Bachelor Thesis

on

External Batched Range Minimum Queries and LCP Construction

by Daniel Feist

Date of submission: April 11th, 2013

Supervisors: Prof. Dr. rer. nat. Peter Sanders
Dipl. Inform. Timo Bingmann

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht zu haben und die Satzung der Universität Karlsruhe (TH) zur Sicherung guter wissenschaftlicher Praxis beachtet zu haben.

Karlsruhe, den 11. April 2013

Zusammenfassung

Diese Arbeit befasst sich mit I/O-optimaler Suffix Array- (SA) und Longest Common Prefix (LCP) Array-Konstruktion im externen Speicher. Dazu wird der I/O-optimale *DC3*-Algorithmus um die LCP-Konstruktion erweitert und anschließend entsprechend angepasst, um in das externe Speichermodell übertragen werden zu können. In diesem Zusammenhang stellen wir eine Methode vor, um die dafür benötigten Range Minimum Queries (RMQs) effizient im externen Speicher zu berechnen. Kern dieser Arbeit stellt die Beschreibung und Implementierung des hieraus resultierenden externen *DC3*-LCP-Algorithmus mithilfe der STXXL - der C++ Standard Template Library for Extra Large Data Sets - dar. Experimentelle Ergebnisse auf Basis realistischer Eingabeinstanzen runden diese Arbeit ab.

Abstract

This work deals with I/O-optimal suffix array (SA) and longest common prefix (LCP) array construction in external memory. For this purpose, the I/O-optimale *DC3* algorithm is enhanced by LCP construction and adapted accordingly to the external memory model. In this context we present a method to construct the required range minimum queries (RMQs) efficiently in external memory. The core of this work is a description and implementation of the resulting external *DC3*-LCP algorithm using the STXXL - the C++ Standard Template Library for Extra Large Data Sets. Experimental results based on realistic, real-world instances rounds off this work.

Acknowledgements

At this point, I would like to take this opportunity to thank Timo Bingmann for the patience with me while disussing algorithm development and implementation issues. Furthermore, I am indebted to Bettina Umminger for proofreading the written part.

Contents

1 Introduction

1.1 Motivation and Previous Related Work

The suffix array (SA) is defined as the lexicographically sorted array of the suffixes of a string T, as first described in [24,25]. Suffix arrays in general are one of the most used full text indexing structures and have multiple applications such as string matching [19,24,25], text compression [7] and genome analysis [20].

The first results on suffix array construction date back to the early 1990s. Let T be an arbitrary string of size n, a pattern M of size m, and occ the number of occurences. A binary search algorithm using the suffix array helps to locates all occurence positions of M in T in $\mathcal{O}(m \log(n) + occ)$ time in the internal memory model. This boundery can be lowered to $\mathcal{O}(m + \log(n) + occ)$ [24] by means of the array of the lowest common prefixes (LCPs) of adjacent suffixes in the suffix array. Later results such as [23] or [26] for example allow $\mathcal{O}(n)$ time LCP array construction using the already preprocessed SA. None of them were designed for external memory (EM) usage.

Until 2003, a direct, I/O-optimal, linear time suffix array construction algorithm was unknown and mentioned as an important open problem by A. Crauser and P. Ferragina [8]. Direct suffix array construction algorithms needed $\mathcal{O}(n \log(n))$ time even on constant alphabets (i. e. the size of the alphabet is bounded). The doubling [2] and doubling combined with discarding [8] algorithms are two representations of this idea. In 2003, three independent approaches of a direct, linear time suffix array construction algorithm were published. The so-called *DC3* (abbreviation for difference cover modulo 3) algorithm by J. Kärkkäinen and P. Sanders [6,22] was one of them and will be the focus of this work. The alphabet model used for *DC3* is an integer alphabet (i. e. the elements of the alphabet are integers in a range of $n^{\mathcal{O}(1)}$ size). J. Mehnert's work [27] compares *DC3* with optimized versions of doubling and doubling combined with discarding. According to their tests, *DC3* proved to be "the fastest one with the smallest I/O volume for most of the input instances" also in practice. Other external memory *DC3* implementations [13] confirmed a distinct superiority over other external memory suffix array constructors at that time.

Apart from the introductional description of *DC3* in [22] the authors also describe in short words how their algorithm could be extended to compute both suffix and the LCP array in internal memory. They show that the new algorithm can be implemented to run in linear time like *DC3*. This so-called *DC3-LCP* (LCP-enhanced external memory *DC3*) algorithm is at the same time the only external memory LCP array construction algorithm known till January 2013.

While this work was being written, T. Bingmann, J. Fischer and D. Osipov [5] presented an implementation of an algorithm for external memory called *eSAIS* which is based on J. Fischer's "Inducing the LCP-array" [15]. They "implement[ed] the first external memory LCP array construction algorithm that is faster than a *DC3*-based approach". In addition, they state that "*eSAIS* is about two times faster than the external memory implementation of *DC3*, the I/O volume is reduced by a similar factor" and *eSAIS* is "3-4 times faster" than the LCP construction made by the extended *DC3*. Additionally, they observe that the "increase in both time and I/O volume of *eSAIS* with LCP array construction compared to pure suffix array construction is only around two".

1.2 Our Contribution

Further details on *DC3-LCP*, with which their *eSAIS* algorithm [5] is compared, are missing. As a result, this work can be considered as an add-on concerning this matter. Chapter 2 will contain some basic definitions. After explaining fundamental techniques and design of external memory algorithms in chapter 3, we will introduce *RMQ* computation in general and in the parallel disk model (PDM), the external memory model. The latter are needed in the

LCP-enhanced *DC3-LCP* algorithm. Then the external memory *DC3-LCP* algorithm will be explained by using a pseudocode representation as well as a flow graph. In chapter 5, a detailed example will be given. In order to narrow the gap between theory and practice, we have tested our implementation on a huge range of possible inputs. The results thereof will be shown in chapter 6. At the same time we will pay close attention to realistic, real-world inputs. Chapter 7 is the conclusive part of this work, where we interpret our measures and compare our results with those presented in [5] (for *eSAIS*). A parallel and distributed of algorithm [5] (the *eSAIS*) is probably not applicable. The *DC3* algorithm, however, has been parallelized in EM. Consequently, *DC3-LCP* is the today's most promising approach for parallel and distributed construction of large text indexes.

2 Basics

This chapter introduces several fundamental notations and definitions of data structures which occur several times in this work. In each section we first give a formal definition and if reasonable a short example. The following chapter 3 extends this section by introducing basics concerning external memory. Note that construction details on data structures were omitted because they are considered in subsequent sections in detail.

2.1 Alphabets and Strings

An alphabet Σ in general is defined as a set of totally ordered elements (characters). Following the alphabet definitions in [22], we distinguish three different types, namely constant alphabets with $|\Sigma| = \mathcal{O}(1)$, integer alphabets (where the characters are integers in the range n) with $|\Sigma| = n^{\mathcal{O}(1)}$ and general alphabets which only assume that characters can be compared in constant time. Consequently, the sorting complexity $\mathcal{O}(Sort(n))$ is determined by the underlying alphabet (and the sorting algorithm).

Let Σ be a totally ordered alphabet with `$` as its smallest character. A string $T := [0, n-1]$ (or a substring $T_i := [i, n-1]$) is defined as an element $T := S \circ \texttt{\$}$ with $(\Sigma \setminus \texttt{\$})^{\mathbb{N}}$, whereas $\circ$ is the concatenation operator. Given two strings A and B, we call A lexicographically smaller than B, i. e. $A < B \iff \exists i \in [0, \ldots, n-1] : (A[0, i-1] = B[0, i-1]) \wedge A[i] < B[i]$. A and B are lexicographically equal, i. e. $A = B \iff \forall i \in [0, \ldots, n-1] : A[i] = B[i]$. In this context we introduce the term *lexicographical names* as defined in [22]. The lexicographical names n_1 and n_2 for two strings T_1 and T_2 are numbers with the property that they hold $n_1 \leq n_2$ iff $T_1 \leq T_2$. Furthermore, the term *overlap* between two strings $A[0, n]$ and $B[0, n]$ is defined by the value of $lcp(A, B)$ (see chapter 2.3 below). Consider an arbitrary string T of length k, then $T_{(k)} := [0, k-1]$ (index k in enclosed parentheses) denotes the whole string.

2.2 Suffix Array

The suffix array (SA) as described by their introducers U. Manber and G. Myers in [24, 25] is the lexicographically sorted array of string suffixes. For a given string T, the suffix array SA of T is the permutation of integers in the range of $[0, n)$ holding $i < j \iff T_{SA[i]} < T_{SA[j]}$ for $0 \leq i < j < n$. The inverse permutation of SA is defined as $ISA[i] := j :\Leftrightarrow SA[j] = i$ in general. The *DC3-LCP* algorithm explained later will use $ISA[i] := j + 1 :\Leftrightarrow SA[j] = i$ instead. As an example, assume $T :=$ `papaya$`. Then $SA_T := 5\,1\,3\,0\,2\,4$ satisfies the property $T_{SA[i]} < T_{SA[i+1]}\ \forall i \in [0, 4]$ since `a$` < `apaya$` < `aya$` < `papaya$` < `paya$` < `ya$`. While the suffix array position $SA[i]$ represents the position of the i-th smallest suffix $T_{SA[i]}$ of T, the inverse suffix array $ISA_T := 3\,1\,4\,2\,5\,0$ stores the position where a given suffix T_i appears in SA. Note that we always exclude the suffix `$` from SA.

2.3 Longest Common Prefix Array

The array of the longest common prefixes (= LCP array) was introduced together with the improvement of pattern matching with suffix arrays in [24]. It contains the lengths (called the *lcp*-values) of the longest common prefix of suffixes that are adjacent (which means lexicographically successive) in the suffix array SA - formally: $LCP[i+1] := lcp(T_{SA[i]}, T_{SA[i+1]})$ for $i \in [0, n)$, whereas $LCP[0] := \perp$ and $LCP[i] := \perp$ for $i < 0 \wedge i \geq n$ stays undefined by definition. As an example, assume $T :=$ `papaya$` with $SA_T := 5\,1\,3\,0\,2\,4$ as before. Thus, $LCP_T := \perp\,1\,1\,0\,2\,0$ since $LCP[0] := \perp$ by definition, $LCP[1] := lcp($`a$`, `apaya$`$) = 1$, $LCP[2] := lcp($`apaya$`, `aya$`$) = 1$, $LCP[3] := lcp($`aya$`, `papaya$`$) = 0$, $LCP[4] := lcp($`papaya$`, `paya$`$) = 2$, $LCP[5] := lcp($`paya$`, `ya$`$) = 0$.

2.4 Range Minimum Queries in General

In accordance with the definition given in [16], the range minimum query (RMQ) problem can be formulated as follows: Given a static array $A := [1, n]$ of n objects from a totally ordered set and two integers $i \in \mathbb{N}$ and $j \in \mathbb{N}$ with $1 \leq i \leq j \leq n$, $RMQ_A(i, j)$ returns the position of the minimum object of the sub-array $T[i, j]$; in short $RMQ_A(i, j) := \arg\min_{i \leq k \leq j}\{A[k]\}$. As an example, consider the array $A := [2, 4, 3, 9, 1, 5, 4]$. Here, the result of the range minimum query $RMQ_A[3, 6]$ would be position 6 with $A[6] = 1$. In most cases it is necessary to answer many arbitrary queries (i, j) on A. This can be very costly without any precomputation of A. Additionally, subsets of queries may be batched which means they are not answered immediately but as a whole. Investing time in preprocessing A to answer future queries will also be our approach [17]. Consequently, we need a fast precomputeable and space-preserving data structure that allows to compute subsequent RMQs on a static array $A := [1, n]$ of n totally ordered objects in $\mathcal{O}(1)$ time.

2.5 Pseudocode Conventions

Our pseudocode syntax is fairly similar to the Pascal Programming Language. We use *if then else-if, else* condition expressions, the "C" ternary operator *?:* as abbreviation for *if then, else*, *while do* and *for do* loops. Elements in round brackets such as $(e_1, e_2, \ldots, e_k)$ represent k-tuples. Sets of elements are accurately specified in curly brackets $\{\}$ and assigned to their codomain by $:=$ as the assignment operator. For example, let T be an arbitrary string, then the set $M := \{T[i] : i \in \mathbb{P}\}$ stores every element of T where the index position is a prime number. The '.' operator executes an operation on a set of elements. M.`Sort`(in lexicographical order) e. g. executes the *Sort* method (which itself sorts elements in lexicographical order) on the set M. The $\oplus$ operator is only used once and works as follows: Let $S_1, \ldots, S_n$ be sets of elements of equal cardinality m, i. e. $\forall i \in [1, \ldots, n] : |S_i| = m$, then $S_1 \oplus, \ldots, \oplus S_n := \{(S_1[i] + \ldots + S_n[i])\ \forall i \in [1, \ldots, m]\}$. To put it simple, the $\oplus$ operator unites multiple sets into a new set by adding them component-wise.

3 Operations and Techniques in External Memory

This chapter can be seen as a continuation of chapter 2. Before we go into detail, we first classify external memory with respect to memory hierarchies. The following paragraph is based on [9] and [28].

