123.1	143.6	174.1	197.5	227.4
crochB7	10.2	9.5	9.8	11.1	11.2	12.5	24.5	32.1	42.8	53.7	65.5	79.4	94.6	104.1	119.1	136.1	155.1	183.4	203.0
runFinder	3.7	3.3	3.8	3.7	4.2	4.0	8.3	11.7	14.6	18.4	21.5	26.4	29.2	33.6	37.0	40.6	44.5	48.3	52.5

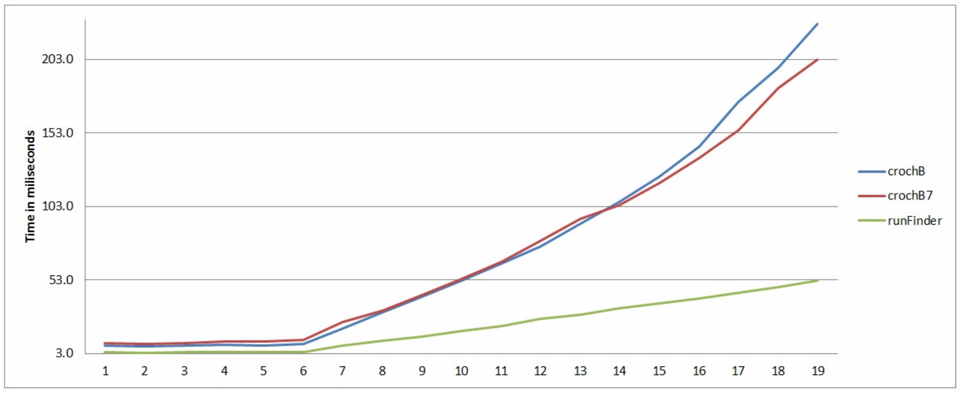

Table 6.7: DNA Data – Long Strings

English chart (time units are in miliseconds)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
length	50	100	150	200	250	300	350	400	450	500	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500	6000	6500	7000	7500
crochB	0.02	0.03	0.05	0.07	0.09	0.11	0.13	0.15	0.15	0.19	0.37	0.67	0.96	1.30	1.73	1.92	2.56	2.85	2.93	3.60	4.46	4.42	5.24	6.50
crochB7	0.01	0.03	0.05	0.07	0.09	0.11	0.13	0.17	0.18	0.23	0.43	0.75	0.94	1.33	1.77	1.98	2.87	2.82	3.15	3.74	4.42	5.16	5.76	5.92
runFinder	0.34	0.35	0.38	0.38	0.39	0.42	0.39	0.42	0.45	0.45	0.55	1.07	1.16	1.10	1.47	1.26	1.73	1.64	2.01	1.84	2.24	2.54	1.98	2.24

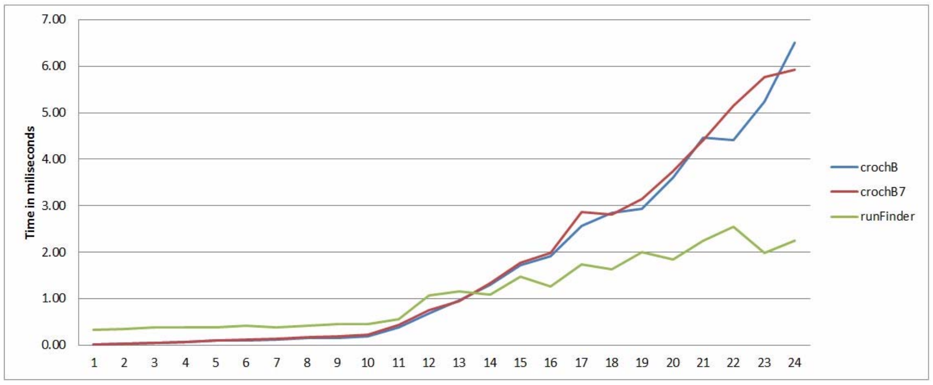

Table 6.8: English Data – Short Strings

English chart (time units are in miliseconds)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
length	7500	8000	8500	9000	9500	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000	110000	120000	130000	140000
crochB	6.50	6.32	7.14	6.86	7.40	7.90	16.86	25.26	34.44	45.56	55.34	67.44	82.76	97.60	116.42	131.48	154.54	188.74	224.20
crochB7	5.92	6.14	6.60	7.46	7.70	8.16	16.16	25.08	34.52	47.30	56.42	66.60	83.48	95.38	107.32	122.96	140.22	163.54	181.30
runFinder	2.24	3.06	2.24	3.70	2.68	2.86	5.74	8.40	10.62	12.20	15.74	16.70	20.68	22.98	25.42	28.70	31.96	34.54	37.70

