The Complexity of Finding SUBSEQ(A)

Stephen Fenner [*] Univ. of South Carolina

William Gasarch [†] Univ. of MD at College Park

Brian Postow [‡] Union College

Abstract

Higman showed that if A is any language then SUBSEQ(A) is regular. His proof was nonconstructive. We show that the result cannot be made constructive. In particular we show that if f takes as input an index e of a total Turing Machine M_e, and outputs a DFA for SUBSEQ($L(M_e)$), then $\emptyset'' \leq_{\mathrm{T}} f$ (f is Σ_2-hard). We also study the complexity of going from A to SUBSEQ(A) for several representations of A and SUBSEQ(A).

Keywords: Computability, computable function, recursive function, computably bounded, automata theory, subsequence, Higman, bounded queries

1 Introduction

Consider the string $x = aaba$. The string x has many subsequences, namely,

a, b, aa, ab, ba, aaa, aab, aba, and $aaba$.

Given a language A, the language SUBSEQ(A) is the set of all subsequences of strings in A. How do the complexity of A and SUBSEQ(A) compare?

The following are easy exercises for a course in automata theory.

1. Show that if A is regular then SUBSEQ(A) is regular.
2. Show that if A is context free then SUBSEQ(A) is context free.
3. Show that if A is c.e.[1] then SUBSEQ(A) is c.e.

What happens if A is decidable? Clearly if A is decidable then SUBSEQ(A) is c.e. But is SUBSEQ(A) decidable? A corollary of a theorem of Higman ([19] but also see the appendix for his proof and a new proof) supplies far more information:

If A is *any language whatsoever*, then SUBSEQ(A) is regular.

[*]University of South Carolina, Department of Computer Science and Engineering, Columbia, SC 29208. fenner@cse.sc.edu, Partially supported by NSF grant CCF-05-15269.

[†]University of Maryland, Dept. of Computer Science and Institute for Advanced Computer Studies, College Park, MD 20742. gasarch@cs.umd.edu, Partially supported by NSF grant CCR-01-05413

[‡]Union College, Department of Computer Science, Schenectady, NY 12308. postow@acm.org.

[1]The notation "c.e." means "computably enumerable" and is used in this paper to denote what are otherwise called r.e. (recursively enumerable) sets. Our notation follows a suggestion of Soare [27].

Higman did not state or prove his theorem in these terms; nonetheless, we refer to it as Higman's theorem.

Could we assign a student in automata theory the following problems?

1. Show that if A is decidable then SUBSEQ(A) is decidable.

2. Show that if A is in P then SUBSEQ(A) is in P.

We will show that these are hard problems. Higman's original proof was noneffective. That is, you cannot use the proof to tell you how to write a program that would, given (say) the index for a Turing Machine recognizing (or even deciding) A, output the index of a DFA for SUBSEQ(A). But the question arises as to whether an effective proof is possible. We show that it is not. More precisely, we show that there is no computable function that will take an index for a Turing machine (polynomial time Turing Machine) deciding A and output even a Turing Machine deciding SUBSEQ(A) (polynomial time Turing Machine for SUBSEQ(A)).

What if we give a stronger hypothesis on A and will settle for weaker information about SUBSEQ(A):

1. Show that if A is in P then SUBSEQ(A) is decidable.

2. Show that if A is in coNTIME($\log n$) then SUBSEQ(A) is decidable.

We will show that these are hard problems in that the proofs that they are true cannot be made effective.

How weak can we make the hypothesis on A and still have that the proof that SUBSEQ(A) is decidable is non-effective? When can the proof be effective? In this paper we will prove the following (terms defined more formally in Section 2):

1. There exists a function computable in $\emptyset''$ that does the following: Given a c.e. index for a language A, outputs a DFA for SUBSEQ(A) (Corollary 3.3).

2. Let F be any function that, given a nondeterministic 1-counter machine for a language B, outputs the index of a total Turing machine deciding SUBSEQ($\overline{B}$) (or equivalently, a c.e. index for $\overline{\text{SUBSEQ}(\overline{B})}$). Let G be any function such that $F(x) \leq G(x)$ for all x. Then $\emptyset'' \leq_{\text{T}} G$ (Theorem 4.5, Corollary 4.6). The same goes for a nondeterministic log-time machine for B.

3. There exists a computable function that does the following: Given a context-free grammar for a language A, outputs a DFA for SUBSEQ(A). This is not our result; however, we present the proof for completeness (Theorem 7.1).

4. There exists a computable function that does the following: Given a c.e. index for a language A, outputs a c.e. index for SUBSEQ(A). (This is trivial but we include the statement of it, though not the proof, for completeness.)

From these results one can determine, for virtually any classes of computing devices $\mathcal{D}_1$ and $\mathcal{D}_2$, whether or not the following function is computable: given a device in $\mathcal{D}_1$ for a language A, output a device in $\mathcal{D}_2$ for SUBSEQ(A). Moreover, these results allow us to determine exactly the minimum Turing degree of such a function. It is usually either $\emptyset$ (i.e., computable) or $\emptyset''$, although we say more about this in Section 8.

There is a sharp contrast between Items 2 and 3, above. The latter says that given a CFG for a language A, one can effectively find a DFA for SUBSEQ(A), but it follows from Item 2 that given a CFG for the *complement* of A, one cannot even find a decision procedure for SUBSEQ(A) effectively (or equivalently by Item 4, a c.e. index for $\overline{\text{SUBSEQ}(A)}$).

We give definitions and notation in Section 2. The first two results enumerated above are proved in Sections 3 and 4, respectively, and a proof of the third is in Section 7. In Section 6 we look at a different model where the Turing machine can make queries to the language A (or perhaps other related languages like A' or A'') and tries to find a representation for SUBSEQ(A) in the limit. This is similar to the Inductive Inference model of machine learning, and we explore in more depth the problem of learning SUBSEQ(A) in a companion paper [10]. The results of Sections 3, 4, and 7 suggest that finding SUBSEQ(A) in various representations is always of complexity either $\emptyset$ or $\emptyset''$. In Section 8 we construct, for every c.e. set X and for every Σ_2 set $X \geq_{\text{T}} \emptyset'$, a class $\mathcal{C}$ of c.e. sets such that finding a DFA or CFG for SUBSEQ(A) given a $\mathcal{C}$-index for A is Turing-equivalent to X. In Section 9 we discuss our results in the context of recursive and reverse mathematics. We discuss open problems in Section 10. In the appendices we give two proofs of Higman's theorem. The first essentially follows Higman's original argument, and the second is original.

This paper was inspired by papers of Hartmanis [17] and Hay [18]. In particular, the questions in Section 4 are similar to questions they asked.

2 Definitions

2.1 Language and Machine Conventions

We fix a finite alphabet Σ.

Definition 2.1 Let $x, y \in \Sigma^*$. We say that x is a *subsequence* of y if $x = x_1 \cdots x_n$ and $y \in \Sigma^* x_1 \Sigma^* x_2 \cdots x_{n-1} \Sigma^* x_n \Sigma^*$. We denote this by $x \preceq y$.

Notation 2.2 If A is a set of strings, then SUBSEQ(A) is the set of subsequences of strings in A.

We define classes of languages by defining the sets of machines that recognize them.

Notation 2.3 Let M be any machine described below (e.g., DFA, PDA, Turing machine).

1. All input strings to M are assumed to be over the input alphabet of M, which is given as part of the description of M. The input alphabets may vary from machine to machine within the same class, so for the machines to be uniformly enumerable, we assume that all the input alphabets are finite subsets of some fixed denumerable set such as $\mathbb{N}$.

2. If x is some input to M, we may write $M(x) = 1$ to mean that M halts and accepts on input x, and we may write $M(x) = 0$ to mean that M halts and rejects on input x.

3. We use M to denote both the machine and the language recognized by it, i.e., the set of inputs accepted by M. The intended meaning will usually be clear from the context. If it is not, we will let $L(M)$ denote the language recognized by M.

Convention 2.4 Let M be a Turing machine.

1. We assume that M has exactly two possible types of halting behaviors: accepting and rejecting.
2. We say that M is a *total* machine if M halts on all inputs.

Notation 2.5

1. $F_1, F_2, \ldots$ is a standard enumeration of DFAs. Let REG $= \{F_1, F_2, \ldots\}$.
2. $G_1, G_2, \ldots$ is a standard enumeration of nondeterministic PDAs. Let CFL $= \{G_1, G_2, \ldots\}$. All PDAs in the paper are assumed to be nondeterministic.
3. $H_1, H_2, \ldots$ is a standard enumeration of nondeterministic real-time one-counter automata (NROCAs); see below. Let NROCA $= \{H_1, H_2, \ldots\}$.
4. $P_1, P_2, \ldots$ is a standard enumeration of 0,1-valued polynomial-time Turing Machines. Let P $= \{P_1, P_2, \ldots\}$.
5. $M_1, M_2, \ldots$ is a standard enumeration of Turing Machines. Let CE $= \{M_1, M_2, \ldots\}$.
6. Let DEC $= \{M_i : M_i$ is a total machine$\}$.

Notation 2.6 For any class $\mathcal{C}$ above, we let co$\mathcal{C}$ be the same as $\mathcal{C}$ except that we take each $\mathcal{C}$-device to recognize the *complement* of what it would normally recognize with respect to $\mathcal{C}$. By *complement* we mean the complement relative to the set of all strings over the device's input alphabet.

Note 2.7 Our definition of CE is slightly different from computability theory, in which the ith c.e. set is usually defined to be the set of inputs on which M_i *halts*. Here, we define the ith c.e. set to be the set of inputs that M_i *accepts*. Thus M_i may halt on (and reject) an input not in the set. We adopt this convention in order to have a uniform language recognition criterion, regardless of whether or not the machines are total. All the usual results of computability theory carry over to this convention with minimal alterations.

2.2 NROCAs

Informally, a *one-counter automaton (OCA)* is a finite automaton that comes equipped with a single counter that holds a nonnegative integer. Initially, the counter's value is zero. At each step, the counter can be incremented, decremented (if positive), or left unchanged. The only aspect of the counter that the automaton can use in its transition function is whether or not it is zero. Deterministic and nondeterministic versions of OCAs are defined in the usual way.

There are several variants of this model. A particularly weak variant of interest to us is that of real-time one-counter automata (ROCAs). A ROCA must read its input from left to right, advancing on each step (i.e., no λ-moves allowed), and either accept or reject at the end. A nondeterministic ROCA (NROCA) is essentially equivalent to a PDA with no λ-moves, a unary

stack alphabet (except for a stack-empty marker), and at most one symbol pushed or popped at a time.

Counter machines have been studied by many people. Two-counter machines were shown to be universal by Minsky [21] (improved by Fischer [11]). Deterministic one-counter automata (DOCAs) were first defined and their properties studied by Valiant & Paterson [28]. Real-time counter machines were studied by Fischer, Meyer, & Rosenberg [12]. ROCAs were introduced in the context of machine learning by Fahmy & Roos [9].

2.3 Rectangular Traces

The concepts in this section will be used for the proofs of Lemma 4.4 and Theorem 4.5.

We'll use a standard notion of a Turing machine with a single one-way infinite tape. See Sipser [25] for details. We do *not* need to assume that $M_1, M_2, M_3, \ldots$ all share the same input alphabet, but we *will* assume WLOG that 0 belongs to the input alphabets of all the M_e.

Fix a deterministic Turing machine M with state set Q, input alphabet Σ, and tape alphabet Γ disjoint from Q. We represent a *configuration* of M in the usual way by a string of the form $C = \ell q r \in (Q \cup \Gamma)^*$, where $q \in Q$ and $\ell, r \in \Gamma^*$ with $|r| > 0$. C represents the configuration where M is in state q, ℓ is the entire contents of the tape to the left of the head, and r is the contents of the tape starting at the head and extending at least to the rightmost nonblank symbol or possibly beyond. Note that the same configuration is represented by infinitely many strings by padding r to the right with blanks. A string C representing a configuration is *minimal* if no shorter string represents the same configuration; equivalently, C is minimal iff either its last symbol is nonblank or its penultimate symbol is in Q. Let $x \in \Sigma^*$ be a string. A *rectangular trace* of M on input x is a string of the form

$$\#C_0\#C_1\#\cdots\#C_k\#,$$

where

- $k \geq 0$,
- $\#$ is a symbol not in $Q \cup \Gamma$,
- C_0 represents the initial configuration of M on input x (that is, $C_0 = q_0 x$, possibly padded on the right with blank symbols, where q_0 is the start state of M),
- for $0 \leq i < k$, C_{i+1} represents the successor to the configuration represented by C_i, according to the transition function of M, and
- $|C_0| = |C_1| = \cdots = |C_k| = n$ for some n (n must be large enough so that this is possible).

We say that k and n are the *depth* and *width* of the trace, respectively, and that $\#$ is the *separator* for the trace. If C_k represents a halting configuration of M, then we say that the trace is *accepting* or *rejecting* according to the type of halting configuration; otherwise, we say that the trace is *incomplete*. The trace has *minimum width* if at least one of the C_i is minimal.

Note 2.8 Let M be a machine, x an input string, and $\#$ some appropriate separator symbol. The minimum-width accepting rectangular trace of M on x with separator $\#$ is unique if it exists, and there exists such a trace iff M accepts x.

2.4 Function Classes and Computability

Definition 2.9 Let $\mathcal{C} = \{C_1, C_2, \ldots\}$ and $\mathcal{D} = \{D_1, D_2, \ldots\}$ be sets of machines with known interpretations. Then $\mathcal{F}^{\mathcal{C},\mathcal{D}}$ is the class of all functions that take an index i for a machine in $\mathcal{C}$ and produce an index j for a machine in $\mathcal{D}$ such that $L(D_j) = \text{SUBSEQ}(L(C_i))$.

We use the notion of Muchnik reducibility (also known as weak reducibility) [22] to measure the complexity of the various $\mathcal{F}^{\mathcal{C},\mathcal{D}}$.