Computer architectures nowadays contain memory hierarchies of increasing size, decreasing speed and decreasing costs from top to bottom. Beginning at the top we have registers integrated in the CPU, then a number of caches, after that the main memory and finally disks at the bottom. The bottom is often referred to as external memory (secondary memory) as in contrast to the internal memory (primary memory). Regarding the situation that very big data sets occur in many applications and the internal memory can only keep a small fraction of them, processing needs external memory access from time to time. One such access can be 10^6 times slower than a main memory access. Therefore the external memory accesses become the main bottleneck in such cases. Minimizing the number of I/Os will lower the time complexity of the algorithm as well. A single I/O denotes the transfer of one block between the disk(s) and internal memory.

Hence chapter 3.3 outlines a technique which can lower the number of I/Os. Chapter 3.1 presents the most common external memory model which is used to design I/O efficient algorithms. Chapter 3.2 introduces two elementary operations used in external memory.

3.1 A Theoretical External Memory Model

In case of problem instances which do not fit into internal memory completely, the instances are typically stored in external memory (EM) on one or more hard disks with the consequence that the latency for accessing data on hard disks generally amounts to several milliseconds which is slow compared to CPU registers or cache memory access in less than one nanosecond [29]. Thus, the (internal memory) RAM model cannot be suitable in such cases so that the parallel disk model (PDM) presented by J. S. Vitter and E. A. M. Shriver [30] can be used instead. The PDM, explained in [29], has the following parameters:

(i) $N :=$ problem size,

(ii) $M :=$ size of fast internal memory,

(iii) $B :=$ block transfer size,

(iv) $D :=$ number of independent hard disks,

(v) $P :=$ number of CPUs, (whereby it applies that $M < N$ and $1 \leq D \cdot B \leq M/2$).

We have D disks and each of them can simultaneously transfer a block of size B of contiguous data items in a single I/O (i. e. D disks can move $D \cdot B$ elements in one I/O access). We assume that the data for the problem is initially "striped" across the D disks in units of blocks. The performance measures in PDM are:

(i) the number of I/O operations,

(ii) the disk space used,

(iii) the internal computation time (in the RAM model).

Here, in our external memory model PDM we merely take the number of I/O operations into account. A serious problem of the PDM is explained in [27]. The authors mention that the PDM does not distinguish between random disk accesses and bulk accesses. Bulk access means in this case reading consecutive blocks, which is much faster than seeking the position of every block. Therefore, the PDM model is only practical iff there is no or a very little number of random accesses. As we will see later during discussion of how to implement our algorithm in the EM model, we will use fundamental (see chapter 3.2), non-random-access operations exclusively. Hence the PDM model is appropriate here.

3.2 Fundamental I/O Operations

Apart from explaining the PDM, J. S. Vitter's work [29] also gives an overview of I/O bounds for fundamental operations. Following that work, we will take a closer look at two of them, namely *Sort* and *Scan* (=*Stream*), of which we make heavy use in our algorithms. The I/O bounds of these fundamental operations are $Sort(N) := \mathcal{O}\left(\frac{N}{D \cdot B} \log_{M/B} \frac{N}{M}\right)$ I/Os and accordingly $Scan(N) := \mathcal{O}\left(\frac{N}{D \cdot B}\right)$ I/Os. [28] names two so-called "golden rules" in the context of these bounds and describes the reason for their need. Random access on external memory is often very expensive because it comes with one I/O per operation whereas we want $1/B$ I/Os per operation for an efficient algorithm. The first rule would be to scan the external memory instead by always loading the next due block of size B in one step and processing it in internal memory. This costs $Scan(N)$ I/Os. If the data is not scannable, the second rule implies the use of sorting (which costs $Sort(N)$ I/Os) to reorder and then use scanning.

3.3 Pipelining

As mentioned in the introduction chapter 1, we want to lower the number of I/Os as far as possible, because it dominates the time complexity of our external memory algorithm. According to the description of pipelining in [12], the idea is to equip the external memory algorithms with a new interface that allows them to feed the output as a data stream directly to the algorithm that consumes the output, rather than writing it to external memory. Consequently, the input of an external memory algorithm does not have to reside on hard disks, it could rather be a data stream produced by another external memory algorithm.

We describe the characteristic elements of external memory algorithms according to [12] and [27]. These elements can be modelled as a data flow through a directed acyclic graph $G := (V, E)$ with $V := (F \cup S \cup R)$. As a result, edges E in G denote the directions of data flow between nodes. We distinguish between file nodes F, streaming nodes S and sorting nodes R as the most generic nodes.

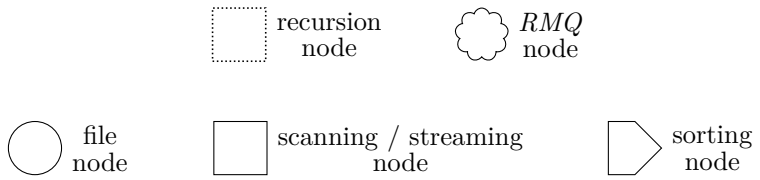


Figure 1: Pipelining objects displayed as symbols

Figure 1 shows a summary of the different nodes which are used in this work. The representation and definition is taken from [27]. The sorter run formation node and sorter merge node are united in the sorting node, the streaming or scanning node is used synonymously. A recursion node represents the result computed by the streaming in the recursion call. A new addition is the *RMQ* node which will be discussed in detail in chapter 4.

A file node F represents physical data sources and data sinks which are stored on hard disks. The streaming node S expects none, one or more input streams and outputs none, one ore more new ones. Note that streaming nodes usually do not perform any I/O, unless access to external memory data structures is needed. Sorting nodes R read a stream and output it in a sorted order. Analogous to the pipelining structure, representation and definition introduced in [27], the sorting nodes in this work represent the pipelined external merge sort [1] which itself consists of a sorter run formation node and a sorter merge node. Suppose we want to sort a sequence of N elements. At the beginning, the sorter run formation node first sorts $\left\lceil \frac{N}{M} \right\rceil$ partial sequences of size M internally and writes them to the disk(s). In the second step, the sorter merge node reads the sorted subsequences from the disk(s) and merges them to the final sorted sequence.

The graph representation of the sorter run formation node would be a streaming node $s_1 \in S$ as input node and a file node $f_1 \in F$ as output node (connected with an edge $e_1 \in E$) recursively if more than $\frac{M}{B}$ streams. The graph representation of the sorter merge node would be just the other way round, having a file node $f_2 \in F$ as input node and a streaming node $s_2 \in S$ as output node (connected with an edge $e_2 \in E$). If we apply this to the example above, the runs will be sorted in s_1 internally and the sorted sequence will be stored in f_1. The merger reads the sorted subsequences from the new file node f_2 and merges them to the final sorted sequence inside of s_2.

4 Range Minimum Queries in External Memory

4.1 Overview

There have been many new results in research on *RMQ* construction in the RAM model in the past few years as the introduction of [17] indicates. Those new data structures have a construction time complexity of $\mathcal{O}(n)$ and allow answering *RMQ* in $\mathcal{O}(1)$ time each. The challenge in internal memory is to minimize the additional space consumption. Provided that the input array is available at construction time, J. Fischer and V. Heun [14,17] present a data structure of size $2n + o(n)$ bits allowing answering *RMQ*s in $\mathcal{O}(1)$ time which is asymptotically optimal under the mentioned condition.

In contrast to *RMQ* in internal memory, the situation in our PDM is a different one. Note that we only take the number of I/O operations into account (the additional space consumption or the internal computation time remain unconsidered). As the time complexity of the latter introduced *DC3-LCP* algorithm is dominated by external sorting, we may assume that *RMQ* construction in $\mathcal{O}(n \log(n))$ time complexity (which allows $\mathcal{O}(1)$-time *RMQ*-retrieval) does not change the asymptotical complexity of our algorithm. Since we make use of internal *RMQ*s as described in chapter 4.3, we only need external sorting and scanning.

4.2 The Sparse Table Approach

In order to answer *RMQ* queries, the so-called sparse table approach was introduced by M. A. Bender and M. Farach-Colton [4]. It provides $\mathcal{O}(n \log(n))$ construction time and space consumption to improve the naive idea of storing a quadratic table which stores every possible query combination (resulting in both construction time and space consumption of $\mathcal{O}(n^2)$).

We refer to the remarks of [21] and [16] where the algorithm is presented as follows: Given a static array $A := [1, \ldots, n]$, we answer $RMQ_A(i, j)$ by a simple look-up data structure. The idea is to store the answers of all possible index-pairs i and j whose differences $j - i + 1$ are a power of two. This results in a 2-dimensional table $Q[i, k]$ for $i \in [1, \ldots, n]$ and $k \in [0, \ldots, \lfloor \log(n) \rfloor]$ with $Q[i, k] := \arg\min_\ell \{A[\ell] : \ell \in [i, i + 2^k - 1]\}$. The sparse table has a width of $\log(n)$ and a depth of n as resulting from above. Accordingly, it needs $\mathcal{O}(n \log(n))$ space to store the elements. Algorithm 1 outlines the procedure of the construction of the sparse table by dynamic programming in $\mathcal{O}(n \log(n))$ time.

The first column of Q is notably easy to compute since the position of the minimum of every element is the position of the element itself (line 2). The outer for-loop (line 3) iterates over the columns k, the inner for-loop (line 5) uses k as an exponent to generate the power of two in ascending order. In order to construct a column in Q, get the position of minimums with a power of two difference from the previous column and compare them by looking up their values in A (line 6). To prevent accessing values in Q which do not exist, the inner for-loop breaks in time (line 8).

But how does $\mathcal{O}(1)$-time *RMQ* retrieval work on Q? This is shown in algorithm 2. Compute $k := \lfloor \log(j - i + 1) \rfloor$ to define two overlapping blocks of length 2^k (line 2). 2^k is the maximum block-length which fits in the subarray between i and j entirely without moving out of range, i. e. $2^k \leq j - i + 1 \leq 2^{k+1}$. Then determine the indices of the minima of $A[i, i + 2^k - 1]$ (represents the maximum block-length starting from at i-th position) and of $A[j - 2^k + 1, j]$ (represents the maximum block-length starting at the j-th position) respectively, using $r := Q[i, k]$ and $s := Q[j - 2^k + 1, k]$ (line 3 and 4) respectively. Now, r as well as s represent the indices of the minima in A in the two ranges of length 2^k. If $A[r] \leq A[s]$ then position r in A will contain the minimum we are looking for, else position s in A will show the minimum.

Algorithm 1: Construction of the sparse table data structure

```
1 constructSparseTable(A) begin
2     limit := 0, Q[i, 0] := i ∀i ∈ [1, n]
3     for (k := 0; k < ⌊log(n)⌋); k++) do
4         limit := limit + 2^k
5         for (i := 1; (i + 2^k) ≤ n; i++) do
6             if (A[Q[i, k]] ≤ A[Q[i + 2^k, k]]) then Q[i, k + 1] := Q[i, k]
7             else Q[i, k + 1] := Q[i + 2^k; k]
8         if (equals(i, (n − limit))) then break
9     return Q
```

Algorithm 2: Answering *RMQ*s by sparse table usage

```
1 answerRMQs(Q, i, j) begin
2     k := ⌊log(j − i + 1)⌋
3     r := Q[i, k]
4     s := Q[j − 2^k + 1, k]
5     if (A[r] ≤ A[s]) then return r
6     else return s
```

In consistency with the definition of *RMQ*s on A as given in chapter 2.4, we return the position of the minimum. A slight improvement can be achieved by storing the current minima itself in Q instead of storing their positions. As a result, we have to scan A only once.

Figure 2 depicts an illustrative example of how the sparse table of the given array A is processed and how it can be used to answer *RMQ* queries. At the beginning, the first column considers ranges of 0 in A. Consequently, the position of the minimum between a single value is the position itself and we have increasing values in the first column. The second column can be computed by using the first column and consider ranges of $2^0 = 1$. The values encircled by rounded rectangles indicate a comparison between all $A[value]$'s. The comparison of the elements $A[1] = 4$ and $A[2] = 3$ (first red rounded rectangle in first column) e. g. gives us the information that if we consider the range beginning at $i = 1$ of length $2^0 = 1$, then the minimum is at position 2. As one can see, this information is stored in the row $k = 1$. The third column again can be computed by using the second one, considering ranges of $2^1 = 2$ and so on.

Determining $RMQ_A(1, 7)$ by using Q works as follows: First, compute $k := \lfloor \log(7 - 1 + 1) \rfloor = 2$ to define two overlapping blocks of length $2^k = 4$ (see gray shaded rounded rectangles below). The minimum in the two block sections can be computed by two simple look-ups in Q, namely: $r := Q[i, k] = Q[1, 2] = 3$ and $s := Q[j - 2^k + 1, k] = Q[4, 2] = 5$, respectively. A single comparison of $A[r]$ and $A[s]$ identifies $A[s] = 0$ as the smallest element in the given range. Therefore, $RMQ_A(1, 7) = 5$ is returned. This implies the minimum in the range between index position 1 and 7 is located at position 5.