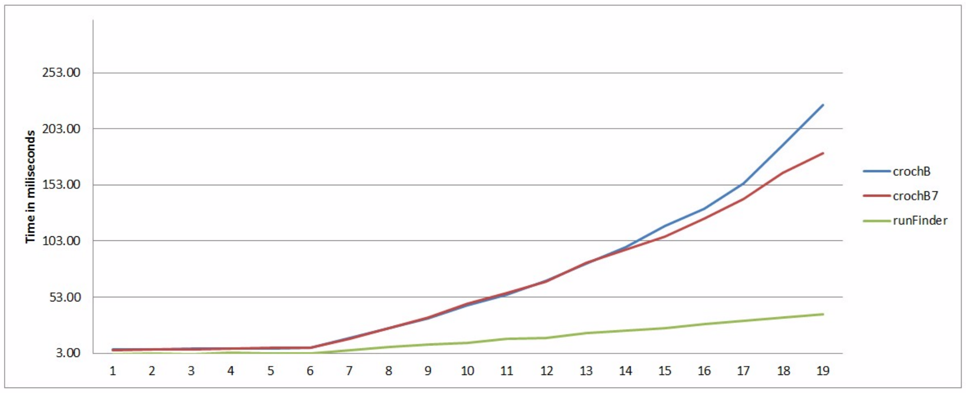

Table 6.9: English Data – Long Strings

Fibo chart (time units are in miliseconds)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
ngth	50	100	150	200	250	300	350	400	450	500	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500	6000	6500	7000	7500
ochB	0.0	0.1	0.2	0.4	0.6	0.9	1.1	1.5	1.8	2.2	8.4	18.8	33.7	50.8	74.6	99.6	131.9	168.1	208.4	248.2	296.4	349.5	408.1	475.0
ochB7	0.0	0.1	0.1	0.2	0.2	0.4	0.4	0.5	0.7	0.8	2.2	5.4	9.2	13.2	20.2	28.8	38.2	49.9	67.0	85.5	105.9	124.7	142.5	169.2
ınFinder	0.3	0.4	0.4	0.4	0.4	0.4	0.4	0.5	0.5	0.5	1.0	1.3	1.5	1.7	1.8	2.3	2.4	2.6	3.2	2.9	3.5	3.5	4.2	3.6

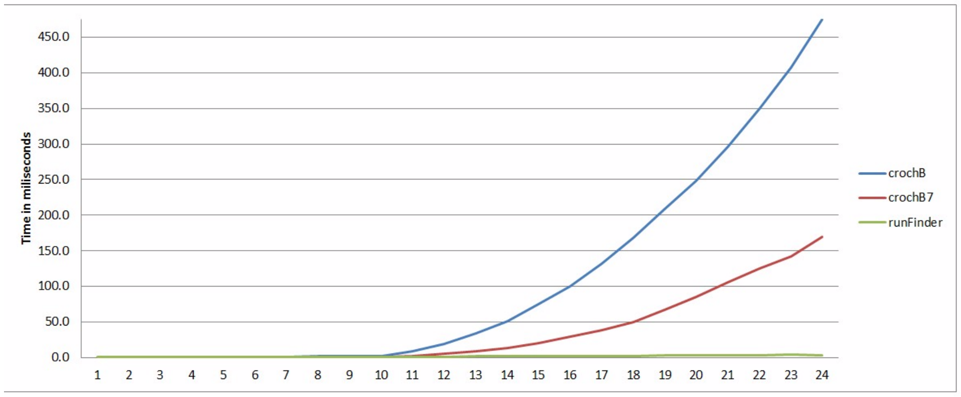

Table 6.10: Fibonacci Data – Short Strings

Fibo chart (time units are in miliseconds)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
length	7500	8000	8500	9000	9500	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000	110000	120000
crochB	475.0	534.6	597.5	674.2	753.5	831.6	3468.5	7824.5	13746.1	21989.3	31093.5	42753.9	57033.6	71333.4	87375.1	105246.0	125407.0
crochB7	169.2	197.7	229.1	265.4	302.6	328.0	1389.6	3170.5	5932.9	8772.3	13517.2	17918.1	22889.4	30213.0	39344.0	47354.6	52808.7
runFinder	3.6	4.4	4.3	4.9	4.7	5.3	10.4	14.7	19.6	25.1	28.8	35.0	40.4	45.7	51.0	55.9	61.7

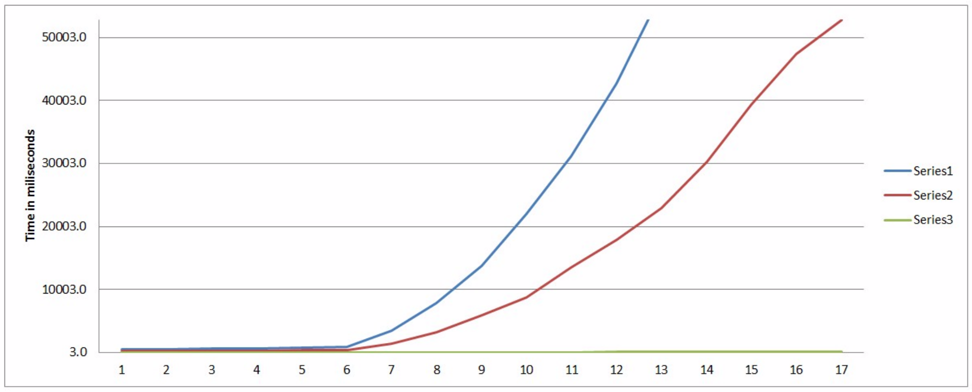

Table 6.11: Fibonacci Data – Long Strings

DNA chart (time units are in miliseconds)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
length	50	100	150	200	250	300	350	400	450	500	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500	6000	6500	7000	7500
crochB	0.0	0.1	0.1	0.1	0.1	0.2	0.2	0.2	0.2	0.3	0.7	1.1	1.5	1.9	2.3	2.8	3.3	3.8	4.3	5.0	6.3	6.6	7.1	7.9
crochB7	0.0	0.1	0.1	0.1	0.1	0.2	0.2	0.3	0.3	0.3	0.7	1.2	1.7	1.9	2.8	3.3	4.4	4.9	5.6	5.1	5.7	6.3	8.5	8.8
runFinder	0.3	0.4	0.4	0.4	0.4	0.5	0.5	0.5	0.6	0.5	0.9	1.1	1.6	1.7	1.9	2.2	2.3	2.6	2.9	3.3	3.6	3.4	3.3	4.6