Definition 2.10 (after Muchnik [22]) Let $\mathcal{F}$ and $\mathcal{G}$ be classes of functions $\mathbb{N} \to \mathbb{N}$. We say that $\mathcal{F}$ *Muchnik reduces to* $\mathcal{G}$ (denoted $\mathcal{F} \leq_w \mathcal{G}$)[2] if $(\forall g \in \mathcal{G})(\exists f \in \mathcal{F})[\, f \leq_{\text{T}} g\,]$. $\mathcal{F}$ and $\mathcal{G}$ are *Muchnik equivalent* ($\mathcal{F} \equiv_w \mathcal{G}$) if $\mathcal{F} \leq_w \mathcal{G}$ and $\mathcal{G} \leq_w \mathcal{F}$. Equivalence classes under $\equiv_w$ are *Muchnik degrees*.

The following note is crucial for the definition of $\mathcal{F}^{\mathcal{C},\mathcal{D}}$.

Note 2.11

1. If $A \subseteq \mathbb{N}$ is a set, we informally write $\mathcal{F}^{\mathcal{C},\mathcal{D}} \leq_w A$ to mean $\mathcal{F}^{\mathcal{C},\mathcal{D}} \leq_w \{\chi_A\}$, where $\chi_A : \mathbb{N} \to \{0,1\}$ denotes the characteristic function of A. This means that there is some function $f \in \mathcal{F}^{\mathcal{C},\mathcal{D}}$ such that $f \leq_{\text{T}} A$. Likewise, we write $A \leq_w \mathcal{F}^{\mathcal{C},\mathcal{D}}$ to mean $\{\chi_A\} \leq_w \mathcal{F}^{\mathcal{C},\mathcal{D}}$. This means that, for any $f \in \mathcal{F}^{\mathcal{C},\mathcal{D}}$, $A \leq_{\text{T}} f$. We also write $\mathcal{F}^{\mathcal{C},\mathcal{D}} \equiv_w A$ to mean that both of the above statements hold.

2. "$\mathcal{F}^{\mathcal{C},\mathcal{D}}$ is computable" means that $\mathcal{F}^{\mathcal{C},\mathcal{D}} \leq_w \emptyset$ in the sense of the previous item, i.e., $\mathcal{F}^{\mathcal{C},\mathcal{D}}$ contains a computable function.

3. Muchnik reducibility is defined in terms of Turing reducibility. For other reductions r we can define $A \leq_{\text{r}} \mathcal{F}^{\mathcal{C},\mathcal{D}}$, etc. in analogy with Muchnik reducibility by replacing $\leq_{\text{T}}$ with $\leq_{\text{r}}$ in the definition.

Notation 2.12 For $\mathcal{C}$ as above, we let $\mathcal{F}^{\mathcal{C},\text{DEC}}$ be the subclass of $\mathcal{F}^{\mathcal{C},\text{CE}}$ consisting of the functions $f \in \mathcal{F}^{\mathcal{C},\text{CE}}$ that always output indices of total machines. (We will never use DEC as the first superscript in $\mathcal{F}^{\mathcal{C},\mathcal{D}}$.)

The following theorem is an exercise we leave to the reader.

Theorem 2.13

1. *$\mathcal{F}^{\text{CE,CE}}$ is computable.*
2. *$\mathcal{F}^{\text{REG,REG}}$ is computable.*

Definition 2.14 If f and g are functions $\mathbb{N} \to \mathbb{N}$, then g *bounds* f if $(\forall e)[f(e) \leq g(e)]$. If $\mathcal{F}$ is a class of functions, then we say g *bounds* $\mathcal{F}$ to mean that g bounds some function in $\mathcal{F}$. A function f is *computably bounded* if there is a computable g that bounds f. Likewise, a function class $\mathcal{F}$ is *computably bounded* if there is a computable g that bounds $\mathcal{F}$.

[2] This should not be confused with weak truth-table reducibility (*wtt*-reducibility), which is sometimes also denoted by $\leq_w$.

Note that this definition is nonstandard. For example, a class of functions is typically regarded as computably bounded if there is a single computable function g that bounds every function in the class. Here, g only needs to bound at least one function in the class.

Definition 2.15 If $\mathcal{C}, \mathcal{D}$ are sets of machines, we use the notation $\mathcal{C} \sqsubseteq \mathcal{D}$ to mean that there exists a computable function h such that $L(C_i) = L(D_{h(i)})$ for every i.

It is well-known that REG $\sqsubseteq$ NROCA $\sqsubseteq$ CFL $\sqsubseteq$ P $\sqsubseteq$ DEC $\sqsubseteq$ CE and that REG $\sqsubseteq$ coNROCA $\sqsubseteq$ coCFL $\sqsubseteq$ P $\sqsubseteq$ DEC $\sqsubseteq$ coCE.

Notation 2.16 If f and g are functions $\mathbb{N} \to \mathbb{N}$, we say that $f \leq_{\text{1-tt}} g$ if $f \leq_{\text{T}} g$ via a machine that halts on every input and every oracle, making at most one oracle query.

The following lemma is obvious.

Lemma 2.17 *Let $\mathcal{C}_1, \mathcal{C}_2, \mathcal{D}_1, \mathcal{D}_2, \mathcal{C}, \mathcal{D}$ be sets of machines.*

1. *If $\mathcal{C}_1 \sqsubseteq \mathcal{C}_2$, then $\mathcal{F}^{\mathcal{C}_1,\mathcal{D}} \leq_{\text{1-tt}} \mathcal{F}^{\mathcal{C}_2,\mathcal{D}}$.*
2. *If $\mathcal{C}_1 \sqsubseteq \mathcal{C}_2$, then for every function g bounding $\mathcal{F}^{\mathcal{C}_2,\mathcal{D}}$ there is a function $f \leq_{\text{1-tt}} g$ bounding $\mathcal{F}^{\mathcal{C}_1,\mathcal{D}}$. In particular, if $\mathcal{F}^{\mathcal{C}_2,\mathcal{D}}$ is computably bounded, then $\mathcal{F}^{\mathcal{C}_1,\mathcal{D}}$ is computably bounded.*
3. *If $\mathcal{D}_1 \sqsubseteq \mathcal{D}_2$, then $\mathcal{F}^{\mathcal{C},\mathcal{D}_2} \leq_{\text{1-tt}} \mathcal{F}^{\mathcal{C},\mathcal{D}_1}$.*
4. *If $\mathcal{D}_1 \sqsubseteq \mathcal{D}_2$, then for every function g bounding $\mathcal{F}^{\mathcal{C},\mathcal{D}_1}$ there is a function $f \leq_{\text{T}} g$ bounding $\mathcal{F}^{\mathcal{C},\mathcal{D}_2}$. In particular, if $\mathcal{F}^{\mathcal{C},\mathcal{D}_1}$ is computably bounded, then $\mathcal{F}^{\mathcal{C},\mathcal{D}_2}$ is computably bounded.*

Corollary 2.18

1. *If $\mathcal{C} \in \{\text{REG}, \text{NROCA}, \text{coNROCA}, \text{CFL}, \text{coCFL}, \text{P}, \text{CE}\}$, then $\mathcal{F}^{\mathcal{C},\text{CE}}$ is computable.*
2. *If $\mathcal{D} \in \{\text{REG}, \text{NROCA}, \text{coNROCA}, \text{CFL}, \text{coCFL}, \text{P}, \text{DEC}, \text{CE}\}$, then $\mathcal{F}^{\text{REG},\mathcal{D}}$ is computable.*

The notation below is standard. For the notation that relates to computability theory, our reference is [26].

Notation 2.19 Let $e, s \in \mathbb{N}$ and $x \in \Sigma^*$.

1. The empty string is denoted by λ.
2. $M_{e,s}(x)$ is the result of running M_e on x for s steps.
3. $W_e = L(M_e) = \{x : M_e(x) = 1\}$. Note that $\{W_1, W_2, \ldots\} = \text{CE}$ is the set of all c.e. sets.
4. $W_{e,s} = \{x : |x| \leq s \wedge M_{e,s}(x) = 1\}$.
5. $\Sigma_0, \Pi_0, \Delta_0, \Sigma_1, \Pi_1, \Delta_1, \Sigma_2, \Pi_2, \Delta_2, \ldots$ are defined in the usual way in the context of computability theory.
6. Let $M_1^{()}, M_2^{()}, \ldots$ be a standard list of all oracle Turing machines. $A' = \{e : M_e^A(e) = 1\}$. This is pronounced, "$A$ jump."

7. $\emptyset$ is the empty set.
8. $\emptyset'$ is defined using the jump operator and is actually equivalent to the halting set.
9. $\emptyset''$ is defined using the jump operator and is known to be Σ_2-complete.
10. EMPTY is $\{e : W_e = \emptyset\}$. EMPTY is known to be Π_1-complete.
11. FIN is $\{e : W_e \text{ is finite}\}$. FIN is known to be Σ_2-complete.
12. If A and B are sets, then $A \oplus B = \{2x : x \in A\} \cup \{2x+1 : x \in B\}$. Note that $A \leq_{\mathrm{m}} A \oplus B$ and $B \leq_{\mathrm{m}} A \oplus B$. One can generalize the $\oplus$ operator to functions as well.

Definition 2.20 Let $\mathcal{C} = \{C_1, C_2, \ldots\}$ be a set of devices.

1. EMPTY$_{\mathcal{C}}$ is $\{e : L(C_e) = \emptyset\}$.
2. FIN$_{\mathcal{C}}$ is $\{e : L(C_e) \text{ is finite}\}$.

3 A General Upper Bound on the Muchnik Degree of $\mathcal{F}^{\mathcal{C},\mathcal{D}}$

Throughout this section $\mathcal{C} = \{C_1, C_2, \ldots\}$ and $\mathcal{D} = \{D_1, D_2, \ldots\}$ are sets of devices, and REG $\sqsubseteq \mathcal{D}$.

Theorem 3.1 *If*

$$\{(e,x) : C_e(x) = 1\} \in \Sigma_1$$

and

$$\{(i,y) : D_i(y) = 1\} \in \Delta_2$$

then $\mathcal{F}^{\mathcal{C},\mathcal{D}} \leq_w \emptyset''$.

Proof: To obtain $\mathcal{F}^{\mathcal{C},\mathcal{D}} \leq_w \emptyset''$ we construct the following sentence parameterized by e and i. The sentence is asking if D_i decides SUBSEQ(C_e). The sentence, which we denote $\alpha(e,i)$, is

$$(\forall x)\Big[D_i(x) = 1 \iff (\exists y)[x \preceq y \wedge C_e(y) = 1]\Big].$$

Since $\{(i,x) | D_i(x) = 1\} \in \Delta_2$ and $\{(e,y) : C_e(y) = 1\} \in \Sigma_1$ the sentence $\alpha(e,i)$ can be written as a Π_2 statement, and hence its truth is computable in $\emptyset''$.

We can compute an $f \in \mathcal{F}^{\mathcal{C},\mathcal{D}}$ as follows. Given e, ask $\alpha(e,0)$, $\alpha(e,1)$, ... until you get an answer of YES on some $\alpha(e,i)$, then output i. Each question is computable in $\emptyset''$. The algorithm must halt since *some* D_i works. ∎

Note 3.2 The algorithm in Theorem 3.1 is a Turing reduction to $\emptyset''$. We will see in Corollary 4.9 that in many cases the algorithm cannot be improved to be a truth-table reduction. This will hold no matter what oracle is used.

Corollary 3.3 $\mathcal{F}^{\mathrm{CE,REG}} \leq_w \emptyset''$.

Corollary 3.4 *Assume*

1. $\mathcal{C} \in \{\text{REG}, \text{NROCA}, \text{coNROCA}, \text{CFL}, \text{coCFL}, \text{P}, \text{CE}\}$, *and*
2. $\mathcal{D} \in \{\text{REG}, \text{NROCA}, \text{coNROCA}, \text{CFL}, \text{coCFL}, \text{P}\}$.

Then $\mathcal{F}^{\mathcal{C},\mathcal{D}} \leq_w \emptyset''$.

Proof: By Corollary 3.3 and Lemma 2.17. ▮

Corollary 3.5 *If* $\mathcal{C} \in \{\text{REG}, \text{NROCA}, \text{coNROCA}, \text{CFL}, \text{coCFL}, \text{P}, \text{CE}\}$, *then* $\mathcal{F}^{\mathcal{C},\text{DEC}} \leq_w \emptyset''$.

Proof: We have $\mathcal{F}^{\mathcal{C},\text{REG}} \leq_w \emptyset''$ by Corollary 3.4. Since REG $\sqsubseteq$ DEC, we have $\mathcal{F}^{\mathcal{C},\text{DEC}} \leq_{\text{1-tt}} \mathcal{F}^{\mathcal{C},\text{REG}}$ by (3) of Lemma 2.17. ▮

4 The Growth Rate of $\mathcal{F}^{\mathcal{C},\mathcal{D}}$: A Lower Bound

In Section 3 we gave a general upper bound of $\emptyset''$ for the Muchnik degree of $\mathcal{F}^{\mathcal{C},\mathcal{D}}$ for many $\mathcal{C}$ and $\mathcal{D}$. In this section, we show that this bound is tight by showing that $\emptyset''$ is computable in any function *bounding* some member of $\mathcal{F}^{\mathcal{C},\mathcal{D}}$. It follows that for many $\mathcal{C}$ and $\mathcal{D}$, $\mathcal{F}^{\mathcal{C},\mathcal{D}}$ is not computably bounded, i.e., no function in $\mathcal{F}^{\mathcal{C},\mathcal{D}}$ is computably bounded.

The minimum complexity of $\mathcal{F}^{\mathcal{C},\mathcal{D}}$ is easy to determine in some special cases. For example, note that for any c.e. language A: (i) A is finite iff SUBSEQ(A) is finite; (ii) given a DFA for SUBSEQ(A), one can tell effectively whether or not SUBSEQ(A) is finite; (iii) it requires $\emptyset''$ to decide, given a c.e. index for A, whether or not A is finite. Thus any function that translates a c.e. index for A into a DFA for SUBSEQ(A) can be used to solve the finiteness problem for c.e. sets and hence compute $\emptyset''$. It remains $\emptyset''$-hard, however, to find other types of devices for SUBSEQ(A), even though the finiteness problem for these other devices is undecidable (see Corollary 4.10).