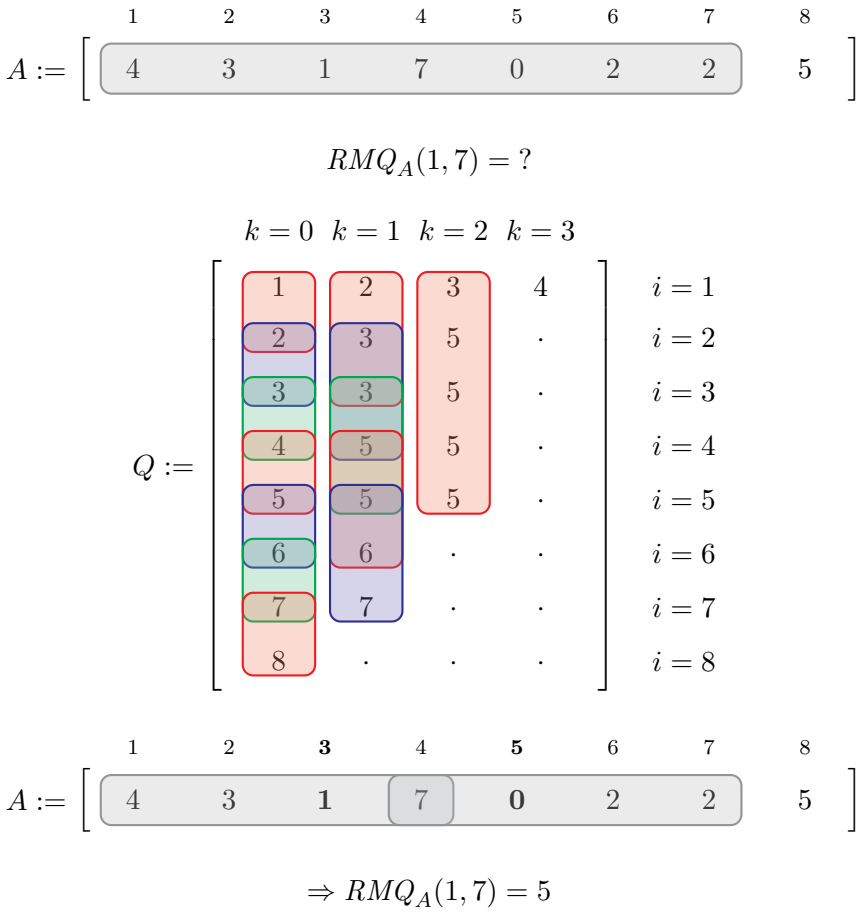


Figure 2: Example of sparse table construction and usage

4.3 The Sparse Table Approach Applied in External Memory

This section describes how we compute RMQs in external memory by using the sparse table approach. The procedure is formulated as pseudocode in algorithm 3 and as a graphical representation in figure 3. The static array A and a sequence of (RMQ) triples represented as $RMQ(id, i, j)$ are both stored in external memory. The id parameter is needed to identify a specific RMQ at a later point in time if query results don't arrive in order. We now want to answer all $RMQ_A(id, i, j)$ (on A).

Consider we have an internal memory of size M whereas A does not fit into M completely. We split A into longest possible parts v (called pages) of equal size under the constraint of having enough space for the sparse table which consumes $|v| \cdot |v| \log(|v|)$ space. Moreover we need the additional minBorderArray of size $\frac{n}{|v|}$ and $3 \cdot const$ space for the sorters we use later. The inequation in line 2 describes this constraint. First, stream through the sequence containing the RMQs from the beginning on. While streaming through, distinguish two possible cases (line 3-4):

the trivial RMQs (referred to as RMQ^{T_1} in figure 3) are inside a page and notably easy to handle: i and j are computed relative to the page borders as i' and j' (for smaller storage datatype) of v_k. The resulting $RMQ^{left}(id, i', j', k)$ is pushed into the left sorter L. The non-trivial RMQs (referred to as RMQ^{T_2} in figure 3) are processed differently: split them into two sub-RMQs with respect to the page borders. The left part RMQ^{left} is equal to the trivial case and is pushed into the left sorter L, $RMQ^{right}(id, i', j', k, k^*)$ has the additional information $k^* := k \cdot |v|$ which stores a distance factor between the right and the left part in pages and is pushed into the right sorter R. The upper part of figure 3 illustrates this procedure. Now L and R are sorted externally in ascending order by tuple component k, where k represents the page concerned

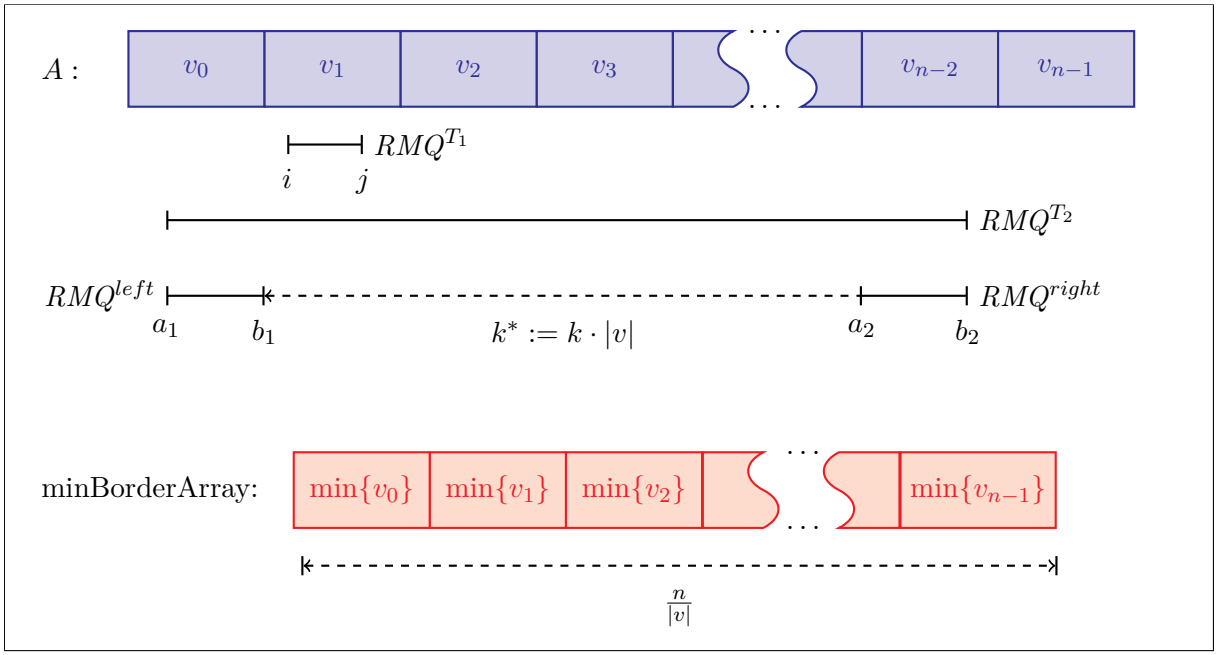


Figure 3: Processing of RMQs in external memory

(line 7 and 8). Then loop through A, load every page v_i and save it to internal memory. While doing this, construct the so-called minBorderArray (see lower part of figure 3), which holds the minimum of every page v_i (line 10). Then build up the sparse table of v_i (line 11). Answering all RMQs of current page separated in L and R is possible by a simple sparse table look up and minBorderArray scanning (if we compute an RMQ^{right} out of R) (line 13). All answered sub-RMQs are pushed into a final sorter C which holds reduced tuples (id, min) (line 14). The finishing step sorts the tuples in C by their id to restore their original order (line 15).

There is a simple idea with which the internal work of repeated scans on minBorderArray can be reduced. The technique described in [18] uses a stack-like succinct data structure which holds youngest positions of all valid minima. Nevertheless, the naive approach is sufficient enough because the minBorderArray is small and we ignore internal work.

Algorithm 3: External RMQs

```
1  externalRMQ(A, 𝔏 : List of (id, i, j)) begin
2      calculate v so that M ≥ |v_i| + |v_i| · |v_i| log(|v_i|) + n/|v_i| + 3 · const ∀i ∈ [0, ..., n − 1].
3      split up items of 𝔏 into RMQ^left(id, i', j', k) and RMQ^right(id, i', j', k, k*)
4      and push them into 𝔏 (left sorter) and R (right sorter)
5      // split and push while scanning - i. e. 𝔏.Scan()
6      // k defines the page in A
7      L.Sort(by component k)
8      R.Sort(by component k)
9      while ∃ v_i ∈ A do
10         minBorderArray[i] := min{A[v_i]}
11         Q_i := constructSparseTable(A[v_i])
12         // answer RMQs, push results (id, min) into C (collective sorter)
13         answerRMQs(Q_i, L, R)
14     C.Sort(by id)
15     return (id, min) tuples of C
```

4.4 Range Minimum Queries in the Pipelining Model

Finally, we can complete the Pipelining Model introduced in chapter 3.3. The numbers inside the pipeline elements refer to the line numbers in algorithm 3. The *RMQ*s are splitted by scanning, distributed to the sorters L and R which are then sorted. Scanning A as well as L and R answers the sub-*RMQ*s which are pushed into sorter C. After sorting C, the results are returned by scanning.

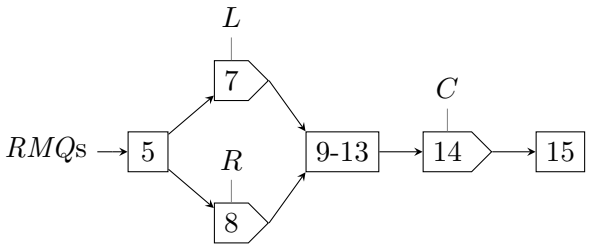


Figure 4: *RMQ*s in the pipelining model

5 External Memory Suffix Array and LCP Array Construction

5.1 Introduction

As already stated in the introductory chapter 1, the also called *DC3* or *skew* algorithm published by J. Kärkkäinen and P. Sanders in 2003 [6, 22], was one of three independent approaches of a direct, linear time suffix array construction algorithm over integer alphabets in internal memory. In general, *DC3* constructs the suffix array in sorting complexity $\mathcal{O}(Sort(n))$. Along with the description of *DC3* in [22] the authors also describe how their algorithm can be extended to compute both the suffix and the LCP array simultaneously in $\mathcal{O}(Sort(n))$ as well. In 2005, R. Dementiev, J. Kärkkäinen, J. Mehnert and P. Sanders [10] presented the first external memory suffix array construction algorithm, a pipelined version of the *DC3* algorithm, that is at the same time asymptotically optimal and the best practical algorithm until then. In the following section we will see in detail how *DC3-LCP*, a direct, pipelined, I/O-optimal external suffix and LCP array construction algorithm based on *DC3* can be designed. In the following, the term *DC3-LCP* refers to the algorithm in external memory. In chapter 5.2 we explain the pseudocode representation and the pipelined model of *DC3-LCP* as presented in chapter 5.4. An illustrative example is given in chapter 5.3.

5.2 The DC3-LCP Algorithm

The following explanations of the different steps are based on the descriptions from [10] and [22]. They explain the single steps of the pseudocode algorithm 4 and the corresponding flow graph of chapter 5.4. Let T be a string over the alphabet $\Sigma := [\$, 1, \ldots, n]$. Obviously, any other imaginable alphabet Σ is possible here since it is can be transformed into the previously stated structure. We want to construct both SA_T and LCP_T simultaneously with the properties mentioned in chapter 5.1 above.

1. **Step**: At first, we pick all suffix triples $T[i, i+2]$ of the input string T at index positions $i \bmod 3 \not\equiv 0$ and store them together with the position i (line 3). Then the triples are sorted in ascending order (line 4). Note again that the sorting complexity depends on the underlying alphabet (and on the sorting algorithm). Thus we have $\mathcal{O}(Sort(n))$ I/Os in external memory.

   After that, we assign lexicographical names $n_i \in [0, \left\lfloor \frac{2 \cdot n}{3} \right\rfloor)$ to the sorted triples so that the k-th different triple $T[i, i+2]$ in the sorted sequence has the lexicographical name $n_i = k$ (line 7). Additionally, the overlap between two lexicographical adjacent triples is saved in LCP^N (line 7).

   In case of unique lexicographical names n_i, we skip the if-section and are done with the first step. Otherwise, we compute a string R which corresponds exactly to the lexicographical names of the $i \bmod 3 \equiv 1$ triples concatenated with the lexicographical names of the $i \bmod 3 \equiv 2$ triples (line 10). Then we make a recursive call using R as input parameter (line 11) and repeat until all lexicographical names are unique.
2. **Step**: The precondition of this step is that the lexicographical names are unique. The consequence is that we now have the SA^{12} array as the lexicographical names (second component) in P with the desired order of suffixes T_i with $i \bmod 3 \not\equiv 0$. To obtain the position where a given suffix T_i appears in SA^{12}, we compute ISA^{12} by sorting P (line 16). Every suffix T_i with $i \bmod 3 \not\equiv 0$ represents exactly the suffix $S_{\varphi(i)}$, whereas $\varphi(i)$ is defined as expression 1.