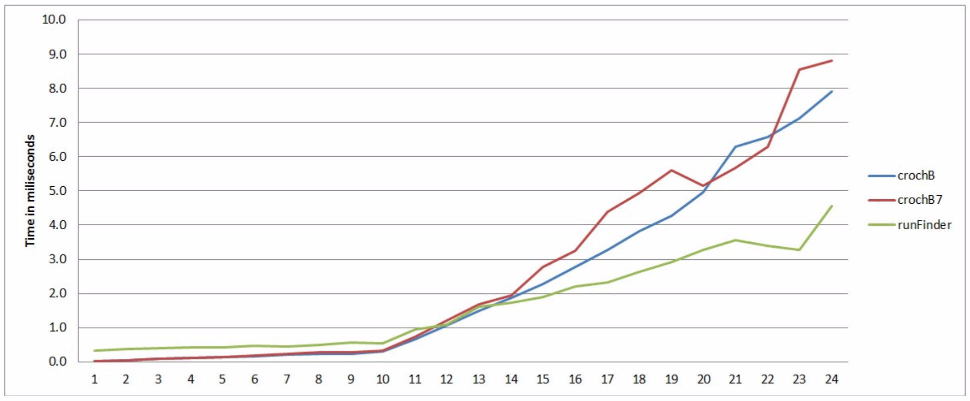

Table 6.12: Random String Data – Short Strings

Random chart (time units are in miliseconds)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
length	7500	8000	8500	9000	9500	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000	110000	120000	130000	140000
crochB	7.9	8.1	8.5	9.1	9.7	10.6	21.5	34.0	47.9	61.2	75.6	91.3	111.3	129.8	152.7	177.0	208.5	253.8	299.4
crochB7	8.8	9.1	10.3	11.9	10.7	12.0	25.2	40.6	55.2	66.4	78.0	94.7	113.7	126.4	144.9	166.2	194.0	223.4	256.9
runFinder	4.6	4.1	4.7	4.9	5.1	5.4	9.7	14.0	17.4	22.4	26.5	31.3	35.7	40.4	45.0	48.9	53.5	58.0	62.5