Throughout this section $\mathcal{C} = \{C_1, C_2, \ldots\}$ and $\mathcal{D} = \{D_1, D_2, \ldots\}$. In addition we assume that there is a notion of running $D_i(x)$ for s steps, which we denote $D_{i,s}(x)$, and that REG $\sqsubseteq \mathcal{D}$.

Lemma 4.1 *There is a computable function F that takes as input an index e for a c.e. set W_e and a bound m on indices for machines from $\mathcal{D}$, and outputs the index of a Turing machine M that behaves as follows: If at least one of $D_1, \ldots, D_m$ decides W_e (i.e., at least one of $D_1, \ldots, D_m$ recognizes W_e and is total), then M is total and decides a finite variant of W_e.*

Proof: On input (e, m) the function F outputs a machine M that implements the following:

ALGORITHM

1. Input x. Let $n = |x|$.
2. Find $W_{e,n}$.
3. Run $D_{i,n}(z)$ for $1 \leq i \leq m$ and $0 \leq |z| \leq n$. For each i:
   (a) If there is a $z \in W_{e,n}$ such that $D_{i,n}(z) = 0$, then declare D_i to be invalid. Note that in this case, $|z| \leq n$ by definition, and we absolutely know that D_i does not decide W_e.

(b) If there is a $z \notin W_{e,n}$ and some i with $1 \leq i \leq n$ such that $D_{i,n}(z) = 1$ then declare D_i to be invalid. Note that in this case we *do not know for sure* that D_i does not decide W_e. It may be the case that $(\exists t > n)[z \in W_{e,t}]$. Hence, although we declare D_i invalid for now, we may declare it valid later in the computation of $F(x)$.

4. Dovetail the following computations.

   (a) Run $D_i(x)$ for all valid D_i. If any of them halt then output the first answer given.

   (b) Enumerate W_e looking for $z \in W_e$ such that some machine D_i was declared invalid because $D_{i,n}(z) = 1 \wedge z \notin W_{e,n}$. If such a z is found (and there are no other reasons for D_i to be invalid), then

      i. Declare D_i valid and have the dovetail procedure of Step 4a use it.

      ii. Keep looking for more z and hence more machines to declare valid.

END OF ALGORITHM

For any e and m, assume that at least one of $D_1, \ldots, D_m$ decides W_e. By renumbering we assume that D_1 is one of the machines that decides W_e.

We first show that M always halts. Let x be an input and $n = |x|$. There are two cases.

- D_1 is not declared invalid. Then since $D_1(x) \downarrow$ we find some D_i halting on x in Step 4a.

- D_1 is declared invalid via z. Since D_1 decides W_e it must be the case that $z \in W_e - W_{e,n} \wedge D_{1,n}(z) = 1$. Because this can happen for only a fixed finite set of possible z's (recall that $|z| \leq n$), it must be that if the computation in Step 4b goes on long enough then all such $z \in W_e$ will be observed and D_1 will be declared valid. Since D_1 is total, the computation of $M(x)$ will halt when $D_1(x)$ halts, if it has not halted already.

We now show that M decides a finite variant of W_e.

Fix i, $1 \leq i \leq m$. We show that if $(\exists z)[D_i(z) = 1 \wedge z \notin W_e]$ or $(\exists z)[D_i(z) = 0 \wedge z \in W_e]$, then there exists n_i such that for all x, $|x| \geq n_i$, machine D_i will be declared invalid and stay invalid during the computation of $M(x)$. There are two cases.

- There exists $z \in W_e$ such that $D_i(z) = 0$. Let n' be least such that $D_{i,n'}(z) = 0$, and let n'' be least such that $z \in W_{e,n''}$. For all x, $|x| \geq \max\{|z|, n', n''\}$, D_i will be declared invalid in Step 3a and stay invalid. Let $n_i = \max\{|z|, n', n''\}$.

- There exists $z \notin W_e$ such that $D_i(z) = 1$. Let n' be least such that $D_{i,n'}(z) = 1$. For all x, $|x| \geq \max\{|z|, n'\}$ D_i will be declared invalid in Step 3b and will stay invalid. Let $n_i = \max\{|z|, n'\}$.

Let n_I be the max of the n_i as i runs through all of the machines that converge to a wrong answer. Let x be such that $|x| \geq n_I$. The valid machines used by $M(x)$ will either converge on x and be correct or not converge. Since at least one of the machines decides W_e, $M(x)$ will be correct. ∎

Theorem 4.2 *Assume that $\mathcal{F}^{\mathcal{C},\mathrm{CE}}$ is computable, i.e., there is a computable function h such that* $\mathrm{SUBSEQ}(C_i) = W_{h(i)}$. *Let $f \in \mathcal{F}^{\mathcal{C},\mathcal{D}}$ be a function that only outputs indices for total devices, and let g be any function bounding f. Then* $\mathrm{FIN}_{\mathcal{C}} \leq_{\mathrm{T}} g \oplus \emptyset'$.

Proof: For all e, one of $D_1, D_2, \ldots, D_{g(e)}$ (namely, $D_{f(e)}$) is a total device which decides $\mathrm{SUBSEQ}(C_e)$. By the premise we know $\mathrm{SUBSEQ}(C_e) = W_{h(e)}$. By Lemma 4.1 we can obtain from e and $g(e)$ a total Turing machine M that recognizes a finite variant of $W_{h(e)}$. Let $A = L(M)$.

Note that:

- $e \in \mathrm{FIN}_{\mathcal{C}} \Rightarrow L(C_e)$ finite $\Rightarrow \mathrm{SUBSEQ}(L(C_e))$ finite $\Rightarrow W_{h(e)}$ finite $\Rightarrow A$ finite $\Rightarrow (\forall^\infty n)[A \cap \Sigma^n = \emptyset]$.
- $e \notin \mathrm{FIN}_{\mathcal{C}} \Rightarrow L(C_e)$ infinite $\Rightarrow (\forall n)[\mathrm{SUBSEQ}(L(C_e)) \cap \Sigma^n \neq \emptyset] \Rightarrow (\forall n)[W_{h(e)} \cap \Sigma^n \neq \emptyset] \Rightarrow (\forall^\infty n)[A \cap \Sigma^n \neq \emptyset]$.

Recalling that A is decidable (uniformly in e and $g(e)$), we can determine which of these two holds of A by asking queries to $\emptyset'$. Keep asking:

- $(\forall n \geq 0)[A \cap \{0,1\}^n = \emptyset]$
- $(\forall n \geq 0)[A \cap \{0,1\}^n \neq \emptyset]$
- $(\forall n \geq 1)[A \cap \{0,1\}^n = \emptyset]$
- $(\forall n \geq 1)[A \cap \{0,1\}^n \neq \emptyset]$
- $(\forall n \geq 2)[A \cap \{0,1\}^n = \emptyset]$
- $(\forall n \geq 2)[A \cap \{0,1\}^n \neq \emptyset]$
- etc.

until you get a YES. If the YES answer was to a query of the form, "$\ldots = \emptyset$," then $e \in \mathrm{FIN}_{\mathcal{C}}$. If the YES answer was to a query of the form, "$\ldots \neq \emptyset$," then $e \notin \mathrm{FIN}_{\mathcal{C}}$.

Thus, we have $\mathrm{FIN}_{\mathcal{C}} \leq_{\mathrm{T}} g \oplus \emptyset'$ as stated. ∎

Lemma 4.3 *Let $K_1 \subseteq K_2 \subseteq \cdots$ be any computable enumeration of $\emptyset'$. Suppose that there is a computable function h such that for all e and s,*

$$\text{if } e \in \emptyset' - K_s \text{ and } D_s \text{ is total, then } L(D_s) \neq \mathrm{SUBSEQ}(L(C_{h(e)})).$$

Then for any function $f \in \mathcal{F}^{\mathcal{C},\mathcal{D}}$ outputting indices for total devices and for any function g bounding f, we have $\emptyset' \leq_{\mathrm{T}} g$.

Proof: Let f, g, and h be as above. For any e, let $m = g(h(e))$. Then $\mathrm{SUBSEQ}(L(C_{h(e)})) = L(D_s)$ for some total device D_s where $s = f(h(e))$ and $1 \leq s \leq m$. Then by assumption, $e \notin \emptyset' - K_s$, and so either $e \in K_s$ or $e \notin \emptyset'$.

The following g-computable algorithm thus decides whether $e \in \emptyset'$:

1. Compute $m = g(h(e))$.

2. If $e \in K_m$ then output YES, else output NO.

∎

The next lemma, used to prove Theorem 4.5 below, relates to nondeterministic real-time one-counter automata (NROCAs—see Section 2.2). The proof technique (also used in the proof of Theorem 4.5) is a routine adaptation of a standard technique of Hartmanis [16], who showed that the set of invalid Turing machine computations is context-free, yielding the undecidability of $\text{EMPTY}_{\text{coCFL}}$. Lemma 4.4 improves this to one-counter machines. Although a result of this sort may easily have been proved decades ago, we are unable to find a reference, and so we will consider it folklore.

To avoid confusion, we will always let $L(H)$ denote the language recognized by an NROCA H in the standard way, i.e., with respect to NROCA rather than coNROCA.

Lemma 4.4 *$\emptyset' \leq_{\text{T}} \text{EMPTY}_{\text{coNROCA}}$ and $\emptyset'' \leq_{\text{T}} \text{FIN}_{\text{coNROCA}}$.*

Proof: Given an input e, we effectively construct an NROCA $H_{g(e)}$ such that $|L(M_e)| = |\overline{L(H_{g(e)})}|$. This suffices, as it gives us reductions $\text{EMPTY} \leq_{\text{m}} \text{EMPTY}_{\text{coNROCA}}$ and $\text{FIN} \leq_{\text{m}} \text{FIN}_{\text{coNROCA}}$, both via g.

Fix a symbol # not in the state set or input alphabet of any M_e. Given any e, let Y_e be the language of all minimum-width accepting rectangular traces of M_e (on any inputs) with separator # (see Section 2.3). By Note 2.8, $|Y_e| = |L(M_e)|$. Given e, we will effectively construct an NROCA $H_{g(e)}$ such that $Y_e = \overline{L(H_{g(e)})}$, which suffices for the lemma.

The input alphabet of $H_{g(e)}$ is $\Sigma_e = Q_e \cup \Gamma_e \cup \{\#\}$, where Q_e and Γ_e are the state set and tape alphabet of M_e, respectively. Given input string $w \in \Sigma_e^*$, the NROCA $H_{g(e)}$ first branches nondeterministically into seven branches, each branch checking one of the conditions below and accepting iff the condition is *violated*:

1. $w = \#C_0\#C_1\#\cdots\#C_k\#$ for some $k \geq 0$ and some strings $C_0, \ldots, C_k \in (Q_e \cup \Gamma_e)^*$. (This condition is regular, so the counter is not needed for this branch.)

2. $C_0, \ldots, C_k$ all represent configurations of M_e. (This condition is also regular.)

3. $|C_0| = |C_1| = \cdots = |C_k|$. (The branch nondeterministcally chooses two adjacent C_i and C_{i+1}, increments the counter while reading C_i, decrements it while reading C_{i+1}, then accepts iff $|C_i| \neq |C_{i+1}|$.

4. C_0 represents the initial configuration of M_e on some input. (This condition is regular.)

5. C_k represents some accepting configuration of M_e. (This condition is regular.)

6. For all $0 \leq i < k$, C_{i+1} represents the successor of C_i according to M_e's transition function. (On this branch, $H_{g(e)}$ first nondeterministically chooses some C_i. It then starts incrementing the counter while reading C_i up to some nondeterministically chosen position j, where it records in its state the $(j-1)$st, jth, $(j+1)$st, and $(j+2)$nd symbols of C_i (if they exist). This gives it enough information to determine what the jth symbol of C_{i+1} should be, based on the transition function of M_e, which is hard-coded into $H_{g(e)}$. Upon reaching C_{i+1} it decrements the counter to find the jth symbol of C_{i+1}, then accepts iff the symbol is wrong. (If it is discovered that C_i is the last configuration, then this branch rejects.))

7. w has minimum width. (This condition is regular.)

The string w is in Y_e iff all these conditions hold. Furthermore, the description of $H_{g(e)}$ above shows that it accepts w iff at least one of these conditions does not hold. Thus $Y_e = \overline{L(H_{g(e)})}$. ∎

Theorem 4.5 *$\emptyset''$ is computable in any function bounding $\mathcal{F}^{\text{coNROCA,DEC}}$.*

Proof: Let g be any function bounding $\mathcal{F}^{\text{coNROCA,DEC}}$. (Recall that a function f in the class $\mathcal{F}^{\text{coNROCA,DEC}}$ takes an NROCA recognizing a language A and outputs a total Turing machine deciding SUBSEQ($\overline{A}$).) We will show that $\mathcal{C} = \text{coNROCA}$ and $\mathcal{D} = \text{CE}$ satisfy the hypotheses of Lemma 4.3 (for a particular computable enumeration of $\emptyset'$), whence $\emptyset' \leq_{\text{T}} g$. Combining this with Theorem 4.2 and Lemma 4.4, we get

$$\emptyset'' \leq_{\text{T}} \text{FIN}_{\text{coNROCA}} \leq_{\text{T}} g \oplus \emptyset' \equiv_{\text{T}} g,$$

which proves the theorem.

Fix a Turing machine U such that $L(U) = \emptyset'$. We can assume that U never rejects any input, i.e., U either accepts or runs forever. We may also assume that for all e, $U(e)$ runs for at least e steps. Define a computable enumeration $K_1 \subseteq K_2 \subseteq \cdots$ of $\emptyset'$ by letting

$$K_s = \{e : U \text{ accepts } e \text{ in at most } s \text{ steps}\}.$$

We can fix another Turing machine V such that, for every $i > 0$ and $m \geq 0$, $V(0^i 10^m)$ simulates $M_i(0^m)$, i.e., V accepts $0^i 10^m$ if and only if M_i accepts 0^m, and V rejects $0^i 10^m$ if and only if M_i rejects 0^m.

