An Efficient Algorithm for Computing the ith Letter of $\varphi^n(a)$

Jeffrey Shallit and David Swart *

Abstract

Let Σ be a finite alphabet, and let $\varphi : \Sigma^* \to \Sigma^*$ be a homomorphism, i.e., a mapping satisfying $\varphi(xy) = \varphi(x)\varphi(y)$ for all $x, y \in \Sigma^*$. Let $a \in \Sigma$, and let $i \geq 1$, $n \geq 0$ be integers. We give the first efficient algorithm for computing the ith letter of $\varphi^n(a)$. Our algorithm runs in time polynomial in the size of the input, i.e., polynomial in $\log n$, $\log i$, and the description size of φ. Our algorithm can be easily modified to give the distribution of letters in the prefix of length i of $\varphi^n(a)$. There are applications of our algorithm to computer graphics and biological modelling. If we consider finite-state transducers instead of homomorphisms, the corresponding problem is EXPTIME-hard.

1 Introduction

Let Σ be a finite alphabet. A *homomorphism* is a map φ from Σ^* to Σ^* such that $\varphi(xy) = \varphi(x)\varphi(y)$ for all $x, y \in \Sigma^*$. Let $a \in \Sigma$; we define $\varphi^0(a) = a$, and $\varphi^i(a) = \varphi(\varphi^{i-1}(a))$ for $i \geq 1$.

For $x \in \Sigma^*$, and $a \in \Sigma$, let $|x|$ denote the length of x, and let $|x|_a$ denote the number of occurrences of the letter a in x. We define the the *depth* d of a homomorphism φ to be the cardinality of its domain Σ, and the *width* w of a homomorphism φ to be the maximum value of $|\varphi(a)|$ over all $a \in \Sigma$.

Consider the following problem:

Given a homomorphism $\varphi : \Sigma^* \to \Sigma^*$, integers $n \geq 0$ and $i \geq 1$, and a letter $a \in \Sigma$, efficiently calculate the ith letter of $\varphi^n(a)$.

In this paper, we present the first algorithm which solves this problem in time bounded by a polynomial in the *size* of the input data. More precisely, the running time of our algorithm is polynomial in $\log n$, $\log i$, w, and d. Our model of computation is the familiar "naive bit complexity" model; see, for example, [?]. In this model, adding together two n-bit integers uses $O(n)$ bit operations, while multiplying two n-bit integers uses $O(n^2)$ bit operations.

Let p be the prefix of $\varphi^n(a)$ of length i. Our algorithm is easily modified to return the distribution of the first i letters in $\varphi^n(a)$ as the Parikh vector $\mathbf{p} = (m_b)_{b\in\Sigma}$, where $m_b = |p|_b$.

Iterated homomorphisms have been studied in the form of deterministic context-free Lindenmayer systems, or *D0L-systems* [?]. A D0L-system is a 3-tuple $G = \{\Sigma, \varphi, z\}$ where Σ is a finite alphabet, φ is a set of production rules, and $z \in \Sigma^*$ is the initial word or *axiom*. The *language* of a D0L-system G is defined as $L(G) = \{\varphi^n(z) \ : \ n \geq 0\}$.

A word of a D0L-language can be interpreted as instructions in a "turtle language" to draw an image [?]. Many have used D0L-systems to generate fractals and to model biological systems such as the branching structure of a tree [?, ?, ?]; see Figure 1. In this context, our new algorithm allows us to *efficiently calculate the structure of small twigs, without having to calculate the structure of the entire tree.*

Another application of our algorithm is the efficient computation of the ith letter of a fixed point of a homomorphism, which was stated as an open problem in [?]. A letter a is *mortal* if $\varphi^n(a) = \epsilon$ for some $n > 0$. We say a is *immortal* if it is not mortal. Let a be a letter such that $\varphi(a) = ax$, where x is a string containing at least one immortal letter. Then φ has a unique fixed point starting with a, of the form

$$\varphi^\omega(a) = a\, x\, \varphi(x)\, \varphi^2(x)\, \varphi^3(x) \cdots .$$

We can efficiently compute the ith letter of such a fixed point by using binary search to determine which factor the ith letter lies in, and then using our algorithm to find the appropriate letter within a factor. Such a binary search uses the subroutine GenericLength described below in Section ??.

This paper is based on results in the second author's M.Math. thesis [?].

2 Previous Work

While much attention has been paid to the applications of D0L-systems, a significant amount of study has also been devoted to their theoretical properties. Vitányi [?] provided a comprehensive collection of theoretical results up to 1980. For example, he discussed D0L-languages and how they relate to the Chomsky hierar-

*Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1. shallit@uwaterloo.ca, dmswart@uwaterloo.ca . Research supported by a grant from NSERC.