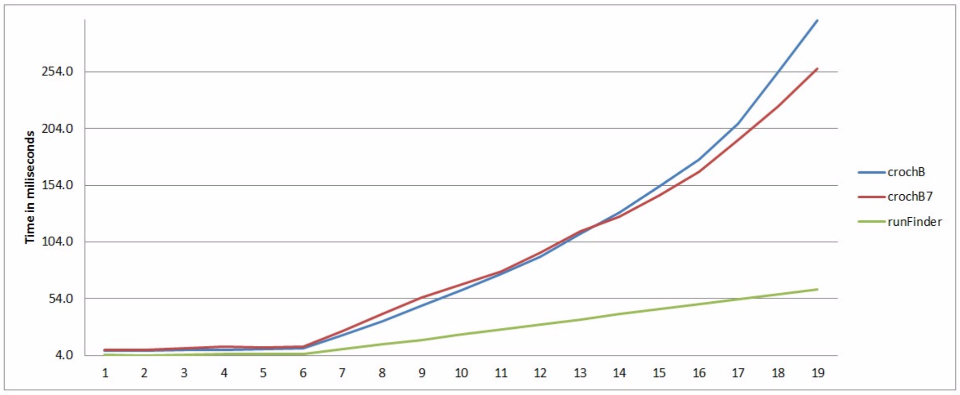

Table 6.13: Random String Data – Long Strings

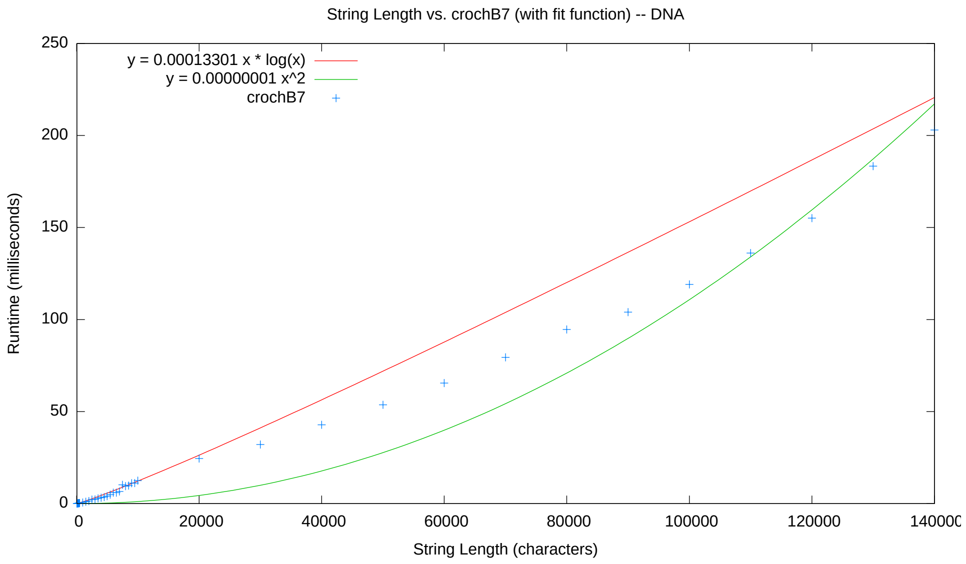

Table 6.14: DNA — crochB7

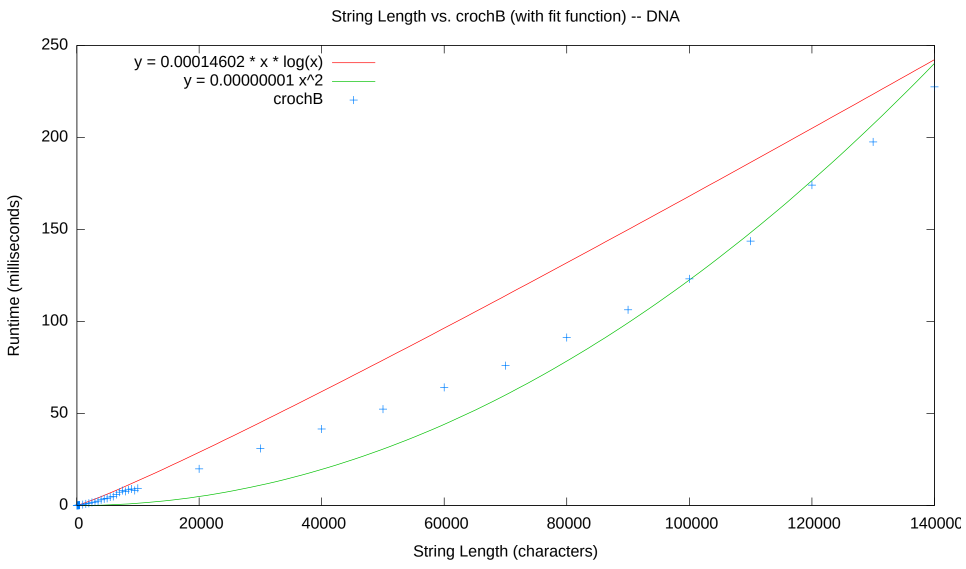

Table 6.15: DNA — crochB

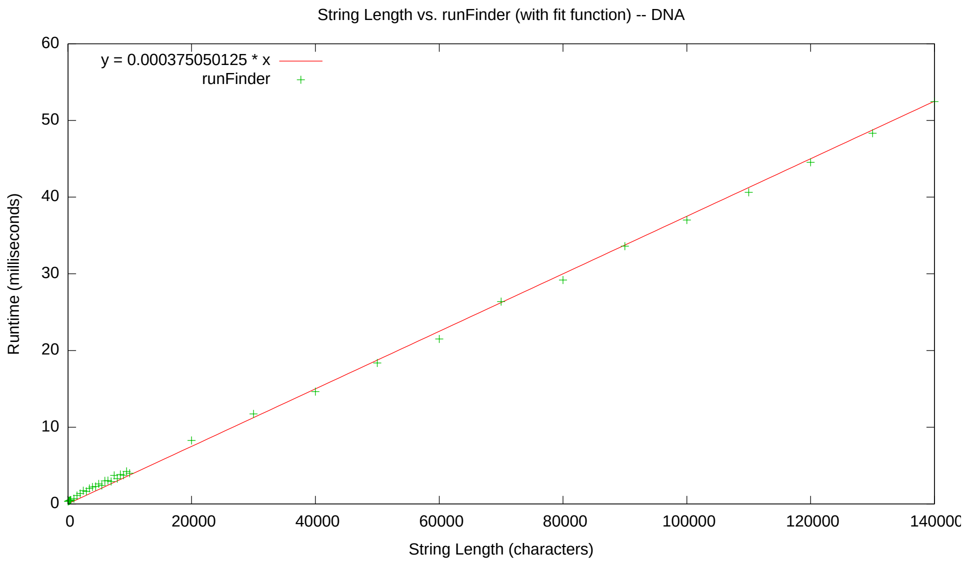

Table 6.16: DNA — runFinder