Fix three distinct symbols $\#, \$, \%$ that are not in the tape alphabets of either U or V. For each e we define Z_e to be the language of all strings of the form

$$\$^n T \% T_0 \% T_1 \% \cdots \% T_\ell,$$

where

1. $n > 0$ and $\ell \geq 0$,
2. T is an incomplete rectangular trace of U on input e with separator $\#$ and width n (let s be the depth of T),
3. T_ℓ is a rejecting rectangular trace of V on input $0^s 10^\ell$ with separator $\#$ and width n,
4. for all $0 \leq j < \ell$, T_j is an accepting rectangular trace of V on input $0^s 10^j$ with separator $\#$ and width n, and
5. n is such that at least one of the traces $T, T_0, \ldots, T_\ell$ has minimum width.

We have $Z_e \subseteq \Sigma^*$, where $\Sigma = Q_U \cup \Gamma_U \cup Q_V \cup \Gamma_V \cup \{\#, \$, \%\}$. Here, U has state set Q_U and tape alphabet Γ_U, and V has state set Q_V and tape alphabet Γ_V. We can assume as before that $Q_U \cap \Gamma_U = Q_V \cap \Gamma_V = \emptyset$.

Given e we will effectively construct an NROCA $H_{h(e)}$ such that $Z_e = \overline{L(H_{h(e)})}$. But first, to see that this h satisfies the hypotheses of Lemma 4.3, note the following:

1. If $e \in \emptyset'$, then Z_e is finite.

   (If k is least such that $e \in K_k$, then all incomplete rectangular traces of U on input e have depth less than k. For each possible $s < k$, if $\$^n T\%T_0\%T_1\%\cdots\%T_\ell$ is a string in Z_e and T has depth s, then clearly, n, ℓ, and $T_0, \ldots, T_\ell$ are all uniquely determined—partially owing to the minimum-width condition, which is essential here. (For example, ℓ must be least such that V rejects 0^s10^ℓ, i.e., $0^\ell \notin L(M_s)$.) Hence for each $s < k$ there is at most one such string. So we have $|Z_e| \leq k$.)

2. If s is such that $e \notin K_s$, M_s is a total machine, and $L(M_s)$ is finite, then there exist strings x and y such that

   (a) $x \preceq y$,
   (b) $x \notin L(M_s)$, and
   (c) $y \in Z_e$.

   (Since $L(M_s)$ is finite and M_s is total, there is an $\ell \geq 0$ such that M_s rejects 0^ℓ but accepts 0^j for all $0 \leq j < \ell$. Equivalently, V rejects 0^s10^ℓ and accepts 0^s10^j for all $j < \ell$. Thus there is a string $y = \$^n T\%T_0\%T_1\%\cdots\%T_\ell \in Z_e$ where T has depth s. Setting $x = 0^\ell$, we see that (2b) and (2c) are satisfied. For (2a), note that 0^ℓ appears in y as part of the initial configuration in T_ℓ, and so $x \preceq y$.)

Suppose $e \in \emptyset' - K_s$ and M_s is total. Then since Z_e is finite, so is SUBSEQ(Z_e). If $L(M_s)$ is infinite, then obviously $L(M_s) \neq \text{SUBSEQ}(Z_e)$. If $L(M_s)$ is finite, then by the second item above, there exist $x \preceq y$ with $x \notin L(M_s)$ but $y \in Z_e$, making $x \in \text{SUBSEQ}(Z_e) - L(M_s)$. In either case, $L(M_s) \neq \text{SUBSEQ}(Z_e)$. The machine $H_{h(e)}$ we construct below recognizes $\overline{Z_e}$, and thus $\text{SUBSEQ}(Z_e) = \text{SUBSEQ}(\overline{L(H_{h(e)})})$, which establishes the hypothesis of Lemma 4.3.

It remains to describe $H_{h(e)}$. The construction of $H_{h(e)}$ is straightforward and mirrors that of $H_{g(e)}$ previously in Lemma 4.4. $H_{h(e)}$ first splits into a number of branches, each branch checking some condition of the input string and accepting iff the condition is violated. The previous discussion suggests how different branches of an NROCA computation can check that T is a rectangular trace of U and that each T_j is a rectangular trace of V, all with separator #. We can check that e is the input string for T since we have e hard-coded into $H_{h(e)}$. Checking that each trace has the required type (accepting, rejecting, or incomplete) is also straightforward. The widths can be checked easily enough by first counting the number of \$'s then nondeterministically choosing a configuration string and verifying that it has the same number of symbols. Minimality of the width of at least one of the traces can also be easily checked by finding some minimal configuration string.

The only remaining check is that each T_j is a trace of V on input 0^s10^j, where s is the depth of T. We can store s in the counter by counting the number of # symbols in T except the first and last ones. Then we nondeterministically choose some j with $0 \leq j \leq \ell$ and check (by decrementing the counter) that the first configuration of T_j corresponds to an input string of the form 0^s10^t for some $t \geq 0$. Finally, we need to check that $t = j$ in each case. On a separate branch we nondeterministically choose some T_j with $0 \leq j \leq \ell$, counting the number of %'s preceding it ($j+1$ of them). We then use the counter to check that there are exactly j many 0's to the right of the 1 on T_j's input string.

This concludes the informal description of $H_{h(e)}$. Proving that $H_{h(e)}$ has all the requisite properties is routine. ∎

Remark. NROCAs can be simulated in nondeterministic 1-way log space. Further, the nondeterminism in each $H_{h(e)}$ above is limited so that it can be simulated in DSPACE($\log n$).

Remark. coNROCA is not the only weak model of computation that satisfies Theorem 4.5. Given e, we can effectively construct a nondeterministic log-*time* machine[3] that recognizes $\overline{Z'_e}$, where Z'_e is just like the Z_e of the above proof except that we replace Condition 5 with

5′. n is the least power of 2 such that Conditions 2–4 are possible.

We leave the details of the proof as an exercise to the reader. (The rectangularity of the traces is useful here, and the fact that n is a power of 2 makes it easier to find the depth s of the trace T.) Thus we have that $\emptyset''$ is computable in any function bounding $\mathcal{F}^{\mathrm{coNTIME}(\log n),\mathrm{DEC}}$. On the other hand, we cannot improve Theorem 4.5 to use deterministic ROCAs (DROCAs). This follows from Theorem 7.1 and the fact that coDROCA $\sqsubseteq$ DROCA $\sqsubseteq$ CFL.

Corollary 4.6 *If g bounds $\mathcal{F}^{\mathrm{coNROCA,coCE}}$ or $\mathcal{F}^{\mathrm{coNTIME}(\log n),\mathrm{coCE}}$, then $\emptyset'' \leq_{\mathrm{T}} g$.*

Proof: By (1) of Corollary 2.18, there is a computable function $f_1 \in \mathcal{F}^{\mathrm{coNROCA,CE}}$. A standard result in computability theory says that there is a computable function s such that, for all e and i, if $L(M_e) = \overline{L(M_i)}$, then $M_{s(e,i)}$ is a total machine deciding $L(M_e)$. Let f_2 be any function in $\mathcal{F}^{\mathrm{coNROCA,coCE}}$. Then for each e, $M_{s(f_1(e),f_2(e))}$ is a total machine recognizing SUBSEQ($\overline{L(H_e)}$), and so the function f defined by $f(e) = s(f_1(e), f_2(e))$ is in $\mathcal{F}^{\mathrm{coNROCA,DEC}}$. Now suppose that g bounds f_2. Defining the function $\hat{g}$ by $\hat{g}(e) = \max_{z \leq g(e)} s(f_1(e), z)$, we get $\hat{g}(e) \geq s(f_1(e), f_2(e)) = f(e)$ for all e, and so $\hat{g}$ bounds f. Thus by Theorem 4.5, we get $\emptyset'' \leq_{\mathrm{T}} \hat{g} \leq_{\mathrm{T}} g$. The case of $\mathcal{F}^{\mathrm{coNTIME}(\log n),\mathrm{coCE}}$ is similar. ∎

The following corollary to Corollary 4.6 is far weaker; however, we use it to motivate the next section.

Corollary 4.7 *If g bounds $\mathcal{F}^{\mathrm{P,REG}}$, then $\emptyset'' \leq_{\mathrm{T}} g$.*

Corollary 4.8 *$\emptyset'' \leq_w \mathcal{F}^{\mathrm{coNROCA,coCE}}$ and $\emptyset'' \leq_w \mathcal{F}^{\mathrm{coNTIME}(\log n),\mathrm{coCE}}$.*

Corollary 4.9 *If X is any oracle, then $\mathcal{F}^{\mathrm{coNROCA,coCE}} \not\leq_{\mathrm{tt}} X$.*

Proof: Assume, by way of contradiction, that there exists a set X and an $f \in \mathcal{F}^{\mathrm{coNROCA,coCE}}$ such that $f \leq_{\mathrm{tt}} X$ via $M^{()}$. We show that f is computably bounded. Given e one can simulate all possible paths of $M^{()}(e)$. (This technique is folklore.) This gives a finite number of candidates for $f(e)$. The largest answer is a bound on $f(e)$. ∎

Corollary 4.10 *If* coNROCA $\sqsubseteq \mathcal{C} \sqsubseteq$ CE *and* REG $\sqsubseteq \mathcal{D} \sqsubseteq$ coCE *then $\mathcal{F}^{\mathcal{C},\mathcal{D}} \equiv_w \emptyset''$.*

Proof: This follows from Corollaries 3.3 and 4.8 using Lemma 2.17. ∎

[3]The kind of log-time machine we have in mind here is equipped with an address tape used to read symbols on the read-only input in a random access fashion. To read the jth symbol, j is written on the address tape in binary, a special query state is entered, and in one step the address tape is erased and the symbol is recorded in the state of the machine. Erasing the address tape each time restricts the machine to making only $O(1)$ many input reads along any path.

5 How Hard Is It to Find the Size?

Consider the function that takes (say) a polynomial-time Turing machine M and outputs the number of states in the minimal DFA for SUBSEQ($L(M)$). Or the number of nonterminals in the smallest CFG in Chomsky Normal Form (CNF) for SUBSEQ($L(M)$). By Corollary 4.7 these functions grow faster than any computable-in-$\emptyset'$ function. What if we are given a polynomial-time Turing Machine M and promised that the number of states in the minimal DFA for SUBSEQ($L(M)$) is $\leq c$? We still cannot compute the number of states but *how many queries does it take to compute it?* What if we are not given a bound? We can still see how many queries it takes as a function of the answer.

In this section we try to pin down the complexity of these functions by their Turing degree and the number of queries (to some oracle) needed to compute them. The number of queries itself may be a function of the output. The final upshot will be that this problem has the exact same results as the unbounded search problem.

5.1 Unbounded Search

The material in this subsection is taken from [5, 2]. The base of the log function is 2 throughout.

Definition 5.1 The *Unbounded Search Problem* is as follows. Alice has a natural number n (there are no bounds on n). Bob is trying to determine what n is. He can ask questions of the form "Is $n \leq a$?" How many questions does Bob need to determine n? The number of questions will be a function of n.

Definition 5.2 The function $h : \mathbb{N} \to \mathbb{N}$ satisfies *Kraft's Inequality* if $\sum_{i=1}^{\infty} 2^{-h(i)} \leq 1$. We denote this sum by $Kr(h)$.

Note 5.3 If h satisfies Kraft's inequality, then there is a prefix-free code for $\mathbb{N}$ where n is coded by a string of length $h(n)$.

Example 5.4

1. $h(n) = (1+\epsilon)\log n + O(1)$ satisfies Kraft's inequality for any fixed $\epsilon > 0$.
2. $h(n) = \log n + \log\log n + \log\log\log n + \cdots + \log^{(\log^* n)} n + O(1)$ satisfies Kraft's inequality [2].
3. $h(n) = \log n - c$ violates Kraft's inequality for any constant c.

Definition 5.5 A real number α is *computable* if the function that maps i to the ith bit of α is computable.

Lemma 5.6 ([2]) *Let h be a monotone increasing, computable function such that $Kr(h)$ is computable. The unbounded search problem can be solved with $h(n)$ queries iff h satisfies Kraft's inequality.*

5.2 Definitions, Notation, and Lemmas from Bounded Queries

The Definitions and Notations in this section are originally from [1, 4], but are also in [14]. We only touch on the parts of bounded queries that we need; there are many variants on these definitions.

Definition 5.7 Let A be a set, and let $n \geq 1$.

1. $\mathrm{C}_n^A : \mathbb{N}^n \to \{0,1\}^n$ is defined by $\mathrm{C}_n^A(x_1, \ldots, x_n) = A(x_1)A(x_2)\cdots A(x_n)$.
2. $\#_n^A : \mathbb{N}^n \to \{0, \ldots, n\}$ is defined by $\#_n^A(x_1, \ldots, x_n) = |A \cap \{x_1, \ldots, x_n\}|$.

Convention: K is the set of all i such that $M_i(0)$ halts. (K is m-equivalent to $\emptyset'$.)

Definition 5.8 Let f, g be functions from $\mathbb{N}$ to $\mathbb{N}$, $A \subseteq \mathbb{N}$, and $n \in \mathbb{N}$. (An input to f may be a tuple of numbers; however, we code it as a number.) We say that $f \in \mathrm{FQ}(g(x), A)$ if $f \leq_{\mathrm{T}} A$ via an algorithm that, on input x, makes at most $g(x)$ queries to A. In particular, we say that $f \in \mathrm{FQ}(g(f(x)), A)$ if $f \leq_{\mathrm{T}} A$ via an algorithm that, on input x, makes at most $g(f(x))$ queries to A. (This latter case, which we will use a lot, is unusual in that the number of queries depends on the *output* of f.)

Definition 5.9 Let f be a partial function, and let $m \geq 1$. $f \in \mathrm{EN}(m)$ if there is a computable function h such that, for every x on which f is defined, $f(x) \in W_{h(x)}$ and $|W_{h(x)}| \leq m$. Intuitively, on input x, a process will enumerate $\leq m$ numbers, one of which will be $f(x)$. It is not known which one is $f(x)$, nor if the process will stop.

The following lemma establishes the relationship between query complexity and enumeration complexity.