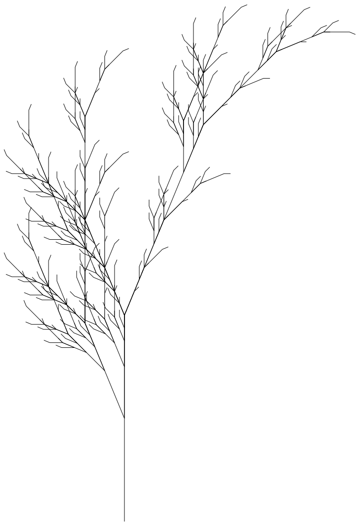
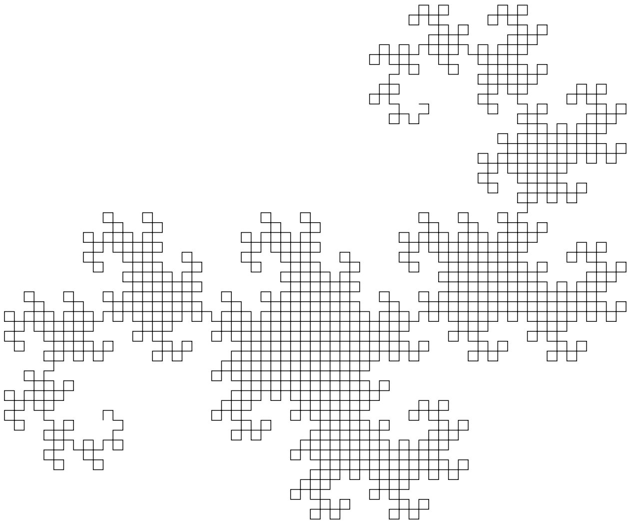

Figure 1: Geometric interpretations of strings produced by D0L-systems [?].

chy; other types of L-systems; and the behavior of the growth functions of D0L-systems.

The *growth function* f_G of a D0L-system G is defined by

$$f_G(n) = |\varphi^n(w)|.$$

Doucet [?] and Salomaa [?] devised methods to obtain an explicit formula for the growth function and showed that the growth function of any D0L-system is either polynomial or exponential. With respect to a homomorphism φ, we say a string $x \in \Sigma^*$ *grows polynomially* if $|\varphi^n(x)| = O(n^c)$ for some constant c. Otherwise, we say x *grows exponentially.* There is an efficient algorithm to determine if a letter grows exponentially [?].

Other authors have also discussed the complexity of decision problems involving D0L-systems. For example, a famous open problem in L-systems was the D0L-language equivalence problem: given D0L-systems, G_1 and G_2, does $L(G_1) = L(G_2)$? Culik and Fris [?] showed the decidability of this problem by giving an explicit algorithm.

Jones and Skyum [?] gave a polynomial time algorithm for solving the D0L membership problem: given G and x, is $x \in G$? Their result is somewhat orthogonal to ours, for in their situation the input is x, z, and φ, and they want to decide if there exists n with $x = \varphi^n(z)$. Note that $|x|$ could be as large as $w^n|z|$, where w is the width of φ. This quantity is *doubly exponential* in $\log n$. Later, Jones and Skyum [?] gave a DSPACE($\log^2 n$) algorithm to solve the D0L membership problem.

3 The BasicTreeDescent Subroutine

We first discuss a simple method to calculate the ith letter of $\varphi^n(a)$, called BasicTreeDescent. This algorithm is similar to one found by Jones and Skyum [?]. While its running time is *not* necessarily polynomial in $\log n$, $\log i$, w, and d, we use BasicTreeDescent to calculate the ith letter of $\varphi^n(a)$ for the cases in which n *is* polynomial in $\log i$, w, and d.

For an integer n and a letter $a \in \Sigma$, we view the sequence of words a, $\varphi(a)$, $\varphi^2(a)$, $\cdots$, $\varphi^n(a)$ as labels of the levels of an ordered tree. More precisely, the *derivation tree* of $\varphi^n(a)$ is the ordered tree of height n, with a as its root, such that every non-leaf node b has children labeled with the letters of $\varphi(b)$. Instead of computing $\varphi^n(a)$ by repeatedly applying φ to each successive string, we calculate the ith letter of $\varphi^n(a)$ by descending n levels down the derivation tree to the appropriate letter. The only difficulty lies in determining which subtree to descend.

Let Length(φ, n, a, i) be a procedure which returns the value $|\varphi^n(a)|$ and denote $\varphi(a)$ by the string $z = z_1 z_2 \cdots z_{|\varphi(a)|}$. BasicTreeDescent uses Length to compute $|\varphi^{n-1}(z_1)|, |\varphi^{n-1}(z_2)|, \cdots$ and by adding, computes $|\varphi^{n-1}(z_1)|, |\varphi^{n-1}(z_1 z_2)|, \cdots$ until a value t is determined such that

$$|\varphi^{n-1}(z_1 z_2 \cdots z_{t-1})| < i \leq |\varphi^{n-1}(z_1 z_2 \cdots z_t)|.$$

Once t has been calculated, the algorithm adjusts the values of n, a, and i to $n-1$, z_t, and $i - |\varphi^{n-1}(z_1 z_2 \cdots z_{t-1})|$ respectively and thus descends one level in the derivation tree. Of course, if $n = 0$, the algorithm simply returns a.