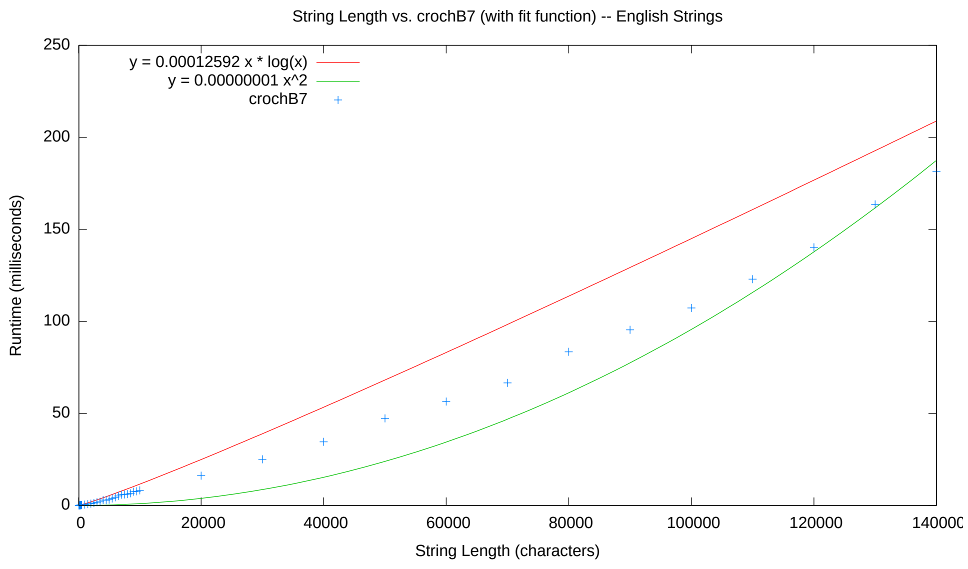

Table 6.17: English — crochB7

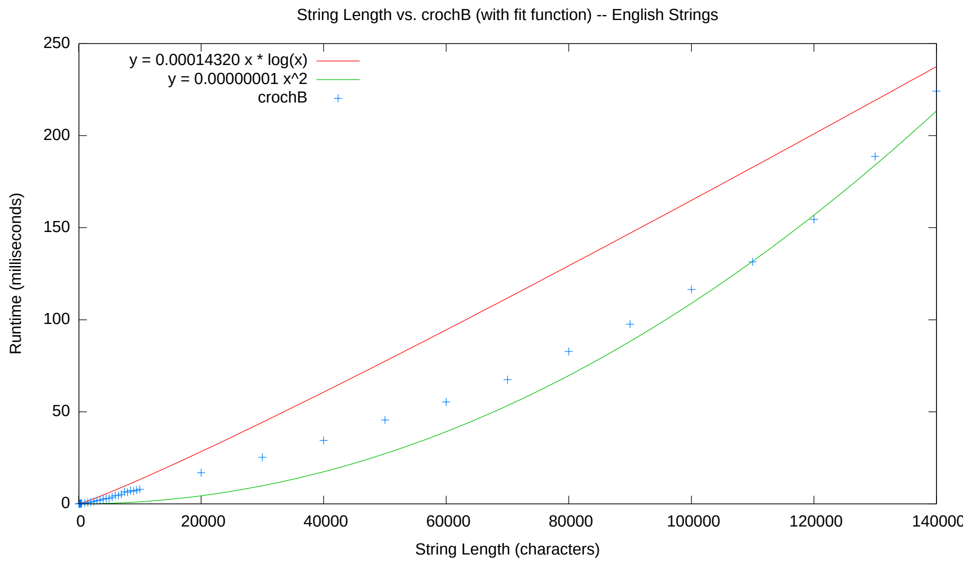

Table 6.18: English — crochB

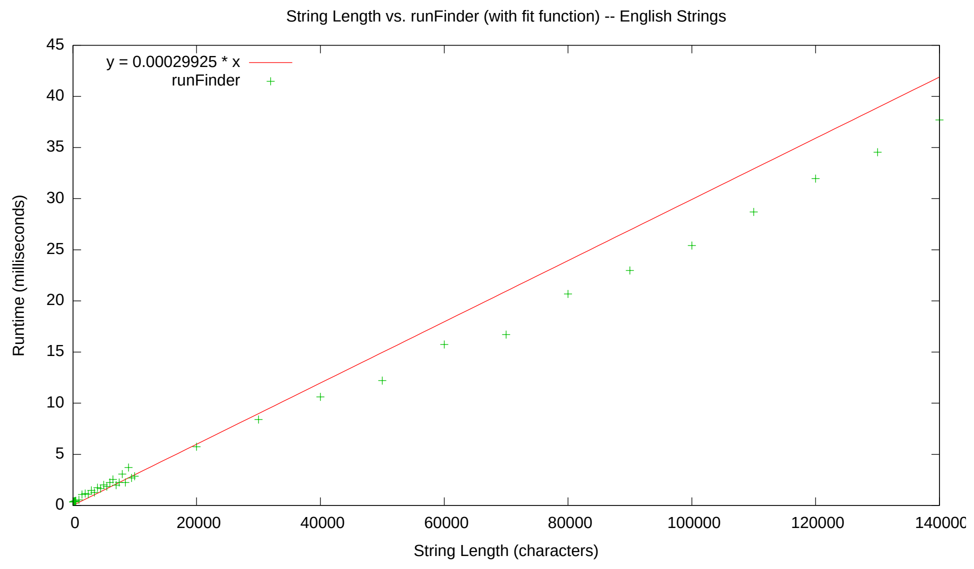

Table 6.19: English — runFinder

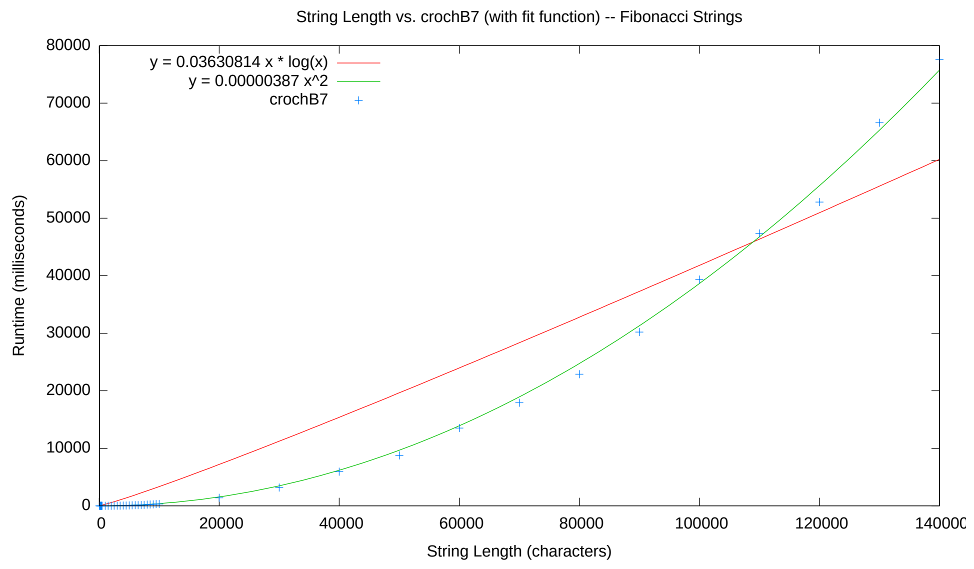