Lemma 5.10 ([4]) *Let $f : \mathbb{N} \to \mathbb{N}$ and let $n \in \mathbb{N}$.*

1. $(\exists X)[f \in \mathrm{FQ}(n, X)] \Rightarrow f \in \mathrm{EN}(2^n)$.
2. $f \in \mathrm{EN}(2^n) \Rightarrow (\exists X \equiv_{\mathrm{T}} f)[f \in \mathrm{FQ}(n, X)]$

The following two lemmas give us upper bounds on the kind of unbounded search problems we will be concerned with.

Lemma 5.11 *Let $f : \mathbb{N} \to \mathbb{N}$. Let h be a monotone increasing, computable function such that $Kr(h)$ is computable and $Kr(h) \leq 1$. Then $(\exists X \equiv_{\mathrm{T}} f)[f \in \mathrm{FQ}(h(f(x)), X)]$.*

Proof: Let $X = \{(n, m) : f(n) \leq m\}$. The rest follows from Lemma 5.6. ∎

Note 5.12 Lemma 5.6 does not give lower bounds on the number of queries to compute f. Lemma 5.6 is about the general unbounded search problem. A particular function f may have properties that allow for computing it with fewer queries. As an extreme example, f could be computable and hence computable with 0 queries.

Notation 5.13 If $\gamma : \mathbb{N} \to \mathbb{N}$ is any function, let

$$\gamma_n(i) = \begin{cases} \gamma(i) & \text{if } 1 \leq \gamma(i) \leq n; \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Lemma 5.14 *Let $\gamma : \mathbb{N} \to \mathbb{N}$. Assume that there exists $k \in \mathbb{N}$ such that $(\forall n)[\gamma_n \in \text{EN}(2^k)]$ uniformly in n. Let h be any monotone increasing, unbounded, computable function. Then there exists X such that $\gamma \in \text{FQ}(h(\gamma(x)) + 1 + k, X)$.*

Proof: Let $X_1 = \{(n, m) : \gamma(n) \leq m\}$. Let H be the following function:

$$H(y) = \text{ the least } x \text{ such that } h(x) \geq y.$$

The following algorithm shows $\gamma \in \text{FQ}(h(\gamma(x)) + k + 1, X)$ for an X defined below.

1. Input(x).
2. Using X_1 ask $\gamma(x) \leq H(1)$? $\gamma(x) \leq H(2)$? etc. until you get a YES answer. (This takes $h(\gamma(x)) + 1$ queries.)
3. Let n be such that we now know $\gamma(x) \leq n$. Hence $\gamma(x) = \gamma_n(x)$. Since $\gamma_n(x) \in \text{EN}(2^k)$ uniformly in n, by Lemma 5.10 we have $(\exists X_2)[\gamma_n \in \text{FQ}(k, X_2)]$. Ask X_2 the relevant queries to find $\gamma_n(x)$.

The entire algorithm took $h(\gamma(x)) + 1 + k$ queries to $X = X_1 \oplus X_2$. ∎

The following two lemmas give us lower bounds on the number of queries needed for certain problems. They will provide problems to reduce to in order to get lower bounds.

Lemma 5.15 ([4]) $\#^K_{n-1} \notin \text{EN}(n-1)$

The following lemma is a modification of a similar lemma from both [3] and [13]. It will enable us to prove lower bounds on the number of queries certain functions require to compute.

Lemma 5.16 *Let $X \subseteq \mathbb{N}$, and $h : \mathbb{N} \to \mathbb{N}$ be any function. Let $\gamma : \mathbb{N} \to \mathbb{N}$. If $\gamma \in \text{FQ}(h(\gamma(x)), X)$ and $(\forall n)[\gamma_n \notin \text{EN}(n-1)]$, then h satisfies Kraft's inequality, namely, $\sum_{i=1}^{\infty} 2^{-h(i)} \leq 1$.*

Definition 5.17 If $f_1, f_2 : \mathbb{N} \to \mathbb{N}$ then $f_1 \leq_1 f_2$ means that there are computable functions S, T such that, for all x, $f_1(x) = S(f_2(T(x)))$. Essentially one can compute $f_1(x)$ with one query to f_2.

The next lemma will be used repeatedly.

Lemma 5.18 *Let $\gamma : \mathbb{N} \to \mathbb{N}$. Assume that, for all n, $\#^K_{n-1} \leq_1 \gamma_n$. Let h be a monotone increasing, computable function such that $Kr(h)$ is computable. If there exists an $X \equiv_{\text{T}} \gamma$ such that $\gamma \in \text{FQ}(h(\gamma(x)), X)$, then h satisfies Kraft's inequality. If h satisfies Kraft's inequality, then $(\exists X \equiv_{\text{T}} \gamma)[\gamma \in \text{FQ}(h(\gamma(x)), X)]$.*

Proof: If h satisfies Kraft's inequality, then, by Lemma 5.11, there exists an $X \equiv_{\text{T}} \gamma$ such that $\gamma \in \text{FQ}(h(\gamma(x)), X)$. (This part did not require the premise on γ_n.)

Assume $(\forall n)[\#^K_{n-1} \leq_1 \gamma_n]$. By Lemma 5.15, $\gamma_n \notin \text{EN}(n-1)$. By Lemma 5.16, if $\gamma \in \text{FQ}(h(\gamma(x)), X)$, then h satisfies Kraft's inequality. ∎

5.3 Given a C.E. Set A, How Hard Is It to Find the Number of States in the Minimal DFA for SUBSEQ(A)?

Convention We allow DFAs to have states q such that there exists $\sigma \in \Sigma$ with $\delta(q, \sigma)$ undefined (and hence any string for which this happens is rejected).[4]

Recall that $\mathcal{F}^{X,\mathrm{REG}}$ was defined so that its members output *any* appropriate DFA. In the next definition we want to look at the minimum DFA.

Definition 5.19 Let $\mathrm{NS}^{X,\mathrm{REG}}(e)$ be the minimum number of states in an output $f(e)$ for any $f \in \mathcal{F}^{X,\mathrm{REG}}$. Let $\mathrm{NS}_n^{X,\mathrm{REG}}(e)$ be defined by letting γ in Notation 5.13 be $\mathrm{NS}^{X,\mathrm{REG}}$.

First we look at the Turing degree of $\mathrm{NS}^{X,\mathrm{REG}}$ for various machine classes X.

Theorem 5.20

1. $\mathrm{NS}^{\mathrm{CFL,REG}}$ *is computable.*
2. *If* coNROCA $\sqsubseteq X \sqsubseteq$ CE*, then* $\mathrm{NS}^{X,\mathrm{REG}} \equiv_{\mathrm{T}} \emptyset''$.

Proof: Item 1 follows immediately from Theorem 7.1 and the fact that we can effectively minimize DFAs. Item 2 follows easily from Theorem 4.5 and Corollary 4.10. (For the lower bound, note that from $\mathrm{NS}^{X,\mathrm{REG}}$ one can effectively compute a function bounding $\mathcal{F}^{X,\mathrm{REG}}$, and thus $\emptyset'' \leq_{\mathrm{T}} \mathrm{NS}^{X,\mathrm{REG}}$.) ∎

Now we concentrate on the number of queries required to compute $\mathrm{NS}^{X,\mathrm{REG}}$. We will be looking at $X \in \{\mathrm{CE}, \mathrm{P}, \mathrm{PU}\}$, where PU is P restricted to unary languages.

Theorem 5.21 *Let h be a monotone increasing, computable function such that $Kr(h)$ is computable. There exists an X such that* $\mathrm{NS}^{\mathrm{CE,REG}} \in \mathrm{FQ}(h(\mathrm{NS}^{\mathrm{CE,REG}}(x)), X)$ *iff h satisfies Kraft's inequality.*

Proof: By Lemma 5.18 we need T such that

$$\#_{n-1}^K(x_1, \ldots, x_{n-1}) = \mathrm{NS}_n^{\mathrm{CE,REG}}(T(x_1, \ldots, x_{n-1})) - 1.$$

Let $T(x_1, \ldots, x_{n-1})$ be an index for the following Turing machine M:

1. Input string $s \in 0^*$.
2. If $s = \lambda$ accept.
3. If $s = 0^i$, run all $n-1$ computations $M_{x_1}(0), \ldots, M_{x_{n-1}}(0)$ until i have halted. If this happens, accept.

Let A be the language recognized by M. Note that $A = \mathrm{SUBSEQ}(A)$. Also notice that, for all i, $0 \leq i \leq n-1$,

$$\#_{n-1}^K(x_1, \ldots, x_{n-1}) = i \;\Rightarrow\; A = A_i = \{0^0, \ldots, 0^i\}.$$

The min DFA for A_i uses $i+1$ states, so

$$\#_{n-1}^K(x_1, \ldots, x_{n-1}) = \mathrm{NS}_n^{\mathrm{CE,REG}}(T(x_1, \ldots, x_{n-1})) - 1.$$

∎

[4] This convention is only used to simplify our expressions and eliminate some minor complications with $n = 2$.

5.4 Given a C.E. Set A, How Hard is it to Find the Number of Nonterminals in the Minimal CFG for SUBSEQ(A)?

Convention We assume that all CFGs are in Chomsky Normal Form (CNF).

Lemma 5.22 *Suppose that G is a CNF CFG and Σ is a set of terminals (not necessarily all the terminals of G) such that $L(G) \cap \Sigma^*$ is finite and contains at least one nonempty string. Let N be the length of the longest string in $L(G) \cap \Sigma^*$. We have the following:*

1. *G has at least $1 + \lceil \log N \rceil$ many nonterminals.*
2. *If $L(G)$ is infinite, then G has at least $2 + \lceil \log N \rceil$ many nonterminals.*

Proof: For the first part, let us look at a parse tree for a given nonempty string in $L(G) \cap \Sigma^*$. No nonterminal can appear twice in a path from the root to a leaf. Otherwise, it could repeat indefinitely, and thus $L(G) \cap \Sigma^*$ would be infinite. Since the CFG is in CNF, each internal node in the parse tree has as children either exactly two nonterminals or only one terminal (leaf). So, the parse tree is a binary tree. Thus, if the longest string in $L(G) \cap \Sigma^*$ is of length N, then the tree must have N leaves, and thus a path of length at least $1 + \lceil \log N \rceil$. Since no nonterminal can appear twice in any path, there must be at least $1 + \lceil \log N \rceil$ nonterminals in the grammar.

For the second part, suppose that G has at most $1 + \lceil \log N \rceil$ many nonterminals. We show that $L(G)$ must be finite. Consider a parse tree T for some string in $L(G) \cap \Sigma^*$ of length N. As in the first part, no nonterminal can be repeated on any path of T, and in order to accommodate N terminals, there must be a path p in T of length at least $1 + \lceil \log N \rceil$. So p contains each of the nonterminals of G exactly once, and this implies that every nonterminal of G is capable of generating a nonempty string in Σ^*.

We claim that this in turn implies that no nonterminal can *ever* be repeated on *any* path of *any* parse tree of G whatsoever.

To prove this claim, suppose for the sake of contradiction that there is some parse tree T' containing a path p' on which some nonterminal A occurs at least twice. Then we "pump" the segment of p' between the two A's repeatedly (at least $N+1$ times) to obtain a parse tree T'' with a path p'' containing at least $N + 1$ occurrences of A. Then, for every nonterminal node B not on p'' but whose parent is on p'', we replace the subtree rooted at B with another one (also with root B) that generates some nonempty string in Σ^*. (We do the same for the deepest nonterminal node on p'' as well.) The resulting tree T''' is a parse tree for some string in Σ^* longer than N, contradicting our choice of N. This proves the claim.

The claim implies that every parse tree of G has depth at most $1 + \lceil \log N \rceil$, and so $L(G)$ is finite. ∎

Definition 5.23 Let $\mathrm{NT}^{X,\mathrm{CFL}}(e)$ be the minimum number of nonterminals in $f(e)$ for any $f \in \mathcal{F}^{X,\mathrm{CFL}}$. Let $\mathrm{NT}_n^{X,\mathrm{CFL}}(e)$ be defined by letting γ in Notation 5.13 be $\mathrm{NT}^{X,\mathrm{CFL}}$.

Theorem 5.24 $\mathrm{NT}^{\mathrm{CFL},\mathrm{CFL}}$ *is computable, and* $\mathrm{NT}^{X,\mathrm{CFL}} \equiv_{\mathrm{T}} \emptyset''$ *for all* coNROCA $\sqsubseteq X \sqsubseteq$ CE.

Proof: This is similar to the proof of Theorem 5.20. Note that there are only finitely many inequivalent context-free grammars in CNF with a given number of nonterminals. ∎

Theorem 5.25 *Let h be a monotone increasing unbounded computable function such that $Kr(h)$ is computable. There exists an X such that $\mathrm{NT}^{\mathrm{CE,CFL}} \in \mathrm{FQ}(h(\mathrm{NT}^{\mathrm{CE,CFL}}(x)), X)$ iff h satisfies Kraft's inequality.*

Proof: By Lemma 5.18 we need T such that

$$\#^K_{n-1}(x_1, \ldots, x_{n-1}) = \mathrm{NT}^{\mathrm{CE,CFL}}_n(T(x_1, \ldots, x_{n-1})) - 1.$$

Let $T(x_1, \ldots, x_{n-1})$ be an index for the following Turing machine M:

1. Input string $s \in 0^*$.
2. If $s = \lambda$ accept.
3. If $s = 0^j$ with $2^{i-1} < j \leq 2^i$, run all $n-1$ computations $M_{x_1}(0), \ldots, M_{x_{n-1}}(0)$ until i have halted. If this happens, then accept.