4 Our Improved Algorithm

The idea behind our new algorithm is to avoid descending n levels in the derivation tree for the cases where n

is not bounded by a polynomial in $\log i$, w, and d, by exploiting the recursive nature of the homomorphism. Basically, we descend the derivation tree until we encounter a letter for the second time and then shortcut to the node where this repeated letter occurs last. We continue descending, taking shortcuts whenever possible until we reach the ith letter of the bottom level.

4.1 Finding a Repeated Letter We discuss some notation concerning the path taken down the derivation tree. Let a_0 be the root of the derivation tree. Let a_j be the letter encountered after descending j levels and set x_j, y_j such that $\varphi(a_{j-1}) = x_j a_j y_j$. Therefore, after descending l levels, the sequence of letters encountered during the tree descent is $a_0, a_1, \cdots, a_l$ and the sequences of strings $x_1, x_2, \cdots, x_l$ and $y_1, y_2, \cdots, y_l$ describe the branches of the tree which were not followed.

We begin our algorithm by descending the derivation tree until we encounter some letter a_l for the second time, where l is the number of levels we have descended thus far. Let $l-q$ be the level containing the previous occurrence of a_l. We need only descend d levels before encountering such a letter a_l since $|\Sigma| = d$. More precisely, $a_l = a_{l-q}$ for some $l \leq d$.

By descending l levels, we have calculated new values for n, a, and i, which yield the same results, namely $i - |\varphi^{n-1}(x_1)\varphi^{n-2}(x_2)\cdots\varphi^{n-l}(x_l)|$, $n-l$, and a_l. We have also discovered strings x and y and an integer q such that $\varphi^q(a) = xay$. Specifically, x and y are $\varphi^{q-1}(x_{l-q+1})\cdots\varphi(x_{l-1})x_l$, and $y_l\varphi(y_{l-1})\cdots\varphi^{q-1}(y_{l-q+1})$ respectively.

4.2 Jumping Ahead Once a repeated letter a is found, the remainder of the algorithm calculates the last occurrence of the letter a in the tree descent and then either invokes the BasicTreeDescent algorithm or restarts our algorithm with new values of n and i.

Setting s and r such that $n = sq + r$ and $r < q$, we use the strings x, y, and the integers q, r, and s to describe $\varphi^n(a)$. We know $\varphi^q(a) = x\,a\,y$ and $\varphi^{2q}(a) = \varphi^q(x)\,x\,a\,y\,\varphi^q(y)$. Continuing this way, we see that

$$\varphi^{sq}(a) = \varphi^{(s-1)q}(x)\cdots\varphi^q(x)\,x\,a\,y\,\varphi^q(y)\cdots\varphi^{(s-1)q}(y).$$

Finally, applying φ^r to both sides yields

$$\varphi^n(a) = \underbrace{\varphi^{(s-1)q+r}(x)\cdots\varphi^{q+r}(x)\varphi^r(x)}_{X_1}\cdot \underbrace{\varphi^r(a)}_{X_2}\cdot \underbrace{\varphi^r(y)\varphi^{q+r}(y)\cdots\varphi^{(s-1)q+r}(y)}_{X_3}$$

Let GenericLength$(\varphi, \mathbf{x}, q, r, s, t, i)$ be a procedure which returns $\sum_{t\leq j<s}|\varphi^{qj+r}(x)|$. Rather than using the word x itself as input, we use the Parikh vector $\mathbf{x}$ to indicate the distribution of letters in x, that is, $\mathbf{x} = (m_j)_{1\leq j\leq d}$, where $m_j = |x|_{a_j}$.

We use GenericLength and Length to determine which of X_1, X_2, or X_3 the ith letter falls in and we handle these three cases separately.

Case 1: If x contains an exponentially growing letter, then Lemma **??** below states that $n = O(d\log i)$ and therefore, we can safely call BasicTreeDescent.

Otherwise, x grows polynomially. Using GenericLength, we do a binary search for a value of t such that the ith letter lies in the subword $\varphi^{tq+r}(x)$, more precisely,

$$|\varphi^{(s-1)q+r}(x)\cdots\varphi^{(t+1)q+r}(x)| < i$$

and

$$i \leq |\varphi^{(s-1)q+r}(x)\cdots\varphi^{tq+r}(x)|.$$

Once t is found, we set $n \leftarrow (t+1)q + r$ and $i \leftarrow i - |\varphi^{(s-1)q+r}(x)\cdots\varphi^{(t+1)q+r}(x)|$. As we prove in Lemma **??** below, adjusting the values of i and n this way lets us descend to the point in the derivation tree where the letter a occurs last.

Case 2: Since we have already calculated the length of X_1, we need only set $i \leftarrow i - |X_1|$, set $n \leftarrow r$, and then call BasicTreeDescent. We call BasicTreeDescent, since now $n = r < q \leq d$.