   Expression 1.

$$\varphi(i) = \begin{cases} 3 \cdot i + 1 & \text{if } 0 \leq i < \left\lfloor \frac{|P|+1}{2} \right\rfloor \\ 3 \cdot (i - \left\lfloor \frac{|P|+1}{2} \right\rfloor) + 2 & \text{if } \left\lfloor \frac{|P|+1}{2} \right\rfloor \leq i < |P| \end{cases}$$

As a consequence, we apply $\varphi(i)$ on the index component (line 14-15). Note that the lexicographical names n_i are unique and called ranks r_i from that point in time to indicate that they never change again. We will see that the ranks are used to annotate each suffix position i with sufficient information to determine its global rank. Consequently, storing at most two ranks and one or two characters will be enough to completely determine the rank of every suffix (line 18-20). The reason for this will become clearer in Step 3. The suffix positions $i \bmod 3 \equiv 0$ can be processed by sorting on $T[i]$ and r_{i+1} (line 21). This works because the order of the suffixes T_{i+1} are already implicit in SA^{12}. Line 22 and 23 reconstruct the order of the $i \bmod 3 \equiv 1$ triples and of the $i \bmod 3 \equiv 2$ triples respectively.

3. **Step**: Finally, we have to merge the sorted sets. Line 24 implements simple comparison based merging with LCP computation added. The comparison function distinguishes three different cases:

   1. Merge an $i \bmod 3 \equiv 0$ suffix T_i with a $j \bmod 3 \equiv 1$ suffix T_j by comparing their first characters $T[i]$ and $T[j]$ and the rank of suffixes T_{i+1} ($\in i \bmod 3 \equiv 1$) and T_{j+1} ($\in i \bmod 3 \equiv 2$) which we already know from Step 1.
   2. Merging an $i \bmod 3 \equiv 0$ suffix T_i with a $j \bmod 3 \equiv 2$ suffix T_j works differently: Since we do not know r_{j+1} we can compare the first two characters $T[i], T[i+1]$ and $T[j], T[j+1]$ respectively, and the ranks of the suffixes T_{i+2} ($\in i \bmod 3 \equiv 2$) and T_{j+2} ($\in i \bmod 3 \equiv 1$).
   3. Merging an $i \bmod 3 \equiv 1$ suffix T_i with a $j \bmod 3 \equiv 2$ suffix T_j is our easiest case because the relative order of those suffixes can be determined from their position in SA^{12} which we already know from Step 1. Therefore, it is sufficient to simply compare rank r_i with rank r_j.

   At this point in time we can already obtain the suffix array SA_T as it is represented by the order of the index component i of the elements in S. To construct the LCP array in addition, we compute (and later unite) three arrays ℓ_1, ℓ_2 and ℓ_3 in this precise order. This order is important since ℓ_2 depends on ℓ_1 and ℓ_3 depends on both ℓ_1 and ℓ_2 as we will see in the next part. ℓ_1 saves the overlap in suffix characters $T[i]$ and $T[j]$ ($T[i+1]$ and $T[j+1]$) we can obtain by the tuple representation, ℓ_2 stores the *lcp* value delivered by the last recursive LCP^{12} array and ℓ_3 stores possible further partial overlapping as memorized in LCP^N.

   Construction of ℓ_1. We can easily extract the information about the partial overlap between triples from the S_0, S_1, S_2 representation. This information is stored in the ℓ_1 array (line 28) which holds values out of $\{0, 1, 2\}$, depending on the combination of the suffixes T_i and T_j we are merging. In case of suffix pairs T_i with $i \bmod 3 \equiv 0$ and T_j with $j \bmod 3 \not\equiv 0$ we can compare the first (first two) character(s) of each suffix and store the corresponding value in ℓ_1; otherwise the position in ℓ_1 is set to zero since this information can be found in the LCP^{12} array (see $\texttt{construct}_{\ell_1}()$ at line 1′).

   Construction of ℓ_2. Now we are able to compute ℓ_2 (line 29) with the $\texttt{construct}_{\ell_2}()$ method (line 13′): Again, in case of suffix pairs T_i with $i \bmod 3 \equiv 0$ and T_j with $j \bmod 3 \not\equiv 0$ we check for the maximum possible overlap by means of ℓ_1. If the overlap is not the maximum, then the starting position(s) of the respective suffixes represent triples with different beginnings and possible subsequent overlappings will be of no further relevance. As a consequence, the respective position in ℓ_2 becomes 0. However, if the overlap is the maximum, we will have to look up the minimal *lcp* value between the positions r_{i+1} and $r_{j+1} - 1$ (r_{i+2} and $r_{j+2} - 1$) in LCP^{12} which is the LCP array provided by the recursion. Due to the fact that the suffix pairs T_i with $i \bmod 3 \equiv 0$ and T_j with $j \bmod 3 \not\equiv 0$ (more precisely the resulting comparative ranks) do neccessarily not represent neighbouring suffixes in the SA^{12} array (and consequently not in LCP^{12}), we need the RMQ procedure (see chapter 4.3) for the first time.

In case of merging suffix pairs T_i with $i \bmod 3 \not\equiv 0$ and T_j with $j \bmod 3 \not\equiv 0$ (i. e. the suffixes out of S_1 and S_2), the situation is different: We can assign the value for ℓ_2 by reading out the position r_j in LCP^{12} due to the fact that these suffixes are neighboured in the calculated SA^{12} array. As two identical triples are represented by the same lexicographical name, we have to multiply the corresponding values in ℓ_2 by 3. However, this will be done later since the values are needed to calculate ℓ_3. Finally, the generated RMQ queries are answered and the corresponding results are written to the ℓ_2 array.

Construction of ℓ_3. Consider more generally, two suffixes T_i and T_j having identical elements e. g. at the first k positions. It is essential to understand that every element represents a triple component (from the first recursion level) which itself possibly represents single triple components. Although those components are identical for the first e. g. k positions, they differ at position $k+1$ in any case. Thus, a situation may occur where triples with different lexicographical names partially overlap. The size of a possible overlap can be obtained by the already calculated LCP^N array. The computation of ℓ_3 is a bit more complicated especially as a pseudocode representation and under our precondition that random access is not permitted (see line 30-42).

For easier exposition, we will proceed on the basis that random access is possible and will show later how to avoid that. As we did before, we now consider the case of suffix pairs with T_i with $i \bmod 3 \equiv 0$ and T_j with $j \bmod 3 \equiv 1$ (the other cases are very similar and differ only in the indices). Δ_2 denotes the respective value of ℓ_2. As a consequence we can obtain the value for ℓ_3 by expression 2:

Expression 2.

$$\begin{cases} RMQ_{LCP^N}[ISA^{12}[SA^{12}[r_{i+1}-1]+\Delta_2], ISA^{12}[SA^{12}[r_{j+1}-1]+\Delta_2]-1] & \text{if } \Delta_2 \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

This is possible because the values in ℓ_3 can only be different from zero, iff the values in ℓ_2 are different from zero as well. That is the case if the values in ℓ_1 indicate a maximum overlap. Otherwise can set the respective value in ℓ_3 to zero from the outset. The reason why expression 2 provides the desired value will be explained in the following:

One can take advantage of the fact that the ranks r_i in general (which were used to calculate SA_T in the previous merge step) are referenced to the LCP^N array (and the SA^{12} / ISA^{12} array) of the last recursive call. The rank r_{i+1} describes the position of the r_{i+1}-smallest suffix in SA^{12}. To skip the number of elements Δ_2, of which we already know that they overlap, we add Δ_2. Let $h := SA^{12}[r_{i+1}-1]+\Delta_2$ (substract one (-1) due to the ISA values are $\in [1,\ldots,n]$ instead of $\in [0,\ldots,n-1]$). Then $ISA^{12}[h]$ provides the position where the suffix T_h can be found in SA^{12}. We can now use this information to look up the overlap of a triple represented by a single element in LCP^{12}. As the suffixes T_h are not necessarily adjacent in SA^{12}, we need RMQs on LCP^N again. Unfortunately, accessing $SA^{12}[r_{i+1}-1]$ would cost one I/O each time since the ranks usually do not arrive in ascending order. Furthermore, the same problem arises for $ISA^{12}[h]$. Nevertheless, in order to scan SA^{12} and ISA^{12} (and avoid any random access) we proceed as follows:

Depending on the currently merged elements out of the S_0, S_1, S_2 representation, the ranks actually needed are determined by the `prepare`$_{\ell_3}$() function (line 31 / 27′). In order to compute the inner SA^{12} part, split it up in triples storing the id, the particular rank r and the value of ℓ_2 $(= \Delta_2)$ for left and right side of the above-mentioned expression and store them in Z (line 32). Then we sort the elements in Z by their rank component (line 33). Now we can scan through SA^{12} and store the answers in new tuples $(id, SA^{12}[r]+\Delta_2)$ (line 34) in Z' where r describes an arbitrary rank and then we sort again (line 35). As a result, this allows us to scan through ISA^{12} and generate new tuples (line 37). To restore the original order, we sort Z'' by the id (line 38) and can now feed the RMQ node which

returns the values for the ℓ_3 array in the correct order (line 40). Finally, we can compose LCP_T by combining ℓ_1, ℓ_2 and ℓ_3 with the component-wise add operator $\oplus$ (line 41). Here, one should not forget to multiply each value in ℓ_2 by 3 as already mentioned above.

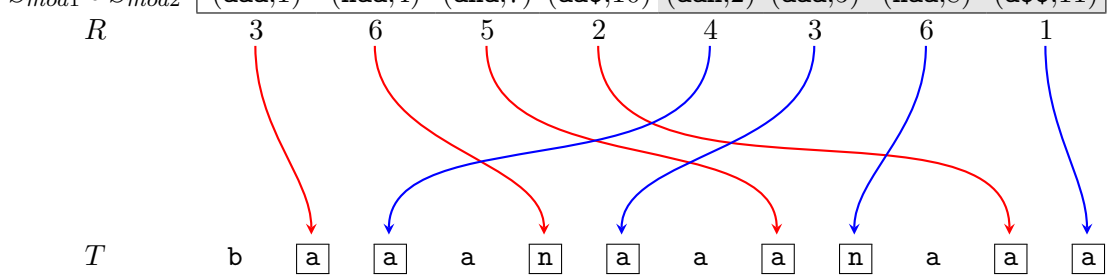
5.3 DC3-LCP in a Small Example

The following detailed example shall make the complex procedure of *DC3-LCP* more comprehensible. Technical details were omitted to give a better overview. Figure 5 presents the algorithm on the input string $T :=$ `baaanaaanaaa`.

	0	1	2	3	4	5	6	7	8	9	10	11
$T :=$	b	a	a	a	n	a	a	a	n	a	a	a

S	(aaa,1)	(aan,2)	(naa,4)	(aaa,5)	(ana,7)	(naa,8)	(aa$,10)	(a$$,11)
sort S	(a$$,11)	(aa$,10)	(aaa,1)	(aaa,5)	(aan,2)	(ana,7)	(naa,4)	(naa,8)
lex.	1	2	3	3	4	5	6	6
LCP^N	$\perp$	1	2	3	2	1	0	3

$S_{mod1} \circ S_{mod2}$	(aaa,1)	(naa,4)	(ana,7)	(aa$,10)	(aan,2)	(aaa,5)	(naa,8)	(a$$,11)
R	3	6	5	2	4	3	6	1
T	b	a	a	a	n	a	a	a	n	a	a	a



$\Downarrow$

Recursion on R

LCP^{12}	0	0	0	2	0	0	0	1
(SA^{12}, i)	(7,1)	(3,2)	(5,3)	(0,4)	(4,5)	(2,6)	(6,7)	(1,8)
$(\varphi(i), ISA^{12})$	(1,4)	(4,8)	(7,6)	(10,2)	(2,5)	(5,3)	(8,7)	(11,1)

i	0	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	b	a	a	a	n	a	a	a	n	a	a	a
r_i	-	4	5	-	8	3	-	6	7	-	2	1

Modulo $\{0\}$ set S_0

	comparison		
5-tuples	2-tuples	3-tuples	suffix T_i
(a,a,2,1,9)	(a,2)	(a,a,1)	$T_9 = [\texttt{aaa}]$
(a,a,6,7,6)	(a,6)	(a,a,7)	$T_6 = [\texttt{aanaaa}]$
(a,n,8,3,3)	(a,8)	(a,n,3)	$T_3 = [\texttt{anaaanaaa}]$
(b,a,4,5,0)	(b,4)	(b,a,5)	$T_0 = [\texttt{baaanaaanaaa}]$

Modulo $\{1, 2\}$ set $S_{12} := S_1 \cup S_2$

	comparison		
5-tuples	2-tuples	3-tuples	suffix T_i
(1,a,\$,⊥,11)		(a,\$,⊥)	$T_{11} = [\texttt{a}]$
(2,a,1,10)	(a,1)		$T_{10} = [\texttt{aa}]$
(3,a,a,6,5)		(a,a,6)	$T_5 = [\texttt{aaanaaa}]$
(4,a,5,1)	(a,5)		$T_1 = [\texttt{aaanaaanaaa}]$
(5,a,a,8,2)		(a,a,8)	$T_2 = [\texttt{aanaaanaaa}]$
(6,a,7,7)	(a,7)		$T_7 = [\texttt{anaaa}]$
(7,n,a,2,8)		(n,a,2)	$T_8 = [\texttt{naaa}]$
(8,n,3,4)	(n,3)		$T_4 = [\texttt{naaanaaa}]$

Merge S_0 with S_{12}

r_i	r_{i+1}	r_{i+2}	suffix T_i	ℓ_1	ℓ_2	ℓ_3	SA_T	LCP_T
1	⊥	⊥	$T_{11} = [\texttt{a}]$	⊥	⊥	⊥	11	⊥
2	1	-	$T_{10} = [\texttt{aa}]$	0	0	1	10	1
-	2	1	$T_9 = [\texttt{aaa}]$	1	0	1	9	2
3	-	6	$T_5 = [\texttt{aaanaaa}]$	2	0	1	5	3
4	5	-	$T_1 = [\texttt{aaanaaanaaa}]$	0	2	1	1	7
-	6	7	$T_6 = [\texttt{aanaaa}]$	1	0	1	6	2
5	-	8	$T_2 = [\texttt{aanaaanaaa}]$	2	1	1	2	6
6	7	-	$T_7 = [\texttt{anaaa}]$	0	0	1	7	1
-	8	3	$T_3 = [\texttt{anaaanaaa}]$	1	1	1	3	5
-	4	5	$T_0 = [\texttt{baaanaaanaaa}]$	0	0	0	0	0
7	-	2	$T_8 = [\texttt{naaa}]$	0	0	0	8	0
8	3	-	$T_4 = [\texttt{naaanaaa}]$	0	1	1	4	4

Figure 5: *DC3-LCP* on input string $T = \texttt{baaanaaanaaa}$

.