Let A be the language recognized by M. Note that $A = \mathrm{SUBSEQ}(A)$. Also notice that, for all i, $0 \leq i \leq n-1$,

$$\#^K_{n-1}(x_1, \ldots, x_{n-1}) = i \ \Rightarrow \ A = A_i = \{0^0, \ldots, 0^{2^i}\}.$$

By Part 1 of Lemma 5.22 with $\Sigma = \{0\}$, any CFG for A_i has at least $i+1$ nonterminals. Further, a CFG for A_i can easily be constructed that has exactly $i+1$ nonterminals. Thus $i+1$ is the minimum number of nonterminals in any CFG for A_i, and so

$$\#^K_{n-1}(x_1, \ldots, x_{n-1}) = \mathrm{NT}^{\mathrm{CE,CFL}}_n(T(x_1, \ldots, x_{n-1})) - 1.$$

∎

5.5 Given a Set $A \in \mathrm{P}$, How Hard Is It to Find the Number of States in the Minimal DFA for $\mathrm{SUBSEQ}(A)$?

Theorem 5.26 *Let h be a monotone increasing, computable function such that $Kr(h)$ is computable. There exists an X such that $\mathrm{NS}^{\mathrm{P,REG}} \in \mathrm{FQ}(h(\mathrm{NS}^{\mathrm{P,REG}}(x)), X)$ iff h satisfies Kraft's inequality.*

Proof: By Lemma 5.18 we need T such that

$$\#^K_{n-1}(x_1, \ldots, x_{n-1}) = \mathrm{NT}^{\mathrm{P,REG}}_n(T(x_1, \ldots, x_{n-1})) - 1.$$

Let $T(x_1, \ldots, x_{n-1})$ be an index for the following polynomial-time Turing machine M, which refers to the computations $M_{x_1}(0), \ldots, M_{x_{n-1}}(0)$:

$$\begin{aligned}
M(0^s) = 1 \quad &\text{iff} \quad \text{at least 0 of the } n-1 \text{ computations halt in} \leq s \text{ steps} \\
M(0^s1) = 1 \quad &\text{iff} \quad \text{at least 1 of the } n-1 \text{ computations halt in} \leq s \text{ steps} \\
M(0^s11) = 1 \quad &\text{iff} \quad \text{at least 2 of the } n-1 \text{ computations halt in} \leq s \text{ steps} \\
&\vdots \\
M(0^s1^i) = 1 \quad &\text{iff} \quad \text{at least } i \text{ of the } n-1 \text{ computations halt in} \leq s \text{ steps} \\
&\vdots
\end{aligned}$$

M rejects all other inputs. Let A be the language decided by M. For $0 \leq i \leq n-1$ note the following:

$$\#_{n-1}^K(x_1, \ldots, x_{n-1}) = i \ \Rightarrow \ \text{SUBSEQ}(A) = A_i = \bigcup_{b=0}^{i} 0^*1^b.$$

The minimal DFA for A_i has $i+1$ states. Hence

$$\#_{n-1}^K(x_1, \ldots, x_{n-1}) = \text{NS}_n^{\text{P,REG}}(T(x_1, \ldots, x_{n-1})) - 1.$$

∎

Remark. Theorem 5.26 can be strengthened by replacing P with coNROCA everywhere in the statement of theorem. This requires a routine modification of the proof, which we omit.

5.6 Given a Set $A \in \text{P}$, How Hard Is It to Find the Number of Nonterminals in the Minimal CFL for $\text{SUBSEQ}(A)$?

Theorem 5.27 *Let h be a monotone increasing, computable function such that $Kr(h)$ is computable. There exists an X such that $\text{NT}^{\text{P,CFL}} \in \text{FQ}(h(\text{NT}^{\text{P,CFL}}(x)), X)$ iff h satisfies Kraft's inequality.*

Proof: By Lemma 5.18 we need T such that

$$\#_{n-1}^K(x_1, \ldots, x_{n-1}) = \text{NT}_n^{\text{P,CFL}}(T(x_1, \ldots, x_{n-1})) - 1.$$

Let $T(x_1, \ldots, x_{n-1})$ be an index for the following polynomial-time Turing machine M, which refers to the computations $M_{x_1}(0), \ldots, M_{x_{n-1}}(0)$:

$$\begin{aligned}
M(0^s1) = 1 \quad &\text{iff} \quad \text{at least 1 of the } n-1 \text{ computations halt in} \leq s \text{ steps} \\
M(0^s11) = 1 \quad &\text{iff} \quad \text{at least 2 of the } n-1 \text{ computations halt in} \leq s \text{ steps} \\
M(0^s1111) = 1 \quad &\text{iff} \quad \text{at least 3 of the } n-1 \text{ computations halt in} \leq s \text{ steps} \\
&\vdots \\
M(0^s1^{2^{i-1}}) = 1 \quad &\text{iff} \quad \text{at least } i \text{ of the } n-1 \text{ computations halt in} \leq s \text{ steps} \\
&\vdots
\end{aligned}$$

M rejects all other inputs.

Let A be the language decided by M. If $\#_{n-1}^K(x_1, \ldots, x_{n-1}) = 0$, then $\text{SUBSEQ}(A) = \emptyset$, which is the language of the trivial grammar with one nonterminal (the start symbol) and no productions.

Now suppose that $1 \leq i \leq n-1$ and note the following: if $\#_{n-1}^K(x_1, \ldots, x_{n-1}) = i$, then $\text{SUBSEQ}(A) = A_i = \bigcup_{b=0}^{2^{i-1}} 0^*1^b$. By Part 1 of Lemma 5.22 with $\Sigma = \{1\}$, the minimal CFL for A_i has at least i nonterminals, but this cannot be tight by Part 2 of the same lemma, since A_i is infinite. Thus the minimal CFL for A_i has at least $i+1$ nonterminals. The following grammar G_i for A_i has $i+1$ nonterminals:

- The nonterminals are $A_1, \ldots, A_i$ and Z.

- The start symbol is A_1.
- The productions are
  - $Z \to 0$
  - $A_1 \to \lambda$
  - $A_1 \to ZA_1$
  - $A_{n-1} \to A_n A_n$ (for all $2 \le n \le i$)
  - $A_n \to 1$ (for all $1 \le n \le i$)

Hence the minimal CFG for A_i has exactly $i+1$ nonterminals.

In any case, we have

$$\#^K_{n-1}(x_1, \ldots, x_{n-1}) = \mathrm{NT}^{\mathrm{P},\mathrm{CFL}}_n(T(x_1, \ldots, x_{n-1}))) - 1.$$

∎

Remark. Theorem 5.27 can be strengthened by replacing P with coNROCA everywhere in the statement of theorem. This requires a routine modification of the proof, which we omit.

5.7 Polynomial-Time Unary Languages

In the proofs of Theorems 5.21 and 5.25 we were able to get by with a unary language. In the proofs of Theorems 5.26 and 5.27 we used a binary language. Could we have used a unary language? The theorems in this section say no—in the unary case the complexity is substantially lower in terms of number of queries.

Theorem 5.28 $\mathrm{NS}^{\mathrm{PU},\mathrm{REG}}_n \in \mathrm{EN}(2)$

Proof: Note that the subsequence language of any unary language is either all strings shorter than a certain length, or 0^*.

Given a unary polynomial-time TM M, we run M on all strings of length up to $n-1$. There are two cases:

Case 1 M does not accept any of the strings. Then there are three possibilities, either M accepts nothing, or it accepts an infinite number of strings longer than n, or there is a longest string which it accepts, which has length at least n. The first two possibilities give that $\mathrm{SUBSEQ}(L(M)) = \emptyset$ or 0^* respectively, which require 1-state DFAs. The third gives that $\mathrm{SUBSEQ}(L(M))$ requires a DFA that has more than n states, and so we don't have to worry about it. In this case we enumerate $\{1\}$.

Case 2 M accepts some longest string with length $\ell < n$. Then there are still three possibilities, either M accepts nothing longer than ℓ, or it accepts an infinite number of strings longer than n, or there is a longest string which it accepts, which has length at least n. The first possibility creates an $(\ell+1)$-state DFA. The second possibility gives that $\mathrm{SUBSEQ}(L(M)) = 0^*$, which requires a 1-state DFA. The third gives that $\mathrm{SUBSEQ}(L(M))$ requires a DFA that has more than n states, and so we don't have to worry about it. In this case we enumerate $\{1, \ell+1\}$.

∎

Corollary 5.29 *Let h be any monotone increasing unbounded computable function. Then there exists X such that* $\mathrm{NS}^{\mathrm{PU,REG}} \in \mathrm{FQ}(h(\mathrm{NS}^{\mathrm{PU,REG}}(x)) + 2, X)$.

Proof: This follows from Lemmas 5.14 and Theorem 5.28 ∎

Corollary 5.30 *Let g be any monotone increasing unbounded computable function such that $(\forall x)[g(x) \geq 2]$. Then there exists X such that* $\mathrm{NS}^{\mathrm{PU,REG}} \in \mathrm{FQ}(g(\mathrm{NS}^{\mathrm{PU,REG}}(x)), X)$.

Proof: This follows from Corollary 5.29 with $h = g - 2$. ∎

Lemma 5.31 $\mathrm{NT}_n^{\mathrm{PU,CFL}} \in \mathrm{EN}(2)$

Proof: Note again that the subsequence language of any unary language is either all strings shorter than a certain length, or 0^*.

Given a unary polynomial-time TM M, we run M on all strings of length up to 2^{n-1}. There are two cases:

Case 1 M does not accept any of the strings. Then there are three possibilities, either M accepts nothing, or it accepts an infinite number of strings longer than 2^{n-1}, or there is a longest string which it accepts, which is longer than 2^{n-1}. The first two possibilities give that $\mathrm{SUBSEQ}(L(M)) = \emptyset$ or 0^* respectively, which require 1-nonterminal CFGs. The third gives that $\mathrm{SUBSEQ}(L(M))$ requires a CFG which has more than n nonterminals, and so we don't have to worry about it. In this case we enumerate $\{1\}$.

Case 2 M accepts some longest string with length of $\ell \leq 2^{n-1}$. Then there are still three possibilities, either M accepts nothing longer than ℓ, or it accepts an infinite number of strings longer than 2^{n-1}, or there is a longest string which it accepts, which is longer than 2^{n-1}. The first possibility creates a $k(\ell)$-nonterminal CFG for $\mathrm{SUBSEQ}(L(M))$, where $k(\ell)$ is the least number of nonterminals in any CNF CFG for $\{0^0, \ldots, 0^\ell\}$. (We note that $k(\ell) \leq \ell + 1$, and that $k(\ell)$ is computable from ℓ.) The second possibility gives that $\mathrm{SUBSEQ}(L(M)) = 0^*$, which requires a 1-nonterminal CFG. The third gives that $\mathrm{SUBSEQ}(L(M))$ requires a CFG that has more than n nonterminals, and so we don't have to worry about it. In this case we enumerate $\{1, k(\ell)\}$.

∎

Theorem 5.32 *Let h be any monotone increasing unbounded computable function. Then there exists X such that* $\mathrm{NT}^{\mathrm{PU,CFL}} \in \mathrm{FQ}(h(\mathrm{NT}^{\mathrm{PU,CFL}}(x)) + 2, X)$.

Proof: This follows from Lemmas 5.14 and Lemma 5.31 ∎

Corollary 5.33 *Let g be any monotone increasing unbounded computable function such that $(\forall x)[g(x) \geq 2]$. Then there exists X such that* $\mathrm{NT}^{\mathrm{PU,CFL}} \in \mathrm{FQ}(g(\mathrm{NT}^{\mathrm{PU,CFL}}(x)), X)$.

5.8 Summary of this Section

In the table below the row represents the device for A we are given, the column represents the device for SUBSEQ(A) we seek, and the entry is how many queries you need to determine the size. *Kraft* means (roughly) that the function can be computed in $h(n)$ queries—where n is the output—iff h satisfies Kraft's inequality. *Mono* means (roughly) that for any monotone increasing unbounded function $h \geq 2$ there exists X such that the function can be computed in $h(n)$ queries.

	REG	CFL
CE	Kraft	Kraft
P	Kraft	Kraft
PU	Mono	Mono

6 The Oracle Model

In this section we look at questions along the lines of "Given oracle access to a language A (or to A' or to A''), how hard is it to find a DFA for SUBSEQ(A)?". Recall that A' is the Turing jump of A. Fixing some total order on Σ, we let $\leq$ be the usual length-first lexicographic ordering induced on Σ^*.

In this section the Turing machines will not have an input and an output in the usual sense. Instead, the Turing machines will be oracle Turing machines, and the oracle can be considered to be the input; the Turing machine will output answers from time to time. This is similar to the Inductive Inference model of learning [6, 7, 15], and we study the task of learning SUBSEQ(A) more fully in another paper [10].

Definition 6.1 Let $M^{()}$ be an oracle Turing Machine and A an oracle.

1. $M^A \downarrow= e$ means that M^A, when run, will run forever and output answers from time to time, but eventually they will all be e.
2. $M^A = e$ means that M^A, when run, will output only one value and that value is e.

Note 6.2 By standard results in computability theory, a language is uniformly computable from its jump, i.e., there is an oracle Turing machine $J^{()}$ such that J is total for all oracles and $A = L(J^{A'})$ for all A. Thus oracle access to A' provides one with access to A as well.

Theorem 6.3 *There exists an oracle Turing machine $M^{()}$ such that, for any language $A \subseteq \Sigma^*$, there is a DFA F_e such that $L(F_e) = \text{SUBSEQ}(A)$, and $M^{A'} \downarrow= e$.*

Proof: Recall from Definition 2.1 that $x \preceq y$ means that x is a subsequence of y.

For any language A, the language SUBSEQ(A) is closed downward under $\preceq$, and so by Lemma A.3 there is a finite set $B_A = \{z_1, \ldots, z_n\} \subseteq \Sigma^*$ such that

$$\text{SUBSEQ}(A) = \{w \in \Sigma^* : (\forall z \in B_A)[z \not\preceq w]\}. \tag{1}$$

We note that for each A there is a *unique* $\preceq$-antichain B_A satisfying (1), namely, the set of $\preceq$-minimal elements of the complement of SUBSEQ(A); Higman's result [19] insists that this antichain

must be finite. (A $\preceq$-*antichain* is a set whose elements are pairwise $\preceq$-incomparable.) The idea is that M uses A' to approximate the set B_A and outputs a DFA for its approximation. Since B_A is finite, M will eventually find it.