Case 3: Similarly to Case 1, we use GenericLength to do a binary search for a value t, such that the ith letter lies in the the subword $\varphi^{tq+r}(y)$, that is,

$$|\varphi^r(y)\varphi^{q+r}(y)\cdots\varphi^{(t-1)q+r}(y)| < i - |X_1X_2|$$

and

$$i - |X_1X_2| \leq |\varphi^r(y)\varphi^{q+r}(y)\cdots\varphi^{tq+r}(y)|.$$

Once the value of t is found, we set $n \leftarrow (t+1)q + r$. Since we wish to reduce i by the number of letters to the *left* of the substring $\varphi^{(t+1)q+r}(a)$, we set $i \leftarrow i - |\varphi^{(s-1)q+r}(x)\cdots\varphi^{(t+1)q+r}(x)|$.

If y is polynomially growing, then Lemma **??** below shows that the letter a is not encountered again. Hence, we continue descending the tree until the next shortcut is taken or until the bottom is reached.

Otherwise, y grows exponentially. By Lemma **??** below, $n = O(d\log i)$ and therefore we call BasicTreeDescent.

Figure **??** contains the pseudocode for the procedure Find which finds the ith letter of $\varphi^n(a)$. The algorithm assumes that $1 \leq i \leq |\varphi^n(a)|$.

procedure $\textsc{Find}(\varphi, n, a, i)$; // returns the ith letter of $\varphi^n(a)$ assuming $1 \le i \le |\varphi^n(a)|$.
$a_0 \leftarrow a$;

for $(l \leftarrow 1$ to $\infty)$ //executes for $\le d$ iterations
  $z \leftarrow \varphi(a) = z_1 z_2 \cdots z_{|\varphi(a)|}$;
  find smallest index t such that $i > |\varphi^{n-1}(z_1 z_2 \cdots z_{t-1})|$;
  $i \leftarrow i - |\varphi^{n-1}(z_1 z_2 \cdots z_{t-1})|$; $n \leftarrow n-1$; $a \leftarrow z_t$;
  $x_l \leftarrow z_1 \cdots z_{t-1}$; $a_l \leftarrow z_t$; $y_l \leftarrow z_{t+1} \cdots z_{|\varphi(a)|}$;

  if $(n \le d)$ then return $\textsc{BasicTreeDescent}(\varphi, n, a, i)$;
  else if $(a_j = a_l$ for some $j < l)$ then
    $q \leftarrow l - j$; $s \leftarrow \lfloor n/q \rfloor$; $r \leftarrow n - sq$;
    $\mathbf{x} \leftarrow$ Parikh vector of $\varphi^{q-1}(x_{l-q+1}) \cdots \varphi(x_{l-1}) x_l$;
    $\mathbf{y} \leftarrow$ Parikh vector of $y_l \varphi(y_{l-1}) \cdots \varphi^{q-1}(y_{l-q+1})$;
    $S_1 \leftarrow \textsc{GenericLength}(\varphi, \mathbf{x}, q, r, s, 0, i)$; //Length of X_1.
    $S_2 \leftarrow \textsc{Length}(\varphi, r, a, i - S_1)$; //Length of X_2.

    if $(i \le S_1)$ then //Case 1
      if (x grows exponentially) then return $\textsc{BasicTreeDescent}(\varphi, n, a, i)$;
      use $\textsc{GenericLength}$ to do a binary search for t such that
        $|\varphi^{(s-1)q+r}(x) \cdots \varphi^{(t+1)q+r}(x)| < i \le |\varphi^{(s-1)q+r}(x) \cdots \varphi^{tq+r}(x)|$
      $i \leftarrow i -$ $\textsc{GenericLength}(\varphi, \mathbf{x}, q, r, s, t+1, i)$;
      $n \leftarrow (t+1)q + r$;
      return $\textsc{Find}(\varphi, n, a, i)$;
    else if $(i \le S_1 + S_2)$ then //Case 2
      return $\textsc{BasicTreeDescent}(\varphi, r, a, i - S_1)$;
    else // Case 3
      use $\textsc{GenericLength}$ to do a binary search for t such that
        $|\varphi^r(y)\varphi^{q+r}(y) \cdots \varphi^{(t-1)q+r}(y)| < i - S_1 - S_2 \le |\varphi^r(y)\varphi^{q+r}(y) \cdots \varphi^{tq+r}(y)|$
      $i \leftarrow i -$ $\textsc{GenericLength}(\varphi, \mathbf{x}, q, r, s, t+1, i)$;
      $n \leftarrow (t+1)q + r$;
      if (y grows exponentially) then return $\textsc{BasicTreeDescent}(\varphi, n, a, i)$;
      else return $\textsc{Find}(\varphi, n, a, i)$;

Figure 2: The procedure $\textsc{Find}$

5 Length and GenericLength

Before moving on to the analysis of Find, we discuss details concerning the subroutines it calls. In particular, we show how GenericLength computes $\sum_{t \le j < s} |\varphi^{jq+r}(a)|$, how Length computes $|\varphi^n(a)|$, as well as providing a run-time estimate for these routines.