Description of the example. Pick modulo 1 and modulo 2 triples, sort them lexicographically, give lexicographical names and note their overlap in LCP^N. The lexicographical names are not unique and we make a recursive call with string R arising from concatenating the modulo 1 and modulo 2 triples and their corresponding lexicographical names by their original order in T. The recursion provides LCP^{12} ($= LCP_R$) and SA^{12} ($= SA_R$). Then sort (SA^{12}, i) by the first component to generate the ranks ISA^{12} (where a given suffix appears in SA_R). Each suffix in R represents a suffix $T_{\varphi(i)}$ (as already indicated by the red and blue arrows before). Sorting all modulo 0 triples is simple since we already know the rank r_{i+1} of every following (modulo 1) triple, see set S_0. The modulo $\{1, 2\}$ triples in set S_{12} are already sorted. The merging step distinguishes between several cases. We give an example of each case:

1. Merge($\mathbf{S_0}, \mathbf{S_1}$): pick e. g. (T_{10}, T_9)
   a) are the first characters of T_{10} and T_9 equal?
   Compare $(a, 1)$ with $(a, 2) \Rightarrow \ell_1$ is 1
   b) $\ell_1 = 1, RMQ_{LCP^{12}}[1, 2-1] = 0 \Rightarrow \ell_2$ is 0
   c) $\ell_1 = 1$, perform $RMQ_{LCP^N}[1, 2-1] = 1 \Rightarrow \ell_3$ is 1
2. Merge($\mathbf{S_0}, \mathbf{S_2}$): pick e. g. (T_6, T_2)
   a) are the first and second character of T_6 and T_2 equal?
   Compare $(a, a, 7)$ with $(a, a, 8) \Rightarrow \ell_1$ is 2
   b) $\ell_1 = 2, RMQ_{LCP^{12}}[7, 8-1] = 1 \Rightarrow \ell_2$ is 1
   c) $\ell_1 = 2$, perform $RMQ_{LCP^N}[1, 6-1] = 1 \Rightarrow \ell_3$ is 1
3. Merge($\mathbf{S_1}, \mathbf{S_2}$): pick e. g. (T_5, T_1)
   a) set ℓ_1 to zero
   b) $LCP^{12}[4-1] = 2 \Rightarrow \ell_2$ is 2
   c) perform $RMQ_{LCP^N}[1, 6-1] = 1 \Rightarrow \ell_3$ is 1
4. Merge($\mathbf{S_0}, \mathbf{S_0}$) $\Rightarrow$ equivalent to 1.
5. Merge($\mathbf{S_1}, \mathbf{S_1}$) or Merge($\mathbf{S_2}, \mathbf{S_2}$) $\Rightarrow$ equivalent to 3.

The result of the merging step are lexicographically sorted suffixes which represent SA_T. In order to obtain LCP_T we add ℓ_1, ℓ_2 and ℓ_3 row-wise after multiplying every element in ℓ_2 by 3.

5.4 DC3-LCP in Pseudocode and as a Flow Graph

The following pseudocode in algorithm 4 is an LCP-enhanced and slightly changed version of the external memory *DC3* pseudocode as it can be found in [10] and [27]. The resulting flow graph as a representation of the *DC3-LCP* algorithm is based on the *DC3* flow graph in [10] and is extended by the LCP part, see figure 6.

Algorithm 4: *DC3-LCP*

DC3-LCP(T) **begin**
  $n := |T|$
  $S := \{((T[i, i+2]), i) : i \in [0, n) \wedge i \bmod 3 \not\equiv 0\}$
  S.`Sort`(by first component $T[i, i+2]$)
  // `Scan` S and assign lexicographical names n_i to triples of S
  // P contains tuples (i, n_i) and $LCP^N \in \{0, 1, 2, 3\}$ their overlap
  $P, LCP^N := S$.`Lexnaming`()
  // check lexicographical names n_i in P for uniqueness
  if $\exists((i, n_i) \in P \wedge (j, n_j) \in P) : i \neq j \wedge n_i = n_j$ **then**
    P.`Sort`$((i, n_i)$ by $(i \bmod 3, i \operatorname{div} 3))$
    $SA^{12}, LCP^{12} := DC3\text{-}LCP(R)$ *// $R :=$ sequence of all n_i of P*
    $P := \{(SA^{12}[j], j+1) : j \in [0, \left\lceil \frac{2 \cdot n}{3} \right\rceil)\}$ *// j holds $0 \leq j < |SA^{12}|$*
    P.`Sort`(by first component $SA^{12}[j]$) *// lexrank $r_i :=$ lexname n_i*
    while $\exists(i, r_i) \in P : 0 \leq i < \left\lfloor \frac{|P|+1}{2} \right\rfloor$ **do** $(i, r_i) := (3 \cdot i + 1, r_i)$
    while $\exists(i, r_i) \in P : \left\lfloor \frac{|P|+1}{2} \right\rfloor \leq i < |P|$ **do** $(i, r_i) := (3 \cdot (i - \left\lfloor \frac{|P|+1}{2} \right\rfloor + 2, r_i)$
  else P.`Sort`(by first component i) *// lexrank $r_i :=$ lexname n_i*
  // $ISA^{12} := ISA^{12}_{mod1} \circ ISA^{12}_{mod_2} :=$ sequence of second component n_i of P
  $S_0 := \{(T[i], T[i+1], r_{i+1}, r_{i+2}, i) : i \bmod 3 \equiv 0, (i+1, r_{i+1}) \wedge (i+2, r_{i+2}) \in P\}$
  $S_1 := \{(r_i, T[i], r_{i+1}, i) : i \bmod 3 \equiv 1, (i, r_i) \wedge (i+1, r_{i+1}) \in P\}$
  $S_2 := \{(r_i, T[i], T[i+1], r_{i+2}, i) : i \bmod 3 \equiv 2, (i, r_i) \wedge (i+2, r_{i+2}) \in P\}$
  S_0.`Sort`(by first component $T[i]$ and third component r_{i+1})
  S_1.`Sort`(by first component r_i)
  S_2.`Sort`(by first component r_i)
  $S :=$ `Merge`(S_0, S_1, S_2) with the comparison function:
    $(T[i], T[i+1], r_{i+1}, r_{i+2}, i) \in S_0 \leq (r_j, T[j], r_{j+1}, j) \in S_1$
    $:\Leftrightarrow (T[i], r_{i+1}) \leq (T[j], r_{j+1})$
    $(T[i], T[i+1], r_{i+1}, r_{i+2}, i) \in S_0 \leq (r_j, T[j], T[j+1], r_{j+2}, j) \in S_2$
    $:\Leftrightarrow (T[i], T[i+1], r_{i+2}) \leq (T[j], T[j+1], r_{j+2})$
    $(r_i, T[i], r_{i+1}, i) \in S_1 \leq (r_j, T[j], T[j+1], r_{j+2}, j) \in S_2$
    $:\Leftrightarrow r_i \leq r_j$
    ℓ_1.`PushBack`(`construct`$_{\ell_1}$())
    ℓ_2.`PushBack`(`construct`$_{\ell_2}$())
  while ℓ_2.`NotEmpty`() **do**
    $(r_i, r_j) :=$ `prepare`$_{\ell_3}(\ell_1$.`HasNext`$())$, $\Delta_2 := \ell_2$.`HasNext`()
    Z.`PushBack`$((id, r_i, \Delta_2), ((id+1), r_j, \Delta_2))$
  Z.`Sort`(by second component r)
  while Z.`NotEmpty`() **do** Z'.`PushBack`$((id,\ SA^{12}[r] + \Delta_2) : (id, r, \Delta_2) \in Z)$
  Z'.`Sort`(by second component $SA^{12}[r] + \Delta_2$)
  while Z'.`NotEmpty`() **do**
    Z''.`PushBack`$((id, ISA^{12}[r'_i]) : (id, r'_i) \in Z'$ or $(id, ISA^{12}[r'_j - 1]) : (id, r'_j) \in Z')$
  Z''.`Sort`(by first component id)
  while Z''.`NotEmpty`() **do**
    ℓ_3.`PushBack`$(RMQ_{LCP^N}[id, r_i, r_j] : (id, r_i) \in Z'' \wedge (id, r_j) \in Z'')$
  $LCP_T := \ell_1 \oplus 3 \cdot \ell_2 \oplus \ell_3$
  return SA_T, LCP_T *// index component of $s : s \in S$ represents SA_T*

Algorithm 5: Subroutines as required for *DC3-LCP*

```
1′  construct_{ℓ1}() begin
2′      if i ∈ S0 ∧ j ∈ S0 then
3′          return (T[i] ≠ T[j]) ? 0 : (T[i+1] = T[j+1]) ? 1 : 2
4′      else if i ∈ S0 ∧ j ∈ S1 then
5′          return (T[i] = T[j])
6′      else if i ∈ S0 ∧ j ∈ S2 then
7′          return (T[i] ≠ T[j]) ? 0 : (T[i+1] = T[j+1]) ? 1 : 2
8′      else if i ∈ S1 ∧ j ∈ S0 then
9′          return (T[i] = T[j])
10′     else if i ∈ S2 ∧ j ∈ S0 then
11′         return (T[i] ≠ T[j]) ? 0 : (T[i+1] = T[j+1]) ? 1 : 2
12′     else return 0 // in case of i ∈ {S1, S2} ∧ j ∈ {S1, S2}
13′ construct_{ℓ2}() begin
14′     if i ∈ S0 ∧ j ∈ S0 then
15′         return ℓ1.Last = 2 ? RMQ_{LCP^12}[r_{i+2}, r_{j+2} − 1] : 0
16′     else if i ∈ S0 ∧ j ∈ S1 then
17′         return ℓ1.Last = 1 ? RMQ_{LCP^12}[r_{i+1}, r_{j+1} − 1] : 0
18′     else if i ∈ S0 ∧ j ∈ S2 then
19′         return ℓ1.Last = 2 ? RMQ_{LCP^12}[r_{i+2}, r_{j+2} − 1] : 0
20′     else if i ∈ S1 ∧ j ∈ S0 then
21′         return ℓ1.Last = 1 ? RMQ_{LCP^12}[r_{i+1}, r_{j+1} − 1] : 0
22′     else if i ∈ S2 ∧ j ∈ S0 then
23′         return ℓ1.Last = 2 ? RMQ_{LCP^12}[r_{i+2}, r_{j+2} − 1] : 0
24′     else return LCP^12[r_j − 1] // in case of i ∈ {S1, S2} ∧ j ∈ {S1, S2}
25′ // Δ1 := element out of ℓ1
26′ prepare_{ℓ3}(Δ1) begin
27′     if i ∈ S0 ∧ j ∈ S0 then
28′         return Δ1 = 2 ? (r_{i+2} − 1, r_{j+2} − 1) : (0, 0)
29′     else if i ∈ S0 ∧ j ∈ S1 then
30′         return Δ1 = 1 ? (r_{i+1} − 1, r_{j+1} − 1) : (0, 0)
31′     else if i ∈ S0 ∧ j ∈ S2 then
32′         return Δ1 = 2 ? (r_{i+2} − 1, r_{j+2} − 1) : (0, 0)
33′     else if i ∈ S1 ∧ j ∈ S0 then
34′         return Δ1 = 1 ? (r_{i+1} − 1, r_{j+1} − 1) : (0, 0)
35′     else if i ∈ S2 ∧ j ∈ S0 then
36′         return Δ1 = 2 ? (r_{i+2} − 1, r_{j+2} − 1) : (0, 0)
37′     else return (r_i − 1, r_j − 1) // in case of i ∈ {S1, S2} ∧ j ∈ {S1, S2}
```