For every $n \in \mathbb{N}$, in order, $M^{A'}$ does the following: Runs through all strings $y \in \Sigma^*$ of length less than n. For each such y asks the oracle A' whether there exists a $z \in A$ with $y \preceq z$. Let Y be the set of all y for which the answer is "no," and let $B_{A,n}$ be the set of all $\preceq$-minimal elements of Y. (Evidently, $B_{A,n} = B_A \cap \Sigma^{<n}$, and so $B_{A,n} = B_A$ for sufficiently large n.) Finally construct a DFA F for the language defined by

$$\{w \in \Sigma^* : (\forall z \in B_{A,n})\ z \not\preceq w\} = L(F).$$

Then output the least index e such that $L(F_e) = L(F)$. (Note that although there are many possible DFAs F, $M^{A'}$ must eventually output the same *index* each time, which is why it chooses the least one.)

For all but a finite number of n, $B_{A,n} = B_A$. Hence, for all but finitely many n the same index will be output. We denote this index e and note that $L(F_e) = \text{SUBSEQ}(A)$ as desired. ∎

Theorem 6.4 *There exists an oracle Turing machine $N^{()}$ such that, for any language $A \subseteq \Sigma^*$, there is a DFA F_e such that $L(F_e) = \text{SUBSEQ}(A)$, and $N^{A''} = e$.*

Proof: Let $M^{()}$ be the machine of Theorem 6.3. The machine $N^{A''}$ does the following: run $M^{A'}$ until it outputs an answer e (which may not be its final answer). Then ask A''

$$(\exists y)[F_e(y) = 0 \wedge y \in \text{SUBSEQ}(A)] \vee (\exists y)[F_e(y) = 1 \wedge y \notin \text{SUBSEQ}(A)].$$

If the answer is YES then keep running $M^{A'}$ and repeat. Eventually the answer will be NO, which means that $L(F_e) = \text{SUBSEQ}(A)$. At this point output e. Note that this final e is the only output. ∎

Theorems 6.3 and 6.4 are tight in the sense that, as we now show, no OTM can do the same as $M^{()}$ with just A as an oracle, and no machine can do the same as $N^{()}$ with just A' as an oracle. In fact, for any candidate machine $M^{()}$ (respectively $N^{()}$) that claims

$$(\forall A)[M^A \downarrow = e \wedge L(F_e) = \text{SUBSEQ}(A)]$$

we can effectively find (an index for) a *decidable* language A for which this is not the case.

Theorem 6.5 *There is an effective procedure that, given as input the description of any oracle Turing machine $M^{()}$, outputs an index i of a Turing machine such that*

1. *M_i is total. Let $A = L(M_i)$.*
2. *If there exists e with $M^A \downarrow = e$, then $L(F_e) \neq \text{SUBSEQ}(L(M_i))$.*

Proof: Fix any $a \in \Sigma$. Given any oracle Turing machine $M^{()}$, we effectively construct M_i to behave as follows.

1. Input x

2. If $x \notin a^*$ then reject. (This will ensure that $A \subseteq a^*$.)

3. Let $x = a^n$ for $n \geq 0$. Run M_i recursively on all inputs a^j for $0 \leq j < n$. (Formally we use the Recursion Theorem.) Set $A_n := \{a^j : 0 \leq j < n \text{ and } M_i(a^j) \text{ accepts}\}$. (Note that by induction on n, M_i will halt on all these inputs a^j, and thus we do not get stuck in this step.)

4. Run M^{A_n} for $n-1$ steps. Let e be the most recent output of M^{A_n} within that time. If there is no such e, then reject. (Note that M^{A_n} does not have time within its first $n-1$ steps to query A_n on any string of the form a^j for $j \geq n$, and so M^{A_n} behaves the same in its first $n-1$ steps as M^A.)

5. If $L(F_e)$ is finite then accept, otherwise reject.

It is evident by induction on n that M_i is total. Let $A = L(M_i)$, the set of all strings accepted by M_i. We show that M^A does not converge to e with $\text{SUBSEQ}(A) = L(F_e)$. There are several cases.

Case 1: M^A does not converge. Then clearly we are done.

Case 2: M^A converges to e such that $L(F_e) \not\subseteq a^*$. Then since $A \subseteq a^*$ we have $\text{SUBSEQ}(A) \subseteq a^*$, and so $L(F_e) \neq \text{SUBSEQ}(A)$.

Case 3: M^A converges to e and $L(F_e)$ is infinite. Then for all large enough n, M_i does not accept a^n in Step 5, and so A is finite. Hence $\text{SUBSEQ}(A)$ is finite and $L(F_e) \neq \text{SUBSEQ}(A)$.

Case 4: M^A converges to e such that $L(F_e)$ is finite. Then for all large enough n, M_i accepts a^n in Step 5, and so A is infinite. Hence $\text{SUBSEQ}(A)$ is infinite and $L(F_e) \neq \text{SUBSEQ}(A)$.

The conclusion of these four cases is that if M^A converges to e, then $L(F_e) \neq \text{SUBSEQ}(A)$. ∎

Corollary 6.6 *There is an effective procedure that, given as input the description of any oracle Turing machine $N^{()}$, outputs a Turing machine M_i that recognizes a language $A = L(M_i) \subseteq \Sigma^*$ such that the following hold:*

1. *M_i is total, and hence M_i decides A.*

2. *If there exists e with $N^{A'} = e$, then $L(F_e) \neq \text{SUBSEQ}(A)$.*

Proof: For any language A, let $A_0 \subseteq A_1 \subseteq A_2 \subseteq \cdots \subseteq A'$ be a standard A-computable enumeration of A'. There is an oracle Turing machine that computes this enumeration relative to any A.

Given an oracle Turing machine $N^{()}$ as input, first we effectively construct an oracle Turing machine $M^{()}$ that, given any oracle A, merely simulates N^{A_n} for increasing values of n. More precisely, for $n = 0, 1, 2, 3, \ldots$, the computation M^A

1. dovetails the computations $N^{A_0}, N^{A_1}, \ldots, N^{A_n}$ for n steps each,

2. finds the *largest* $m \leq n$ (if it exists) for which N^{A_m} has output something, then

3. outputs the *first* output of N^{A_m}.

We then apply the effective procedure of Theorem 6.5 to $M^{()}$ to get a machine M_i that decides a language A.

Suppose that $N^{A'} = e$ for some e. Then there is a finite ℓ_0 such that, for all $\ell \geq \ell_0$, N^{A_ℓ} outputs e as its first output. [Here, $A_0 \subseteq A_1 \subseteq \cdots$ is the standard enumeration of A' and is unrelated to the similar notation found in the proof of Theorem 6.5.] But then clearly $M^A \downarrow = e$, and so by Theorem 6.5, $L(F_e) \neq \text{SUBSEQ}(A) = \text{SUBSEQ}(L(M_i))$. ∎

Theorem 6.5 and Corollary 6.6 can be easily relativized to any oracle X. This means that extra information independent of A is of no help in finding a DFA for SUBSEQ(A). Corollaries 6.7 and 6.8 are corollaries of the proofs of Theorem 6.5 and Corollary 6.6, respectively. The new proofs are routine modifications of the old ones, and we omit them.

Corollary 6.7 *There is an effective procedure that, given the description of any oracle Turing machine $M^{()}$, outputs the index i of an oracle Turing machine $M_i^{()}$ such that for any oracle X, M_i^X recognizes a language $A = L(M_i^X)$, and the following hold:*

1. *$M_i^{()}$ is total for all oracles, and hence $A \leq_{\text{tt}} X$ via $M_i^{()}$.*
2. *If there exists an e such that $M^{X \oplus A} \downarrow = e$, then $L(F_e) \neq \text{SUBSEQ}(A)$.*

Corollary 6.8 *There is an effective procedure that, given the description of any oracle Turing machine $N^{()}$, outputs the index i of an oracle Turing machine M_i such that for any oracle X, M_i^X recognizes a language $A = L(M_i^X)$, and the following hold:*

1. *$M_i^{()}$ is total for all oracles, and hence $A \leq_{\text{tt}} X$ via $M_i^{()}$.*
2. *If there is an e such that $N^{(X \oplus A)'} = e$, then $L(F_e) \neq \text{SUBSEQ}(A)$.*

7 The Complexity of $\mathcal{F}^{\text{CFL,REG}}$

The following theorem is due to van Leeuwen [29]. For completeness, we present it here with an altered proof.

Theorem 7.1 (van Leeuwen [29]) *$\mathcal{F}^{\text{CFL,REG}}$ is computable.*

Proof: In this proof, as is customary, we will often identify regular expressions with their corresponding languages. The meaning should be clear from the context.

We will be constructing a regular expression for SUBSEQ(A) by induction on the number of nonterminals in a grammar for A. First, we need a definition.

Definition 7.2 An *R-Grammar* is a structure $G = \langle V, \Sigma, \text{P}, S \rangle$ where

1. V is an alphabet,
2. Σ is the set of terminal symbols ($\Sigma \subseteq V$),
3. $S \in V - \Sigma$ is the start symbol, and

4. P is a finite set of production rules of the form $B \to M$ where $B \in V - \Sigma$ and M is a regular expression on the alphabet V.

An R-Grammar is just like a CFG except that instead of each production rule going to a fixed string, it goes to a regular set of strings. In the same paper van Leeuwen shows that R-grammars are exactly as powerful as CFGs.

In our construction of the regular expression for SUBSEQ(A) at each stage of the induction we will create an R-grammar with one fewer nonterminals. Eventually, we will end up with a regular expression for SUBSEQ(A).

This proof is unusual in that the base case is harder than the inductive step. To help the reader, we do the inductive step first.

Inductive Step

Let $n \geq 2$ and assume that given any R-grammar G with fewer than n nonterminals, we can effectively find a regular expression for SUBSEQ($L(G)$).

Let $G = \langle V, \Sigma, \mathrm{P}, S \rangle$ be an R-grammar with n nonterminals ($|V - \Sigma| = n$). Choose a $B \in V - \Sigma$ ($B \neq S$) and define the following R-grammar G_B:

1. The alphabet is V (though, as we will see, which symbols are terminals and nonterminals will change).

2. The terminals are $V - \{B\}$. Note that all of the nonterminals of G except B are terminals in G_B.

3. The productions are all the productions of G of the form $B \to M$. Note that, in G, M was a regular expression over V. This is still true; however the nonterminals from V that were in M are now terminals (except B).

4. The start nonterminal is B.

Note that G_B is exactly the grammar required to produce the strings of V achievable from B. Notice that it has only one nonterminal, namely B. We apply the inductive hypothesis for $n = 1$ and create the regular expression R_B for SUBSEQ($L(G_B)$).

We now create the grammar G'. Take the grammar G. Whenever B appears on the right hand side of a production, replace it with R_B. Whenever B appears on the left hand side of a production, remove the production. Finally, remove B from the set of nonterminals. It is clear that $L(G) \subseteq L(G')$, and in fact $L(G') \subseteq$ SUBSEQ($L(G)$). Thus SUBSEQ($L(G')$) = SUBSEQ($L(G)$). Also, G' has one fewer nonterminals so by applying the inductive hypothesis once again, we can find R, the regular expression for SUBSEQ($L(G)$). This completes the inductive step.

Base Case

Given an R-grammar $G = \langle \{S\} \cup \Sigma, \Sigma, P, S \rangle$, with only one nonterminal, we must find R—the regular expression for SUBSEQ($L(G)$). We may assume without loss of generality that G has exactly one production

$$S \to M$$

for some regular expression M over $\{S\} \cup \Sigma$. (If not, combine the two or more productions $S \to M_1, S \to M_2, \ldots$ into the single production $S \to M_1 \cup M_2 \cup \cdots$.) We have two cases:

1. Every string that matches M has at most one occurrence of S in it.
2. There is a string matching M that contains at least two occurrences of S.

It's easy to test which case applies.

Case 1:

We replace $S \to M$ with the two productions

$$\begin{aligned} S &\to M_1 \\ S &\to M_2, \end{aligned}$$

where M_1 is a regular expression equivalent to $M \cap \Sigma^*$ and M_2 is equivalent to $M \cap (\Sigma^* S \Sigma^*)$. By assumption, the new grammar is equivalent to the old one. Obviously, no string in M_1 contains an occurrence of S, and *every* string in M_2 contains exactly one S.

We define the following sets:

- $Q_L = \{x : \exists y[xSy \in M_2]\}$. These are all strings over Σ which occur before the S in strings matching M_2. (Since M_2 is a regular expression, there may be many such x.)
- $Q_R = \{y : \exists x[xSy \in M_2]\}$. These are all strings over Σ which occur after the S in strings matching M_2.

Both of these sets are regular, and we can effectively find regular expressions for them. We let

$$C = \text{SUBSEQ}(Q_L^* M_1 Q_R^*)$$

and claim that $C = \text{SUBSEQ}(A)$. $\text{SUBSEQ}(A) \subseteq C$ is obvious.

To show that $C \subseteq \text{SUBSEQ}(A)$ we consider an arbitrary $u \in C$. Let $v \in Q_L^* M_1 Q_R^*$ be such that $u \preceq v$. The string v can be written as

$$v = p_1 p_2 \cdots p_a m q_1 q_2 \cdots q_b$$

where

- For $1 \leq r \leq a$, $p_r \in Q_L$.
- For $1 \leq r \leq b$, $q_r \in Q_R$.
- $m \in M_1$.

For each p_r there must be a p'_r such that $p_r S p'_r \in M_2$. Likewise, for each q_r there must be a q'_r such that $q'_r S q_r \in M_2$.

Thus the following string is in A:

$$w = p_1 p_2 \cdots p_a q'_b \cdots q'_1 m q_1 q_2 \cdots q_b p'_a \cdots p'_1.$$

Since $u \preceq v \preceq w$, the original string u is in $\text{SUBSEQ}(A)$.

Case 2:

Similarly to Case 1, we replace $S \to M$ with two productions $S \to M_1$ and $S \to M_2$, where M_1 is equivalent to $M \cap \Sigma^*$ as in Case 1, but now M_2 is equivalent to $M \cap (\Sigma \cup S)^* S (\Sigma \cup S)^*$. No strings in M_1 contain any S's and every string in M_2 contains at least one S.