Let $\Sigma = \{a_1, a_2, \cdots a_d\}$. We define the *incidence matrix* of φ as $M(\varphi) = (m_{ij})_{1 \le i,j \le d}$, where $m_{ij} = |\varphi(a_j)|_{a_i}$. By multiplying the incidence matrix of φ by itself k times, we get a matrix whose m_{ij} entry equals the number of a_i's in $\varphi^k(a_j)$, that is, $M(\varphi)^k = M(\varphi^k)$ for $k \ge 0$. Hence, Length calculates $M(\varphi)^n$ and then sums the values of the column corresponding to a to yield $|\varphi^n(a)|$.

The well-known *binary method of exponentiation* (e.g., [?]) provides a method for raising a matrix to the nth power with $O(\log n)$ matrix multiplications.

We take some measures to avoid unnecessary calculations which may impede the performance of Length. For our purposes, it suffices to report $|\varphi^n(a)| \ge i$ without having to actually calculate $|\varphi^n(a)|$. Accordingly, after every matrix $M(\varphi^k)$ is calculated, we check whether the number of immortal letters in $\varphi^k(a)$ is less than i. If not, Length stops the calculation and reports that $|\varphi^n(a)| \ge i$. A letter b is *accessible* from a if $\varphi^n(a) = w_1 b w_2$ for some strings $w_1, w_2 \in \Sigma^*$ and for some $n \ge 0$. We restrict the alphabet Σ to only the letters accessible from a.

To summarize, when Length is executed, it constructs the incidence matrix $M(\varphi)$, and then uses the binary method of exponentiation to calculate $M(\varphi^n)$ while taking the precautions noted above. If $M(\varphi^n)$ is finally calculated, then Length returns the sum of the entries in the column corresponding to a.

Now we discuss how GenericLength returns the value of $\sum_{t \le j < s} |\varphi^{qj+r}(x)|$. An instrumental idea used in GenericLength is the ability to calculate matrices of the form $\mathbf{C}_n = \mathbf{I} + \mathbf{A} + \cdots + \mathbf{A}^{n-1}$ efficiently. An inductive argument shows that if

$$\mathbf{B} = \begin{bmatrix} \mathbf{A} & \mathbf{O} \\ \mathbf{I} & \mathbf{I} \end{bmatrix},$$

then

$$\mathbf{B}^n = \begin{bmatrix} \mathbf{A}^n & \mathbf{O} \\ \mathbf{C}_n & \mathbf{I} \end{bmatrix}.$$

Using the binary method of exponentiation again, we can compute $\mathbf{B}^n$ in $\log n$ matrix multiplications.

GenericLength begins by constructing $M(\varphi)$. Using the binary method of exponentiation, it calculates $\mathbf{Y} = M(\varphi^{tq+r})$, and $\mathbf{A} = M(\varphi^q)$. The technique mentioned above is used with $\mathbf{A}$ to compute the matrix $\mathbf{C}_n = M(\varphi^0) + M(\varphi^q) + \cdots + M(\varphi^{(s-t-1)q})$. Finally, we sum the entries in the column vector $\mathbf{Y}\mathbf{C}_n\mathbf{x}$ to obtain the value $\sum_{t \le j < s} |\varphi^{jq+r}(x)|$.

As in Length, GenericLength takes precautions to avoid unnecessary calculations. As before, we restrict the alphabet Σ to only the letters accessible from x, and we perform checks similar to the ones done in Length in order to stop the calculation when necessary and report that $\sum_{t \le j < s} |\varphi^{jq+r}(x)| \ge i$.

The following lemma states the running time of Length and GenericLength.

Lemma 5.1. *The number of bit operations used by* Length *is*

$$O((\log n) d^3 (d \log w + \log i)^2)$$

and the number of bit operations used by Generic-Length *is*

$$O((\log n) d^3 (d^2 \log w + \log i)^2).$$

Proof. Omitted due to space considerations.

6 Correctness and Analysis

We first prove the correctness of Find, and then discuss its running time.

Theorem 6.1. Find(φ, n, a, i) *returns the ith letter of* $\varphi^n(a)$.

Proof. We proceed by induction on n. When $n < d$, the algorithm calls BasicTreeDescent thus returning the correct answer. Otherwise, we assume that our algorithm works for values less than n. It remains to show that every time the value of n is reduced that a and i are modified in such a way that the ith letter of $\varphi^n(a)$ is the same.

The first part of the procedure descends the derivation tree one level at a time. Each time n is decreased by 1, the values for a, and i are z_t and $i - |\varphi^{n-1}(z_1 z_2 \cdots z_{t-1})|$ respectively, since the ith letter of $\varphi^n(a)$ lies in the subtree of z_t.