Table 6.20: Fibonacci Strings – crochB7

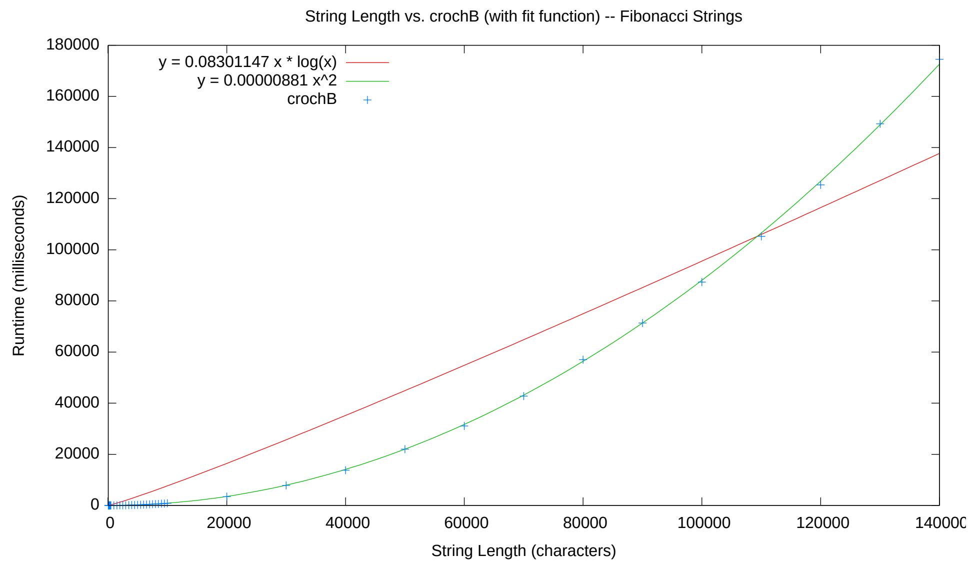

Table 6.21: Fibonacci Strings – crochB

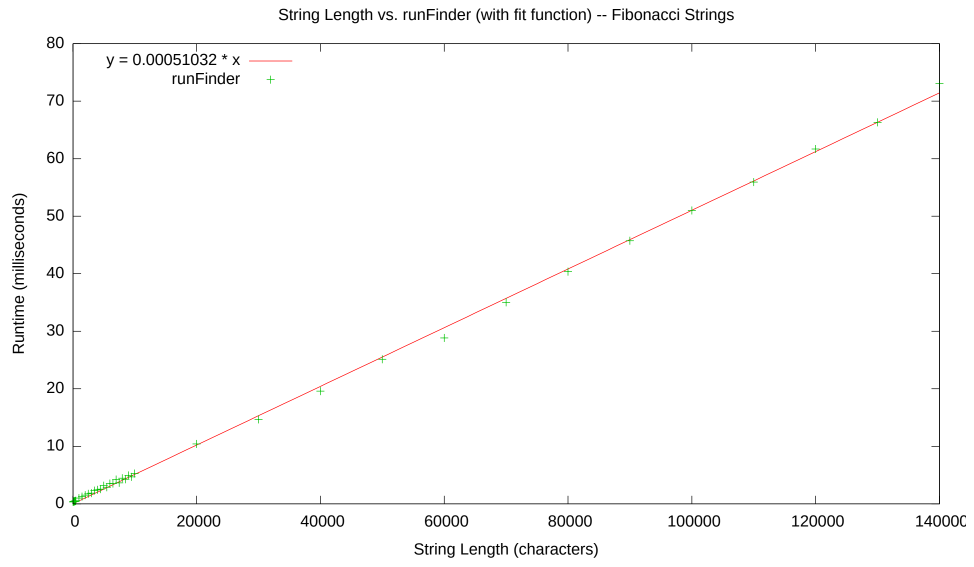

Table 6.22: Fibonacci Strings – runFinder

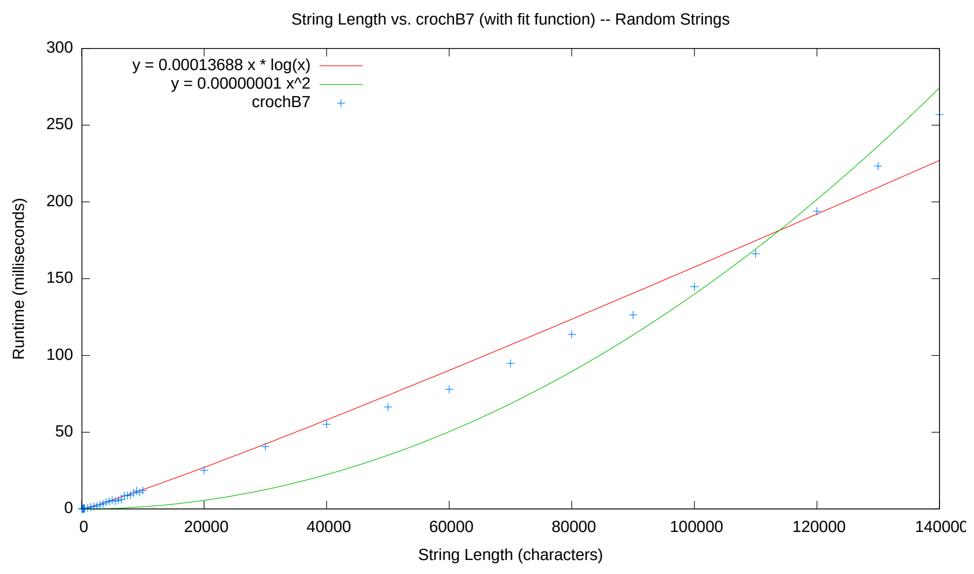

Table 6.23: Random — crochB7