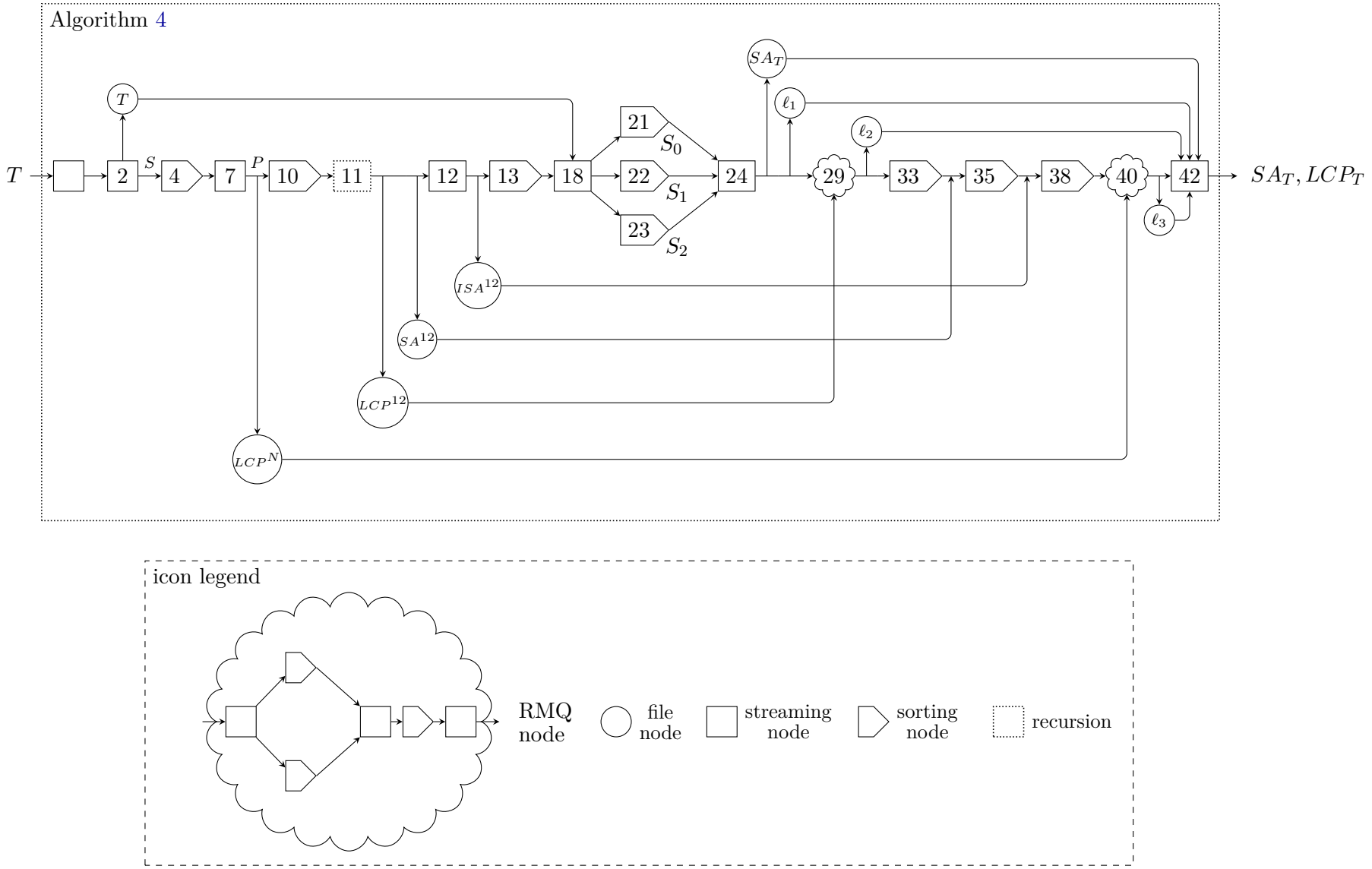


Figure 6: Upper box: pipelining model of the *DC3-LCP* algorithm (as described in algorithm 4) shown as a flow graph.
Lower box: meaning of symbols used in the graph. The numbers inside the symbols refer to the line numbers in algorithm 4
.

6 Experimental Evaluation

6.1 Implementation Details

The lion's share of this work was the implementation of the *DC3* and *DC3-LCP* algorithm which has been developed using the C++ programming language together with the STXXL, an implementation of the C++ Standard Template Library STL for external memory computation [12]. The list below gives a very short outline (from the programmer's perspective) of components we often needed in the implementation as supplied by the STXXL API available in the prerelease version 1.4.0 at `http://tbingmann.de/2013/stxxl/`.

1. **Stxxl stream interface** as described in [12] enables scanning similar to STL Input iterators. As an input iterator, an STXXL stream object can be dereferenced to refer to some object (i. e. `*stream_obj`) and can be preincremented (i. e. `++stream_obj`) to proceed to the next object in the stream. Iff the end of the stream is reached, the boolean member function `empty`() returns true.
2. **Stxxl deque2 container** is our external file node without random access as described in the STXXL API under Modules → STL-user layer → Containers. The deque2 object provides the operations `push_front`() and `push_ back`() to add new elements and `pop_front`() and respectively `pop_back`() to access elements. Scanning a deque2 object is possible in both directions by calling `get_stream`() or `get_reverse_stream`() which itself returns an STXXL stream object. To support overlapping of I/O accesses and computation, a deque2 object uses a write and prefetch block pool. The class definition `stxxl::deque2<ValueType, BlockSize, AllocStrategy, SizeType>` allows to specify the stored type of object (must be a POD), the block size B, an adapted disc allocation scheme and a size data type.
3. **Stxxl sorter container** as a stream layer is our external, pipelined sorting node which has the interface of an STXXL stream as described in [12]. According to the STXXL API under Modules → STL-user layer → Containers, the sorter container combines the two classes of `runs_creator` and `runs_merger` from the stream packages into a two-phase container, as already described in the pipelining chapter 3.3. Hence, in the first phase, the container is filled with elements using `push`(). To start the second phase, call `sort`. This finishes the first phase and sorts all elements in the container. Accessing the sorted elements using `*sorter` to get the top item, `++sorter` to proceed to the next one and `empty`() to check for the end of the stream. The class definition `class stxxl::sorter<ValueType, CompareType, BlockSize, AllocStrategy>` expects the stored type of object (must be a POD), a defined comparator (by overloading the function call operator `operator`()), the block size B and an adapted disc allocation scheme.

6.2 Experimental Settings

To close the gap between theory and practice, we measured the performance of *DC3* and *DC3-LCP* by testing various input instances with different properties. The following list of instances presents some details.

1. **Random Alphabet** is a random string T over the lower case letters of the classical latin alphabet $\Sigma := [\mathtt{\$}, \mathtt{a}, \mathtt{b}, \mathtt{c}, \ldots, \mathtt{z}]$.
2. **Wikipedia** is a copy of the English Wikimedia Wiki in the form of the Wikitext source and metadata embedded in XML available at `http://dumps.wikimedia.org/backup-index.html`. The XML dump we used is dated `enwiki-20130102`. We consider this instance as a very practical one.
3. **Gutenberg** is a concatenation of all free ebooks (in ASCII code) from `http://www.gutenberg.org/robot/harvest` as available in September 2012. Gutenberg can be described as a realistic real-world instance.

4. **Human Genome** is composed of the human genome assembly files "hg19" published on UCSC Genome Browser website http://genome.ucsc.edu/. All files were converted so that T consists of characters over the alphabet $\Sigma := [\$, \mathtt{A}, \mathtt{G}, \mathtt{C}, \mathtt{T}, \mathtt{N}]$ only.
5. **Fibonacci Word** is a recursively defined string T over $\Sigma := [\$, \mathtt{a}, \mathtt{b}]$. Let $S_{(0)} := \mathtt{a}$ and $S_{(1)} := \mathtt{b}$. Then, the recurrence relation of a Fibonacci word is generally defined as $S_{(n)} := S_{(n-2)} \circ S_{(n-1)}$. Due to its fixed frequency Fibonacci is a rather artificial instance and therefore less realistic for real-world data. Interestingly, Fibonacci Word has high LCP values.
6. **Unary Word** is referred to as a string T over the alphabet $\Sigma := [\$, \mathtt{a}]$, i. e. a string containing nothing but the letter 'a'. Unary Word has zero entropy and is unrealistic in real-world applications. The only special thing about Unary Word is that the recursion depth reaches its maximum of $log_3(|T|)$.

All experiments on the input instances mentioned above were executed on the InstitutsCluster II computer (see http://www.scc.kit.edu/dienste/ic2.php for more detailed information). We used the nodes with 2 × Octa-Core Intel Xeon E5-2670 CPUs (Sandy Bridge) clocked at 2.6 GHz having 8 × 256 KB L2-Cache each and providing 2 × 1 TB hard disk space by at least two disks. Each instance was computed five times due to possibly inhomogeneous disk compositions within the nodes. We recorded the fastest time each run. In all experiments the available main memory M was restricted to 1 GiB, the block size B was set to fixed 1 MiB (the best possible value for B which we have experimentally determined on our specific hardware and with our implementation). Additionally, we reserved a single node for every test instance run exclusively to rule out possible side effects. The tested input instances cover sizes between 2^{24} and 2^{32} since the used internal data types allow input instances up to 2^{32} Bytes. Our implementation was compiled with the GNU compiler `g++` in version 4.7.2 with `-O3` optimization and `-march=native` to enable all instruction subsets supported by the machine.

6.3 Verification

To ensure that the computed suffix array SA and LCP array are correct, the results of every instance were verified by other algorithms. We used a simple and fast suffix array checker for external memory as described in [10]. To prove the LCP array's accuracy we used Kasai's semi-external linear time LCP array construction algorithm [23] which needs the suffix array SA and the text T as an input. To make absolutely sure that the main memory consumption complies with the requirement of 1 GiB, we used `malloc_count`, a runtime memory usage analysis and profiling tool (available on http://panthema.net/2013/malloc_count/) which measures the amount of allocated memory of a program at run-time.

6.4 Performance and I/O Volume Measurements

In general, open symbols refer to *DC3*, filled symbols refer to *DC3-LCP* in this section. In figure 7 - figure 12 we distinguish between the construction time depending on the input size (left column) and the I/O volume depending on the input size (right column). A multiplot (figure 15) on the construction time of every instance of both algorithms follows a multiplot of every instance but both algorithms separated (figure 13 and figure 14).