When a repeated letter a is found, we calculate the values for q, r, s, x and y such that for any $t < s$,

$$\begin{aligned} \varphi^n(a) = \; & \varphi^{(s-1)q+r}(x) \cdots \varphi^{tq+r}(x) \cdot \\ & \varphi^{tq+r}(a) \cdot \\ & \varphi^{tq+r}(y) \cdots \varphi^{(s-1)q+r}(y). \end{aligned}$$

Accordingly, for both Case 1 and Case 3, n is set to $(t+1)q + r$, and $|\varphi^{(s-1)q+r}(x) \cdots \varphi^{(t+1)q+r}(x)|$ is subtracted from i.

Therefore, each time n is reduced, the values for a and i are also modified correctly. Hence Find returns the ith letter of $\varphi^n(a)$.

Lemma **??** already tells us the running time of the subroutines LENGTH and GENERICLENGTH.

LEMMA 6.1. *The procedure* BASICTREEDESCENT *uses* $O(nw(\log n)d^3(d\log w + \log i)^2)$ *bit operations.*

Proof. We obtain the result by multiplying the running time of LENGTH by nw.

We need to show that whenever BASICTREEDESCENT is called, n is polynomial in $\log i$, w, or d. In our algorithm, we call BASICTREEDESCENT when $n \leq d$, when $n = r < q \leq d$, or when x or y grows exponentially. The following lemma shows that when x or y grows exponentially, then n is a polynomial in d and $\log i$.

LEMMA 6.2. *Let* $a \in \Sigma$, *and* $u, v \in \Sigma^*$ *such that* $\varphi^q(a) = uv$ *for some* $q < d$. *If the ith letter of* $\varphi^n(a)$ *lies in the substring* $\varphi^{n-q}(v)$, *and* u *contains an exponentially growing letter* b, *then* $n = O(d\log i)$.

Proof. If $\varphi^d(b)$ contains less than 2 exponentially growing letters, then $\varphi^j(b)$ contains at most one exponentially growing letter for all $j \geq 0$, which is impossible. Hence there are at least 2 exponentially growing letters in $\varphi^d(u)$. Therefore $|\varphi^n(u)|$ grows at least as fast as c^n where $c = 2^{1/d}$.

Since, $c^{n-q} \leq |\varphi^{n-q}(u)| < i$, then we know $(n-q)\log c < \log i$. Hence $n \leq (\log i)/(\log c) + q$. But $1/\log c = O(d)$ and $q < d$. Consequently, $n = O(d\log i)$.

We apply Lemma **??** to Case 1 by setting $u = x$ and $v = ay$. For Case 3, we know a grows exponentially whenever y does, Hence, we apply Lemma **??** with $u = xa$ and $v = y$.

We have running times for the subroutines that FIND calls, but FIND also calls itself. The following shows that FIND recursively calls itself at most d times by showing that each of the d letters can be the repeated letter of our algorithm at most one time.

LEMMA 6.3. *Let* q, r, *and* t *be integers such that* $r < q$, *and* $\varphi^q(a) = xay$ *for some* $x, y \in \Sigma^*$ *and* $a \in \Sigma$. *If* x *grows polynomially and* $i \leq |\varphi^{tq+r}(x)|$, *or if* y *grows polynomially and* $i > |\varphi^{tq+r}(xa)|$, *then descending the derivation tree of* $\varphi^{(t+1)q+r}(a)$ *to the ith letter encounters the letter* a *exactly once at the root.*

Proof. Consider the case where x grows polynomially and $i \leq |\varphi^{tq+r}(x)|$. Descending q levels takes us into the subword x. Hence, our descent takes an earlier branch than the path to the letter a in xay at level q.

Since no a is encountered before the tree descent leaves the path to the a at level q, a second a can only be encountered after leaving this path. But encountering another a would imply that a and hence x is exponentially growing, a contradiction.

Otherwise we have that y grows polynomially and $i > |\varphi^{tq+r}(xa)|$. Similarly, we descend q levels into the subword y. If our descent encounters an a after leaving the path to the letter a in xay at level q, then we know y grows exponentially, another contradiction.

Finally, we state our main result: the running time of FIND is polynomial.

THEOREM 6.2. *The number of bit operations used by* FIND *is*

$$O(\ (d+\log i)w(\log n)\ d^4(d\ \log w + \log i)^2 + (\log n)^2 d^4(d^2\log w + \log i)^2\ ).$$

Proof. Let L be the running time of LENGTH, and G, the running time of GENERICLENGTH.

At worst case, the "for" loop of FIND executes d times, calculating the length of the strings of subtrees w times, before one of the three cases is reached. Handling the cases takes at worst case $O(\log nG + L)$ time. This gives one instantiation of FIND a running time of

$$O(wdL + (\log n)G).$$

We know FIND recursively calls itself at most d times before reaching the base case and calling BASICTREEDESCENT with $n < d\log i$. The running time of BASICTREEDESCENT then is

$$O(nwL) = O(wd(\log i)L).$$

Hence the total running time of FIND is

$$O(d(wdL + (\log n)G) + wd(\log i)L).$$

Rearranging and substituting for G and L gives us the final running time.