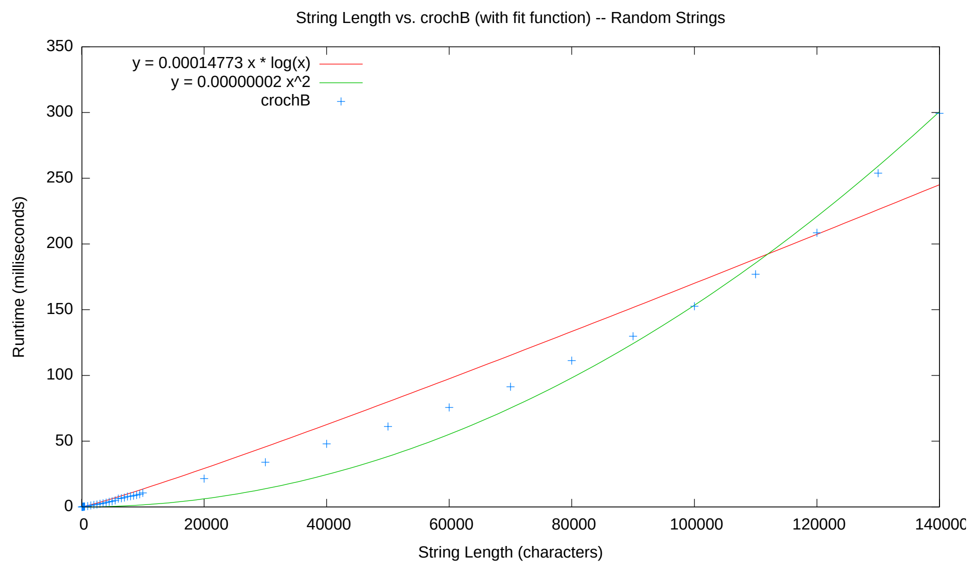

Table 6.24: Random — crochB

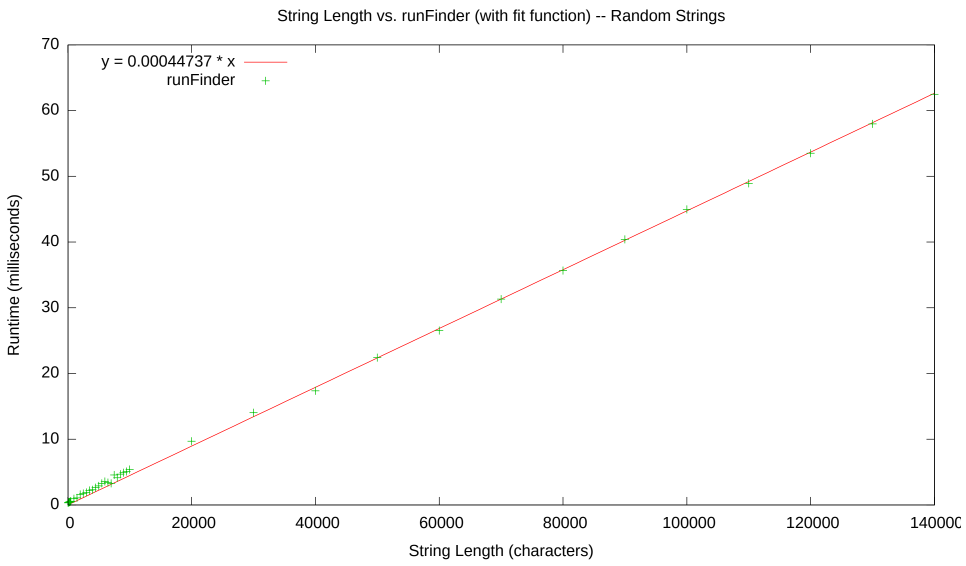

Table 6.25: Random — runFinder

Chapter 7

Conclusions and Future Work

A number of expected, and some unexpected observations came from the work done toward the completion of this thesis.