Construction Time **I/O Volume**

Figure 7: Random Alphabet

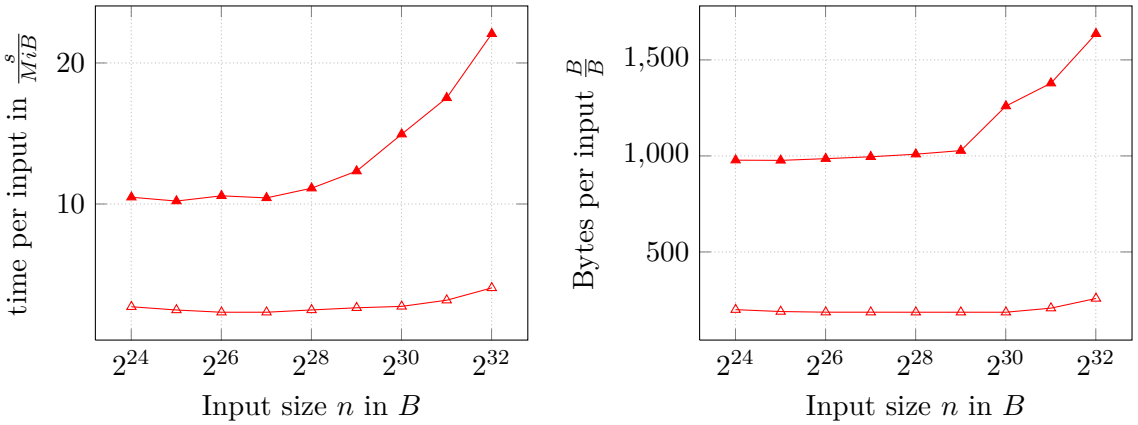


Figure 8: Wikipedia

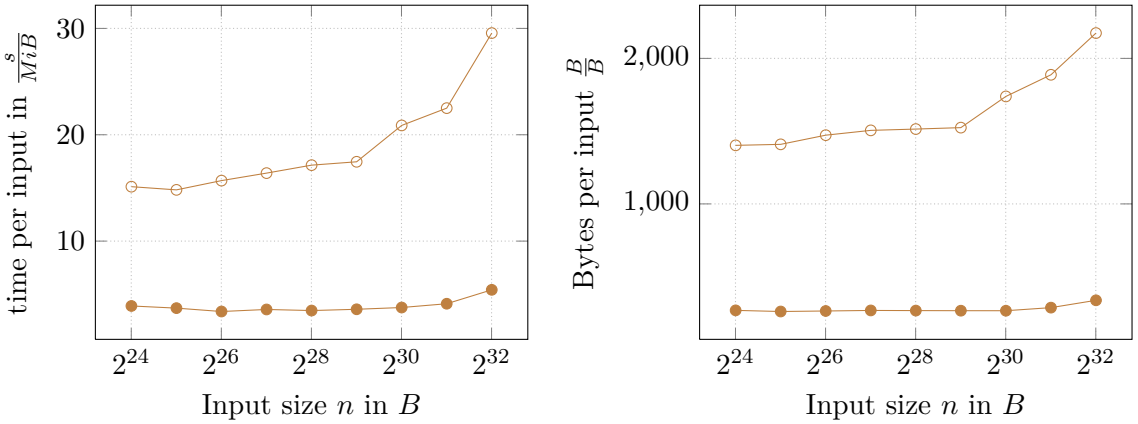
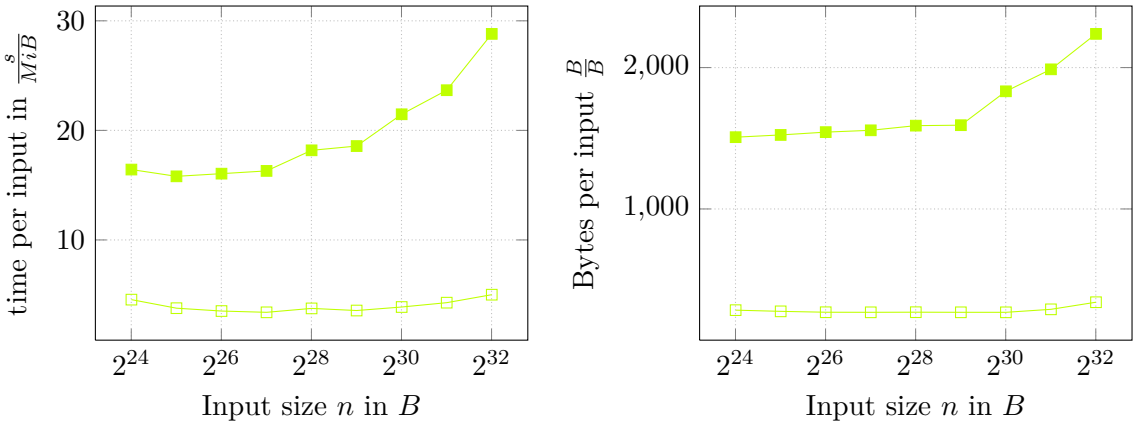


Figure 9: Gutenberg



Construction Time **I/O Volume**

Figure 10: Human Genome

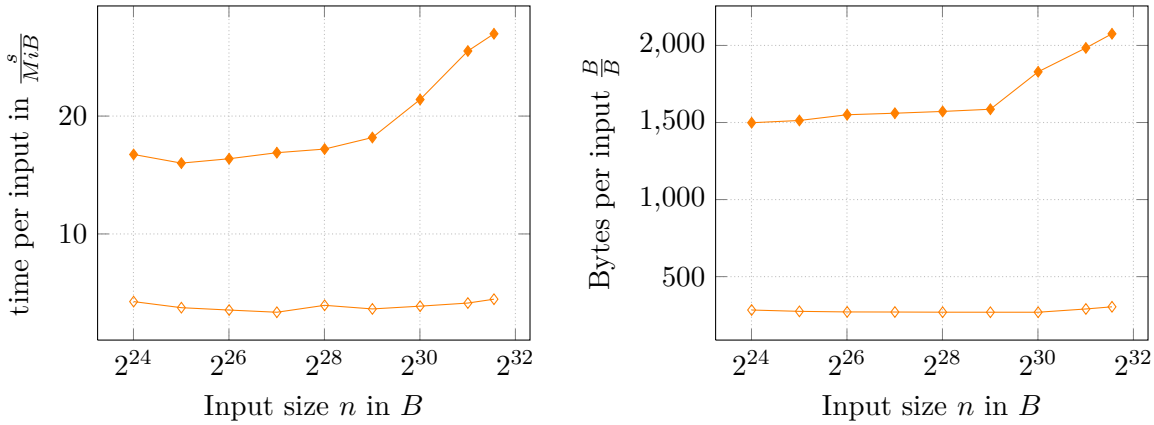


Figure 11: Fibonacci Word

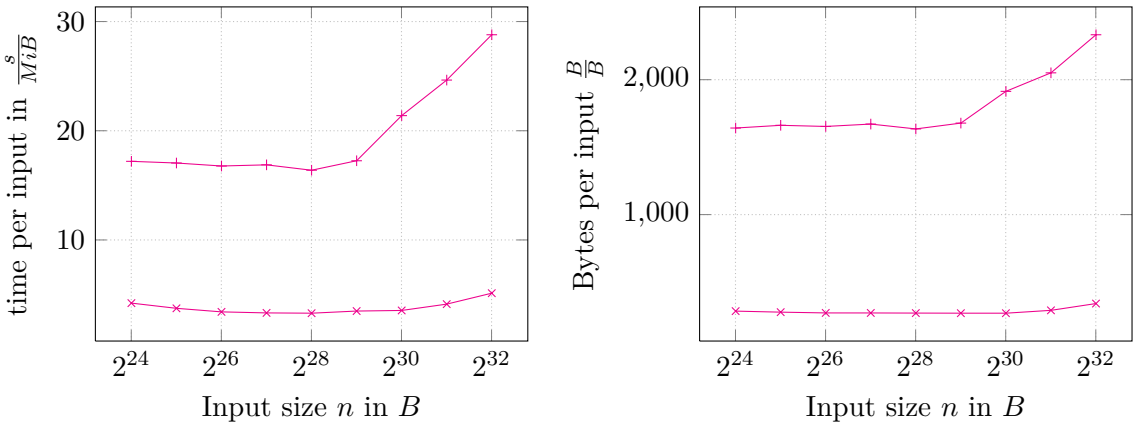


Figure 12: Unary Word

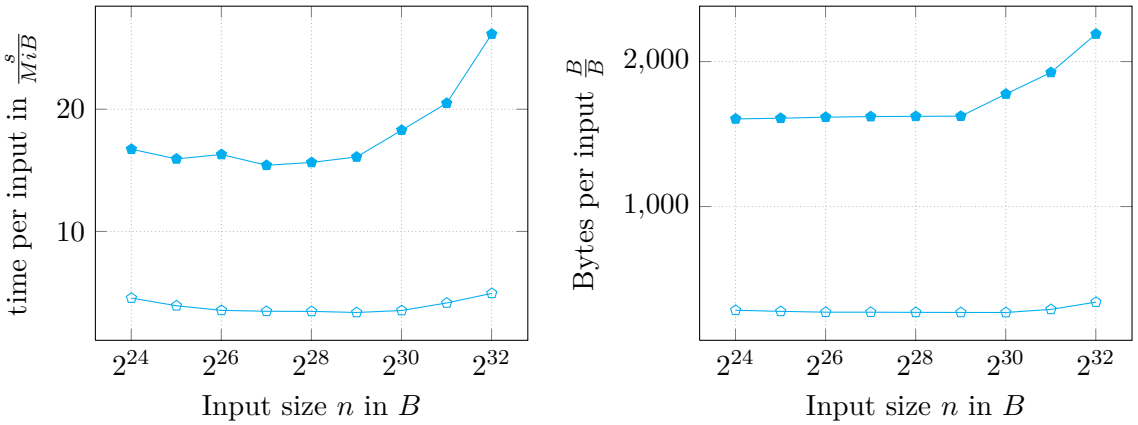


Figure 13: Plot of computation time of *DC3* on the entire set of test instances.

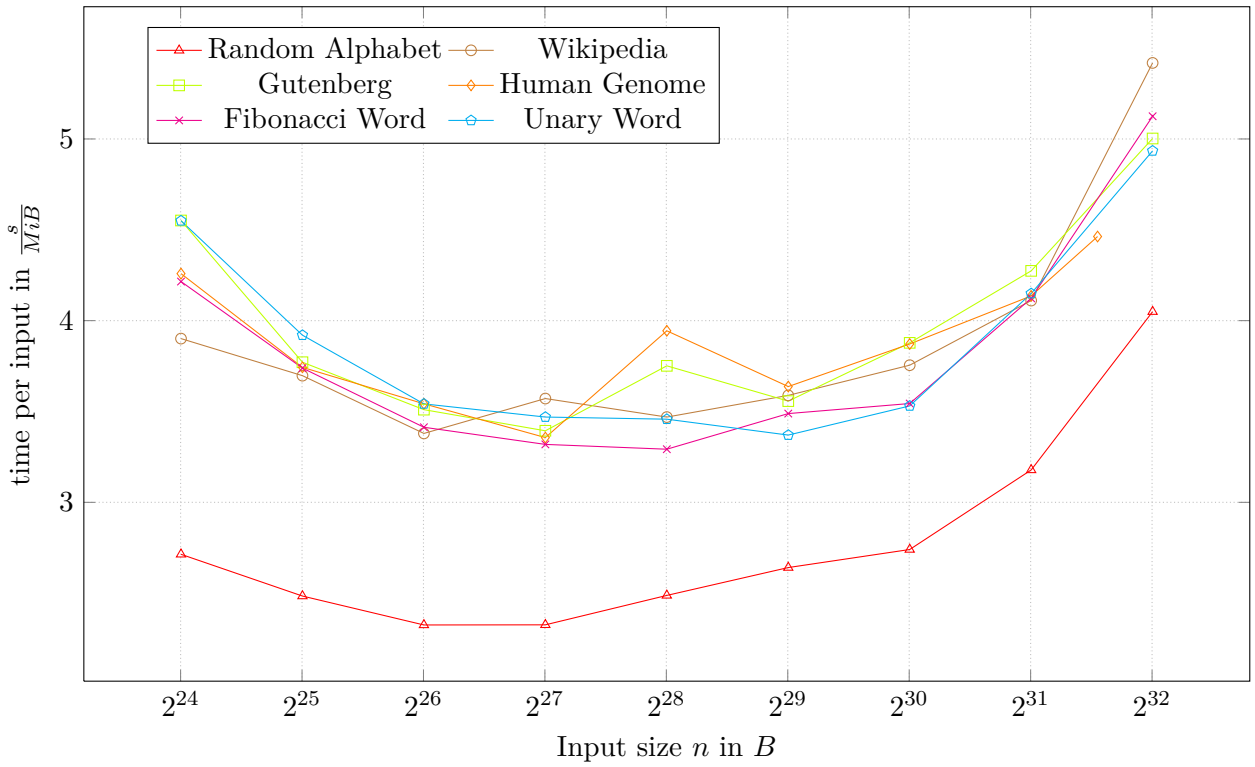


Figure 14: Plot of computation time of *DC3-LCP* on the entire set of test instances.