The most significant difference between the running times of FIND and BASICTREEDESCENT is that there is only a $(\log n)^2$ term in the running time of FIND whereas there is a factor of n in the running time of BASICTREEDESCENT.

The upper bound we obtained in Theorem **??** overstates the actual running time obtained in practice. For most typical input data, FIND is rarely called more than once. It makes sense, then, to implement these procedures and analyze the actual running time of FIND for varying n, i, d, and w.

Table 1 reports the performance of FIND with $n = 1000$, and $n = 1000000$. The running times of BASICTREEDESCENT with $n = 1000$ are included for

Σ	d	w	FIND $n = 1000$			FIND $n = 1000000$			BTD $n = 1000$		
			i=100	10^4	10^6	i=100	10^4	10^6	i=100	10^4	10^6
exp	5	2	0.17	0.25	0.33	0.19	0.27	0.41	3.87	4.40	4.71
		10	0.47	0.75	1.17	0.42	0.59	0.86	9.90	10.35	10.65
	30	2	7.67	16.58	19.27	7.59	16.83	19.93	164.97	214.80	263.64
		10	7.24	10.89	14.13	7.19	10.78	14.09	151.58	200.73	211.40
poly	5	2	0.11	0.22	0.36	0.17	0.19	0.27	4.31	4.15	3.93
		10	0.26	0.30	0.53	0.40	0.41	0.46	9.75	18.19	42.05
	30	2	5.08	5.76	6.32	5.45	6.04	6.35	20.24	23.65	25.47
		10	4.79	4.96	6.27	5.33	5.62	5.68	59.30	116.15	76.08
mix	5	2	0.08	0.12	0.19	0.14	0.15	0.03	4.68	3.11	3.17
		10	0.18	0.23	0.26	0.24	0.33	0.34	7.82	8.34	7.54
	30	2	2.67	3.09	3.36	3.34	3.70	4.04	45.52	47.97	49.54
		10	6.67	9.39	11.98	6.78	9.53	12.10	151.26	161.09	159.70

Table 1: Performance of FIND and BASICTREEDESCENT in seconds

comparison. Each row of the table corresponds to a homomorphism with varying values of w and d and with alphabets of exponentially growing letters, polynomially growing letters, or both. For each homomorphism, we calculated the 100th, the 10000th, and the 1000000th letter of each string.

These results show a number of details concerning the performance of FIND on various inputs. The running times of both FIND and BASICTREEDESCENT have a high dependence on d. In practice however, d is not very large (typically less than 10 letters).

Notice that BASICTREEDESCENT does not perform as well as FIND does when $n = 1000$. For even larger n, the BASICTREEDESCENT would have an even worse performance compared with FIND. For example, obtaining the ith letter of $\varphi^{1000000}(a)$ using BASICTREEDESCENT would have taken about a day for each calculation. This shows that our new algorithm enables us to calculate letters of strings produced by homomorphisms which were not obtainable before, particularly those produced with large values of n.

7 Extensions

Our algorithm may be easily extended to calculate how many times each letter in Σ occurs in the prefix of $\varphi^n(a)$ of length i. The procedures LENGTH and GENERICLENGTH compute the lengths of strings which precede the ith letter of $\varphi^n(a)$. In the process of calculating these lengths, we already compute the Parikh vector of these strings. Computing the total distribution of the letters which precede the ith letter entails only summing the corresponding Parikh vectors each time we reduce i. By subtracting two Parikh vectors for prefixes of different lengths, we can also efficiently compute the letter distributions of subwords.

8 Finite-state Transducers

We turn our attention from homomorphisms to the iterated application of a finite-state transducer; see, for example, [?]. A *finite-state transducer*, or just *transducer* is a machine $T = (Q, \Sigma, \delta, q_0, \Delta, \lambda)$ where Q is a finite set of states, Σ is the input alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function, q_0 is the initial state, Δ is the output alphabet, and $\lambda : Q \times \Sigma \to \Delta^*$ is the output function. Given input $x = x_1x_2x_3 \cdots x_n \in \Sigma^*$, a finite-state transducer starts at state q_0 and at the ith step, it enters the state $q_i = \delta(q_{i-1}, x_i)$ and appends $\lambda(q_{i-1}, x_i)$ to the output. Hence on input $x_1x_2x_3 \cdots x_n$, a finite-state transducer outputs

$$\lambda(q_0, x_1)\lambda(q_1, x_2) \cdots \lambda(q_{n-1}, x_n).$$

Formally, we define TRANSDUCER to be the following decision problem.

Instance:	A finite-state transducer T which implements a function $f : \Sigma^* \to \Sigma^*$; integers n, i in binary; and letters a, b.
Question:	Does the i'th letter of $f^n(a)$ equal b?