7.1 General Observations

It was observed that indeed runFinder did surpass crochB and crochB7 in performance with respect to runtime, and a range of points with respect to string length have been identified within this investigation at which this occurs; the shape of the data generated by test runs appears as it should in most cases, considering the theoretical runtimes of each of the programs, runFinder performing in a linear manner with respect to input string length, and the Crochemore programs in a manner between $O(n^2)$ and $O(n\ log\ n)$, and in the case of Random strings, $n\ log\ n$ for crochB7. Highly periodic strings seem to have proven useful at bringing to light some pathological behaviours (considering the very clear $O(n^2)$ appearance of the result data for crochB and crochB7 when processing Fibonacci strings). This notion brings forward one of the unexpected results observed: that of crochB and crochB7 performance in extreme circumstances (which seems to be representative of its performance in other situations, but less clearly observable). Also, the non-observation of the outperformance of crochB by crochB7 was somewhat unexpected, but reconcilable with some considerations (as discussed below).

7.2 Performance of crochB vs. crochB7

Given that crochB and crochB7 perform the same tasks but with differing numbers of instructions (crochB7 utilizing a greater number), it would of course be expected that at some point crochB would outperform its successor. However, within the domain of observation of this thesis, this was not experienced in a sustained way. Observing the long string results, it can be seen that crochB7 is still outperforming crochB. Different possible explanations exist for this behaviour. Included are considerations regarding crochB's heavy reliance on data access (more so than crochB7's). Because of this fact, it could be that crochB evokes a greater number of cache misses, and the time cost of this might outweigh that incurred by the extra work performed by crochB7's additional calculations for data access (compared with crochB). It is possible as well that implementation issues might exist that could be responsible for some portion of the added running time of crochB. Further investigation into this matter would be interesting. Specifically, consideration of platforms whose cache sizes differ and cache profiling might provide further insight into this unexpected behaviour.

Bibliography

[1] M. Crochemore, *An optimal algorithm for computing the repetitions in a word*, Information Processing Letters, 12 (1981), pp. 297–315.

[2] F. Franek and M. Jiang, *Crochemore's repetition algorithm revisited: Computing runs.*, AdvOL-Report 2009/01, McMaster University, 2009.

[3] ——, *Crochemore's repetitions algorithm revisited: Computing runs.*, Int. J. Found. Comput. Sci., 23 (2012), pp. 389–401.

[4] F. Franek, M. Jiang, and C. Weng, *An improved version of the runs algorithm based on Crochemore's partitioning algorithm*, AdvOL-Report 2011/03, McMaster University, 2011.

[5] F. Franek, M. Jiang, and C. Weng, *An improved version of the runs algorithm based on Crochemore's partitioning algorithm*, in Proceedings of Prague Stringology Conference 2011, Prague, Czech Republic, August 2011, pp. 98–105.

[6] F. Franek, W. Smyth, and X. Xiao, *A note on Crochemore's repetitions algorithms – a fast space-efficient approach*, Nordic Journal of Computing, 10 (2003), pp. 21–28.

[7] H. P. Gavin, *Gnuplot 4.2 - a brief manual and tutorial*, 2008. `http://www.duke.edu/~hpgavin/gnuplot.html` (Last Visited: 2012-10-31).

[8] *gnuplot homepage*, Mar 2012. `http://www.gnuplot.info/` (Last Visited: 2012-10-31).

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

[10] D. Kim, J. Sim, H. Park, and K. Park, *Linear-time construction of suffix arrays*, Proc. of the 13th annual conference on Combinatorial pattern matching, (2003), pp. 186–199.

[11] P. Ko and S. Aluru, *Space-efficient linear-time constructions of suffix arrays*, Combinatorial Pattern Matching, (2003), pp. 203–210.

[12] R. Kolpakov and G. Kucherov, *Finding maximal repetitions in a word in linear time*, in 40th Annual Symposium on Foundations of Computer Science, 1999, pp. 596–604.

[13] W. S. M. Crochemore, L. Ilie, *A simple algorithm for computing the lempel-ziv factorization*, Proc. Data Compression Conference (dcc 2008), (2008), pp. 482–488.

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

[15] U. Manber and G. Myers, *Suffix arrays: A new method for on-line string searches*, siam Journal on Computing, (1993).

[16] W. Smyth, *Computing Patterns in Strings*, Addison-Wesley, 2003.

[17] C.-C. J. Weng, *Implementing efficient algorithms for computing runs*, m.sc. thesis, McMaster University, 2011.