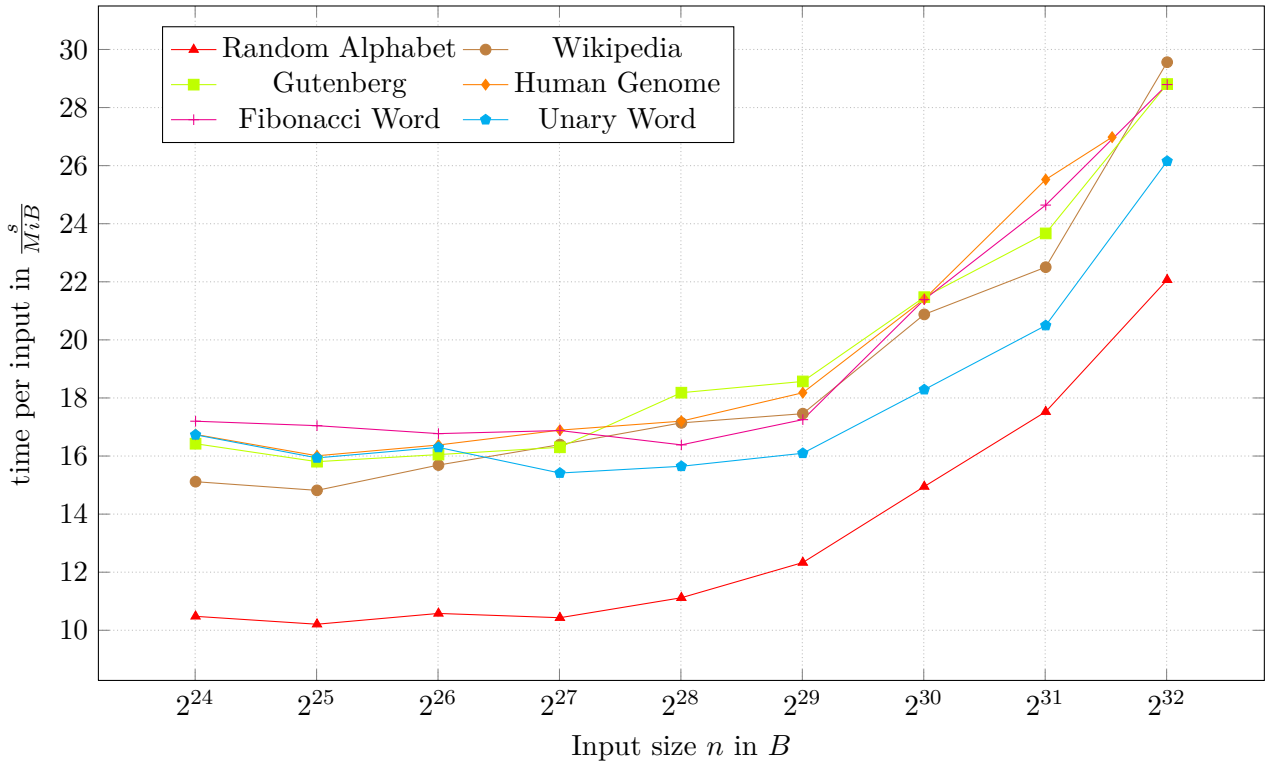
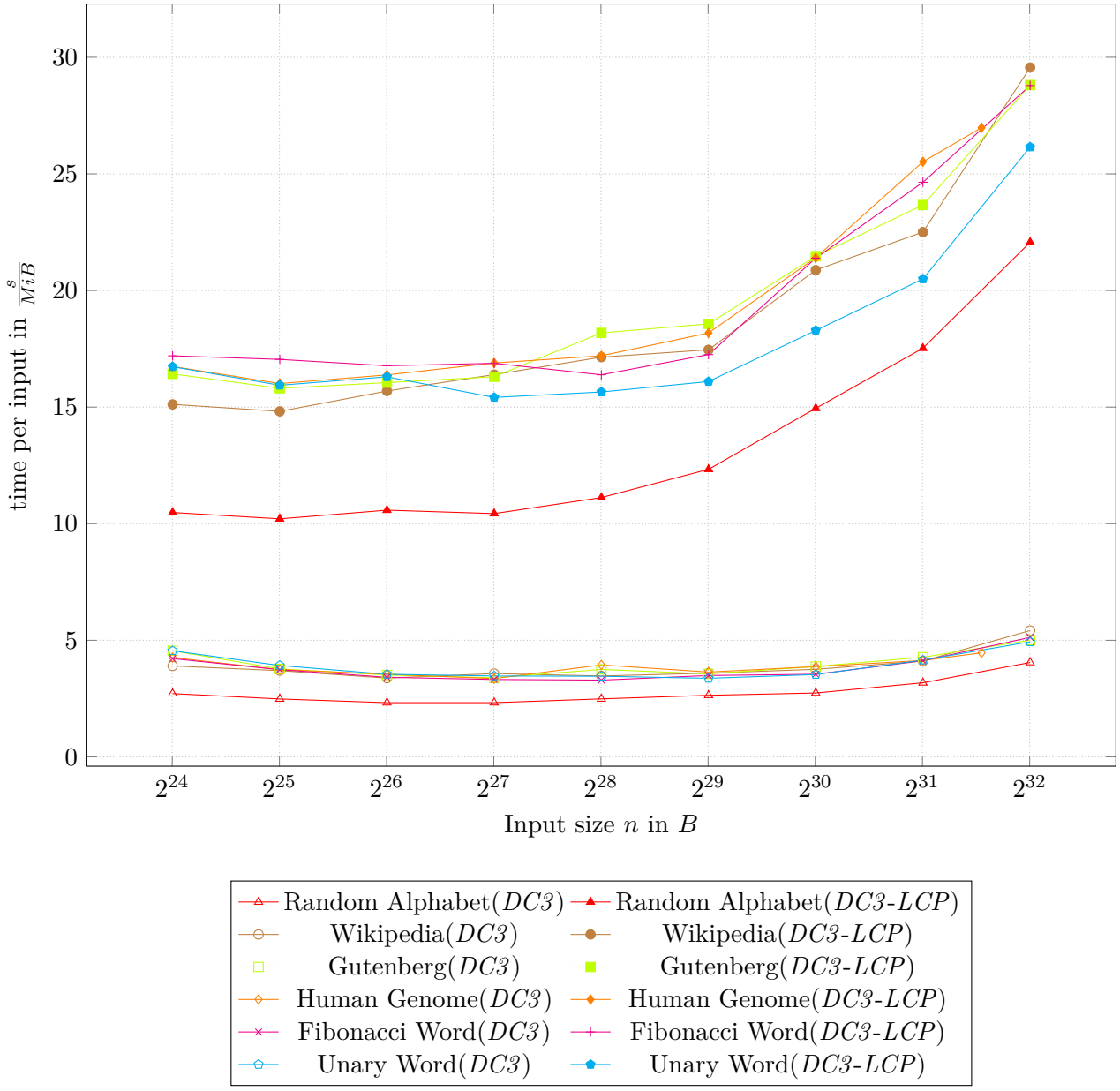


Figure 15: Plot of computation time of *DC3* and *DC3-LCP* on the entire set of test instances.



7 Discussion

7.1 Interpretation

In general, for input lengths between 2^{24}–2^{29}, *DC3-LCP* has a construction time and I/O volume which is about three to four times higher than *DC3* (observed construction time $\approx$ 3-5$\frac{s}{MiB}$ vs. $\approx$ 11–18$\frac{s}{MiB}$). The considered input lengths between 2^{30}-2^{32} showed a construction time and I/O volume which is about four to seven times higher than *DC3* (observed construction time $\approx$ 4–5$\frac{s}{MiB}$ vs. $\approx$ 15–30$\frac{s}{MiB}$).

As expected, the Random Alphabet instance with a low recursion depth of 2 turned out to be the easiest instance for *DC3* and *DC3-LCP*. Unary Word with maximum recursion depth of 20 was asymptotically "easier" than every other input except for the Random Alphabet. For *DC3* and *DC3-LCP*, Wikipedia (recursion depth of 10), Gutenberg (recursion depth of 15), Human Genome (recursion depth of 15) and Fibonacci Word (recursion depth of 18) showed roughly equal asymptotical I/O volume. Surprisingly, Wikipedia needed the highest construction time over all tested input instances on *DC3-LCP* and *DC3*. Consequently, the recursion depth is not mainly responsible for a higher construction time.

One characteristic all the instances have in common is an asymptotically similar behaviour. While *DC3-LCP* scales linear for input sizes between 2^{24}–2^{29}, larger input sizes show a strong increase in the I/O volume and thus in their consumed construction time. We can compare the construction time and I/O volume of *DC3* and *DC3-LCP* for Wikipedia and Human Genome with the results presented in [5]. Practically they used the same instances partially as well as the same memory limitations and block size. Their results for *DC3* does not show any notable differences with the values we have measured. Surprisingly, their results for *DC3-LCP* indicate significant differences. Their I/O volume only amounts to about half of ours (Wikipedia with 2^{32}: $\approx$ 1100$\frac{B}{B}$ vs. $\approx$ 2150$\frac{B}{B}$, Human Genome nearly equal to Wikipedia). Therefore, the measured construction time is about 40% lower (Wikipedia with 2^{32}: $\approx$ 18$\frac{s}{MiB}$ vs. $\approx$ 30$\frac{s}{MiB}$, Human Genome nearly equal to Wikipedia). Their asypmptotical behaviour is indeed the same, however, the gradient of their curves is not as steep as ours. The technique with which they achieve this is beyond our knowledge.

7.2 Future Work

The *DC3* in EM has been parallelized [3] (in contrast to *eSAIS* which probably cannot be parallelized). This approach could be applied on *DC3-LCP* as well to provide today's most promising approach for parallel and distributed construction of large text indexes. One can prove that the I/O volume is optimal regarding the difference cover modulo 7 (*DC7*) [11, 27]. Reinforcing this fact by experiments seems promising. These issues still remain open.

8 References

[1] *SPAA 2003: Proceedings of the Fifteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 7-9, 2003, San Diego, California, USA (part of FCRC 2003)* (2003), ACM.

[2] Arge, L., Ferragina, P., Grossi, R., and Vitter, J. S. On sorting strings in external memory (extended abstract). In *STOC* (1997), F. T. Leighton and P. W. Shor, Eds., ACM, pp. 540–548.

[3] Beckmann, A., Dementiev, R., and Singler, J. Building a parallel pipelined external memory algorithm library. In *IPDPS* (2009), IEEE, pp. 1–10.

[4] Bender, M. A., and Farach-Colton, M. The LCA problem revisited. In *LATIN* (2000), G. H. Gonnet, D. Panario, and A. Viola, Eds., vol. 1776 of *Lecture Notes in Computer Science*, Springer, pp. 88–94.

[5] Bingmann, T., Fischer, J., and Osipov, V. Inducing suffix and LCP arrays in external memory, 2013.

[6] Burkhardt, S., and Kärkkäinen, J. Fast lightweight suffix array construction and checking. In *CPM* (2003), R. A. Baeza-Yates, E. Chávez, and M. Crochemore, Eds., vol. 2676 of *Lecture Notes in Computer Science*, Springer, pp. 55–69.

[7] Burrows, M., Wheeler, D. J., Burrows, M., and Wheeler, D. J. A block-sorting lossless data compression algorithm. Tech. Rep. 124, Digital System Research Center, Palo Alto, May 1994.

[8] Crauser, A., and Ferragina, P. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica 32*, 1 (2002), 1–35.

[9] Dementiev, R. *Algorithm engineering for large data sets.* PhD thesis, 2006.

[10] Dementiev, R., Kärkkäinen, J., Mehnert, J., and Sanders, P. Better external memory suffix array construction. In *ALENEX/ANALCO* (2005), C. Demetrescu, R. Sedgewick, and R. Tamassia, Eds., SIAM, pp. 86–97.

[11] Dementiev, R., Kärkkäinen, J., Mehnert, J., and Sanders, P. Better external memory suffix array construction. *ACM Journal of Experimental Algorithmics 12* (2008).

[12] Dementiev, R., Kettner, L., and Sanders, P. : Standard template library for XXL data sets. In *ESA* (2005), G. S. Brodal and S. Leonardi, Eds., vol. 3669 of *Lecture Notes in Computer Science*, Springer, pp. 640–651.

[13] Döring, A., Weese, D., Rausch, T., and Reinert, K. Seqan an efficient, generic C++ library for sequence analysis. *BMC Bioinformatics 9* (2008).

[14] Fischer, J. Optimal succinctness for range minimum queries. *CoRR abs/0812.2775* (2008).

[15] Fischer, J. Inducing the LCP-array. In *WADS* (2011), F. Dehne, J. Iacono, and J.-R. Sack, Eds., vol. 6844 of *Lecture Notes in Computer Science*, Springer, pp. 374–385.

[16] Fischer, J., and Heun, V. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *CPM* (2006), M. Lewenstein and G. Valiente, Eds., vol. 4009 of *Lecture Notes in Computer Science*, Springer, pp. 36–48.

[17] Fischer, J., and Heun, V. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing 40*, 2 (2011), 465–492.

[18] Gog, S., and Ohlebusch, E. Fast and lightweight LCP-array construction algorithms. In *ALENEX* (2011), M. Müller-Hannemann and R. F. F. Werneck, Eds., SIAM, pp. 25–34.

[19] Gonnet, G. H., Baeza-Yates, R. A., and Snider, T. New indices for text: Pat trees and Pat arrays. In *Information Retrieval: Data Structures & Algorithms*. 1992, pp. 66–82.

[20] Guigó, R., and Gusfield, D., Eds. *Algorithms in Bioinformatics, Second International Workshop, WABI 2002, Rome, Italy, September 17-21, 2002, Proceedings* (2002), vol. 2452 of *Lecture Notes in Computer Science*, Springer.

[21] Heun, V. Skriptum zur Vorlesung Algorithmen und Sequenzen. 138.

[22] Kärkkäinen, J., and Sanders, P. Simple linear work suffix array construction. In *ICALP* (2003), J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, Eds., vol. 2719 of *Lecture Notes in Computer Science*, Springer, pp. 943–955.

[23] Kasai, T., Lee, G., Arimura, H., Arikawa, S., and Park, K. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *CPM* (2001), A. Amir and G. M. Landau, Eds., vol. 2089 of *Lecture Notes in Computer Science*, Springer, pp. 181–192.

[24] Manber, U., and Myers, E. W. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput. 22*, 5 (1993), 935–948.

[25] Manber, U., and Myers, G. Suffix arrays: A new method for on-line string searches. In *SODA* (1990), D. S. Johnson, Ed., SIAM, pp. 319–327.

[26] Manzini, G. Two space saving tricks for linear time LCP array computation. In *SWAT* (2004), T. Hagerup and J. Katajainen, Eds., vol. 3111 of *Lecture Notes in Computer Science*, Springer, pp. 372–383.

[27] Mehnert, J. External Memory Suffix Array Construction. Master's thesis, Saarland University, Nov 2004.

[28] Putze, F., and Sanders, P. Course Notes - Algorithm Engineering. 57–59.

[29] Vitter, J. S. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science 2*, 4 (2006), 305–474.

[30] Vitter, J. S., and Shriver, E. A. M. Algorithms for parallel memory ii: Hierarchical multilevel memories. *Algorithmica 12*, 2/3 (1994), 148–169.