Unlike the case with homomorphisms, which can be computed in polynomial time, TRANSDUCER is EXPTIME-hard as we will see in Theorem ??.

We define a *configuration string* as the non-blank tape contents of a Turing machine, where the current state and the square being scanned are combined into a special symbol and the symbol $\triangleright$ is appended to the end to signify the infinite amount of trailing blanks. For example, the string $(q_0, \mathtt{1})$ `0 1 0 0 1` $\triangleright$ represents the first configuration of a Turing machine on input `101001`.

LEMMA 8.1. *For any Turing machine M, we can create a transducer T which simulates one step of M by mapping a configuration string x to the configuration string which follows x in M's computation.*

Proof. We give an outline for the construction of T; the details are left to the reader. Essentially, by using the states of T to delay the output of each square by one step, we can determine whether the output should become an ordered pair as a result of the tape head moving left. Once the tape head has been handled, the remainder of T's computation simply outputs the rest of the string as it is received. The symbol $\triangleright$ is used to add more blanks to the right when necessary.

We now give a reduction from an EXPTIME-complete problem to TRANSDUCER

THEOREM 8.1. TRANSDUCER *is EXPTIME-hard.*

Proof. Let M_E be a Turing machine which recognizes an EXPTIME-complete language E in less than 2^{n^k} steps and finishes by writing a "yes" or a "no" symbol in the first square. Further, let x be an arbitrary input string for M_E. We specify an instance of TRANSDUCER as follows: let a be a new symbol; let T be a transducer which, on input a, outputs M_E's first configuration string, and which simulates one step of M_E's computation otherwise; let $b =$ "yes"; let $i = 1$; and let $n = 2^{|x|^k}$. Only $|x|^k$ digits are needed to specify n, and the specification of T depends only linearly on $|x|$. Further, $f^n(a)$ is the "yes" symbol if and only if M_E accepts x. Therefore, we have a polynomial reduction from E to TRANSDUCER and the result follows.

A well-known sequence produced by transducers is the Kolakoski sequence [**?**]:

$$\mathbf{K} = 12211212212211211221211\cdots$$

which has the property that the sequence of run-lengths of $\mathbf{K}$ is the same as $\mathbf{K}$ itself. The long range distribution of many of these sequences are unknown despite a great deal of attention [**?**]. Our result suggests an inherent difficulty in computing the digits of such sequences.

References

[1] E. Bach and J. Shallit. *Algorithmic Number Theory.* MIT Press, 1996.

[2] J. Berstel. *Transductions and Context-Free Languages.* B. G. Teubner, Stuttgart, 1979.

[3] I. Culik, K. and I. Fris. The decidability of the equivalence problem for D0L-systems. *Information and Control*, 35:20–39, 1977.

[4] F. M. Dekking. Recurrent sets. *Adv. in Math.* **44** (1982), 78–104.

[5] F. M. Dekking. What is the long range order in the Kolakoski sequence? In R. V. Moody, ed., *The Mathematics of Long-Range Periodic Order*, NATO Adv. Sci. Inst. Ser. C Math. Phys. Sci., Vol. 489, Kluwer, 1997, pp. 115–125.

[6] P. G. Doucet. The growth of word length in D0L-systems. In *Open House in Unusual Automata Theory (Aarhus University, Technical Report DAIMI PB-15)*, pages 83–94, 1973.

[7] N. D. Jones and S. Skyum. Complexity of some problems concerning L systems. *Math. Systems Theory* **13** (1979), 29–43.

[8] N. D. Jones and S. Skyum. A note on the complexity of general D0L membership. *SIAM J. Comput.* **10** (1981), 114–117.

[9] W. Kolakoski. Elementary Problem 5304. *Amer. Math. Monthly* **72** (1965), 674. Solution in **73** (1966), 681–682.

[10] P. Prusinkiewicz and J. Hanan. *Lindenmayer Systems, Fractals, and Plants.* Lecture Notes in Biomathematics, Vol. 79, Springer-Verlag, 1989.

[11] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants.* Springer-Verlag, 1990.

[12] A. Salomaa. On exponential growth in Lindenmayer systems. *Proc. Konin. Neder. Akad. Wet. Series A* **76** (1973), 23–29.

[13] J. Shallit. Number theory and formal languages. In D. A. Hejhal, J. Friedman, M. C. Gutzwiller, and A. M. Odlyzko, eds., *Emerging Applications of Number Theory*, IMA Volumes in Mathematics and Applications, Vol. 109, to appear.

[14] David Swart. *Calculating the ith Letter of $\varphi^n(a)$* M.Math. Thesis, Department of Computer Science, University of Waterloo, 1998 (in preparation).

[15] A. L. Szilard and R. E. Quinton. An interpretation for D0L systems by computer graphics. *The Science Terrapin* **4** (1979), 8–13.

[16] P. Vitányi. *Lindenmayer Systems: Structure, Languages, and Growth Functions.* Mathematical Centre Tracts, Vol. 96, Mathematisch Centrum, Amsterdam, 1980.