If $M_1 = \emptyset$ (which is easy to check), then $A = \text{SUBSEQ}(A) = \emptyset$ and we're done, so assume there is some string $m \in M_1$.

Because we might have many S in a string in M_2, we need to define three sets instead of two:

- $Q_L = \{x \in \Sigma^* : \exists y[xSy \in M_2]\}$. These are the strings that occur before the first S in elements of M_2.
- $Q_M = \{x \in \Sigma^* : \exists y_1, y_2[y_1 S x S y_2 \in M_2]\}$. These are the strings that occur in between two adjacent S's in strings in M_2.
- $Q_R = \{x \in \Sigma^* : \exists y[ySx \in M_2]\}$. These are the strings that occur after the final S in strings in M_2.

Again, all these sets are regular, and we can effectively find regular expressions for them. Note that Q_L, Q_M, Q_R are all nonempty, otherwise Case 1 would hold. Let

$$C = \text{SUBSEQ}((M_1 \cup Q_L \cup Q_M \cup Q_R)^*).$$

Again, we claim that $\text{SUBSEQ}(A) = C$.

To see that $\text{SUBSEQ}(A) \subseteq C$, we derive an arbitrary string $u \in A$ and see that each character at a leaf came from a string in either Q_L, Q_M, Q_R, or M_1, so each character in any $u' \preceq u$ did as well, and thus $u' \in C$.

To show that $C \subseteq \text{SUBSEQ}(A)$, we consider an arbitrary string $u \in C$. Given u, we can find $v \succeq u$ such that

$$v = m_1 p_1 q_1 r_1 m_2 p_2 q_2 r_2 \cdots m_t p_t q_t r_t$$

with the $m_i \in M_i$, $p_i \in Q_L$, $q_i \in Q_M$, and $r_i \in Q_R$.

For each p_i there must be a p_i' such that $p_i S p_i' \in M_2$. Likewise, for each q_i there must be q_i', q_i'' such that $q_i' S q_i S q_i'' \in M_2$, and for each r_i there must be an r_i' such that $r_i' S r_i \in M_2$. Note that the primed strings may contain more copies of S.

Since we are in Case 2, there must be some string z in M_2 with at least two S's. Using z we can expand S out enough times so that we have a string with at least $4t$ many S's—an S for each m_i, p_i, q_i, r_i in the string v. We can replace these S with m_i, $p_i S p_i'$, $q_i' S q_i S q_i''$, or $r_i' S r_i$ as appropriate. Any remaining S can be replaced with the string $m \in M_1$. The resulting string is clearly in A, and thus v, which is a subsequence of this string, is in $\text{SUBSEQ}(A)$. Finally, since $u \preceq v$ we have that $u \in \text{SUBSEQ}(A)$ as well.

Note that requiring two S's in some string in M is necessary for the proof of this case to work, because C allows for arbitrary ordering of the m, p, q, r which would not be possible in Case 1. ∎

8 Is $\mathcal{F}^{\mathcal{C},\mathcal{D}}$ Ever Not Muchnik Equivalent to $\emptyset$ or $\emptyset''$?

By Corollaries 2.18, 4.10, and Theorem 7.1, we have that for $\mathcal{C}, \mathcal{D} \in \{\text{REG}, \text{coNROCA}, \text{CFL}, \text{P}, \text{CE}\}$ either $\mathcal{F}^{\mathcal{C},\mathcal{D}} \equiv_w \emptyset$ or $\mathcal{F}^{\mathcal{C},\mathcal{D}} \equiv_w \emptyset''$. Do there exist two classes $\mathcal{C}$ and $\mathcal{D}$ such that $\mathcal{F}^{\mathcal{C},\mathcal{D}} \equiv_w \emptyset'$? How about other Turing degrees? We know of no natural classes and we doubt such exist. However, we can construct such classes.

Theorem 8.1 *Let X be any c.e. set. There exists a class $\mathcal{C}$ of total Turing machines such that for any $\mathcal{D} \in \{\text{REG}, \text{CFL}\}$, $\mathcal{F}^{\mathcal{C},\mathcal{D}} \equiv_w X$.*

Proof: Let X_s be X after the first s stages of enumeration. Define C_e to be the unary Turing machine that does the following:

$$C_e(0^s) = \begin{cases} 1 & \text{if } e \in X_s; \\ 0 & \text{if } e \notin X_s. \end{cases}$$

Define $\mathcal{C} = \{C_1, C_2, \ldots\}$.

Let $f \in \mathcal{F}^{\mathcal{C},\mathcal{D}}$. Note that

- $e \in X \Rightarrow L(C_e) \neq \emptyset \Rightarrow \lambda \in \text{SUBSEQ}(L(C_e)) \Rightarrow D_{f(e)}(\lambda) = 1$.
- $e \notin X \Rightarrow L(C_e) = \emptyset \Rightarrow \lambda \notin \text{SUBSEQ}(L(C_e)) \Rightarrow D_{f(e)}(\lambda) = 0$.

Hence $e \in X$ iff $D_{f(e)}(\lambda) = 1$. Therefore $X \leq_{\text{T}} f$, and so $X \leq_w \mathcal{F}^{\mathcal{C},\mathcal{D}}$.

To see that $\mathcal{F}^{\mathcal{C},\text{REG}} \leq_w X$, fix indices i_0 and i_1 such that $L(F_{i_0}) = \emptyset$ and $L(F_{i_1}) = 0^*$. Define

$$f(e) = \begin{cases} i_1 & \text{if } e \in X; \\ i_0 & \text{if } e \notin X. \end{cases}$$

Note that

- $e \in X \;\Rightarrow\; |L(C_e)| = \infty \;\Rightarrow\; \text{SUBSEQ}(L(C_e)) = 0^* = L(F_{i_1}) = L(F_{f(e)})$.
- $e \notin X \;\Rightarrow\; L(C_e) = \emptyset \;\Rightarrow\; \text{SUBSEQ}(L(C_e)) = \emptyset = L(F_{i_0}) = L(F_{f(e)})$.

Hence $f \in \mathcal{F}^{\mathcal{C},\text{REG}}$, and clearly $f \leq_{\text{T}} X$. Thus $\mathcal{F}^{\mathcal{C},\text{REG}} \leq_w X$. The fact that $\mathcal{F}^{\mathcal{C},\text{CFL}} \leq_w X$ follows by (3) of Lemma 2.17. ∎

Theorem 8.2 *Let X be any Σ_2 set such that $\emptyset' \leq_{\text{T}} X$. There exists a class $\mathcal{C}$ of Turing machines such that for any $\mathcal{D} \in \{\text{REG}, \text{CFL}\}$, $\mathcal{F}^{\mathcal{C},\mathcal{D}} \equiv_w X$.*

Proof: Let R be a computable predicate such that for all e,

$$e \in X \iff (\exists x)(\forall y)R(e, x, y).$$

Define C_e to be the Turing machine that does the following:

$$C_e(x) = \begin{cases} 1 & \text{if } (\forall x' \leq x)(\exists y)\neg R(e, x', y); \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Define $\mathcal{C} = \{C_1, C_2, \ldots\}$ as before. We have

$$\begin{aligned} e \in X &\Rightarrow L(C_e) \text{ is finite}, \\ e \notin X &\Rightarrow L(C_e) = \Sigma^*. \end{aligned}$$

Let $f \in \mathcal{F}^{\mathcal{C},\mathcal{D}}$. We have

$$e \in X \iff L(C_e) \text{ is finite} \iff \text{SUBSEQ}(L(C_e)) \text{ is finite} \iff L(D_{f(e)}) \text{ is finite}.$$

Since $D_{f(e)}$ is a CFG or a DFA, we can decide whether or not it recognizes a finite language. Thus $X \leq_{\text{1-tt}} f$.

Conversely, here is an $f \in \mathcal{F}^{\mathcal{C},\mathcal{D}}$ such that $f \leq_{\text{T}} X$: Given e, we ask X whether $L(C_e)$ is finite ($\iff e \in X$), and if so (since $\emptyset' \leq_{\text{T}} X$), we find all the elements of $L(C_e)$ computably in X as follows:

1. $D := \emptyset$.

2. For every string x in lexicographical order, do

   (a) If there is no $y \geq x$ such that $C_e(y) = 1$, then halt and output D.

   (b) If $C_e(x) = 1$, then $D := D \cup \{x\}$.

So if $e \in X$, we let $f(e)$ be the index of a CFG or DFA recognizing SUBSEQ(D); otherwise, we let $f(e)$ be the index of a CFG or DFA recognizing Σ^*. ∎

9 Relation to Recursive and Reverse Mathematics

Nerode's *Recursive Mathematics Program* [20, 8] and Simpson and Friedman's *Reverse Math Program* [23, 24] both attempt to pin down what it means for a proof to be noneffective or nonconstructive. Corollary 3.4 yields results in both of these programs.

Recall Higman's result, which we denote by H:

> If $A \subseteq \Sigma^*$ then SUBSEQ(A) is regular.

In the Recursive Math program one asks if an effective version of a theorem is true. If it is not, then the theorem cannot be proven effectively. Then weaker versions are suggested that may be true. In that spirit, we suggest two effective versions of H.

Effective Version 1: Given an index for a Turing machine that decides a language A, one can effectively produce a DFA for the language SUBSEQ(A).

Effective Version 2: Given an index for a Turing machine that decides a language A, one can effectively produce a Turing machine that decides SUBSEQ(A).

By Corollary 4.10 we have shown that the Effective Versions 1, 2 are false. The question then arises as to how non-effective H is. By Corollary 3.4 H is $\emptyset''$-effective. Another question that arises is whether there are (perhaps weaker) effective versions of H. By Theorems 2.13 and 7.1 the following are true:

Effective Version 3: Given an index for a Turing machine that enumerates a language A, one can produce a Turing machine that enumerates SUBSEQ(A).

Effective Version 4: Given an index for a CFG that decides a language A, one can produce a DFA that decides SUBSEQ(A).

In the Reverse Math program the concern is what axiom system is needed to prove a theorem. RCA_0 is the axiom system that is used to denote that proofs are constructive. (For a definition of RCA_0 see [23].) It is known that if a theorem is false in the model that consists of only recursive sets and functions, then it is not provable in RCA_0. Consider the following statement:

> *There exists a function f that, when given as input an index of a total 0–1 valued Turing Machine M, outputs the code for a DFA for* SUBSEQ($L(M)$).

Theorem 9.1 *The statement above cannot be proven in* RCA_0.

A natural question arises: Which, if any, of the usual proof systems is this statement equivalent to? For example, is it equivalent to the Weak König's Lemma? An anonymous referee has pointed

out that the statement is not equivalent to WKL_0 because any ω-model of the statement would have to include $\emptyset''$ by the results of Section 4, but there is an ω-model of WKL_0 which does not. In fact, the statement is not equivalent to any of the usual systems in reverse mathematics because $\{X : X \leq_{\mathrm{T}} \emptyset''\}$ is an ω-model of the statement, but that set is not an ω-model of any standard system other than RCA_0. A more natural statement to consider from the point of view of reverse mathematics is, "For any A, the set $\mathrm{SUBSEQ}(A)$ is regular," but the results of this paper do not appear to answer that question.

10 Open Problems

The complexity of $\mathcal{F}^{\mathcal{C},\mathcal{D}}$ can be either $\emptyset$ or $\emptyset''$ for natural classes $\mathcal{C}$ and $\mathcal{D}$. The complexity can be either that of any c.e. degree or that of any Σ_2 degree computing $\emptyset'$, for contrived $\mathcal{C}$. What about other Turing degrees?

The functions $\mathrm{NS}^{\mathrm{PU,REG}}$ and $\mathrm{NS}^{\mathrm{PU,CFL}}$ are both in $\mathrm{FQ}(g, X)$ where g can be any monotone increasing unbounded computable function of the output—for example, $\log^* n$. However, it is open to show that these functions cannot be computed with a constant number of queries.

Appendix

We give two proofs of the theorem, "If A is any set then $\mathrm{SUBSEQ}(A)$ is regular." The first one is the original one by Higman; however, we use modern terminology. It uses well quasi-orderings. The second is a new one that avoids using well quasi-orderings.

A Higman's Proof

Definition A.1 A set together with an ordering $(X, \preceq)$ is a *well quasi-ordering* (wqo) if for any sequence $x_1, x_2, \ldots$ of elements of X there exist i, j such that $i < j$ and $x_i \preceq x_j$.

Note A.2 If $(X, \preceq)$ is a wqo, then it is both well-founded and has no infinite antichains.

A.1 Subsets of Well Quasi-Orders That Are Closed Downward

Lemma A.3 *Let $(X, \preceq)$ be a wqo and let $Y \subseteq X$. Assume that Y is closed downward under $\preceq$. Then the following occur.*

1. *There exists a finite set of elements $\{z_1, \ldots, z_k\} \subseteq X$ such that*

$$y \in Y \text{ iff } (\forall i)[z_i \not\preceq y].$$

2. *If $X = \Sigma^*$ and $\preceq$ is the subsequence relation, then Y is regular. (This follows easily from (1) and the fact that $(\Sigma^*, \preceq)$ is a wqo, which we show below in Lemma A.4.)*

Proof: For $x, y \in X$, say that $x \equiv y$ iff $x \preceq y$ and $y \preceq x$. It is clear that $\equiv$ is an equivalence relation. Let $\overline{Y} = X - Y$. Say that x is *minimal* for $\overline{Y}$ if $x \in \overline{Y}$ and, for all $y \in \overline{Y}$, if $y \preceq x$ then $x \preceq y$. Clearly, if x is minimal for $\overline{Y}$ and $x \equiv y$, then y is minimal for $\overline{Y}$. Let $M \subseteq \overline{Y}$ be the set of all minimal elements of $\overline{Y}$, and let $C \subseteq M$ be some set obtained by choosing arbitrarily one element from each $\equiv$-equivalence class in M. The following facts are straightforward:

- C is an antichain, hence C is finite.
- For any $x \in X$, we have $x \in \overline{Y}$ if and only if there is a $z \in C$ with $z \preceq x$.

Setting $C = \{z_1, \ldots, z_k\}$ shows Part (1) of the lemma. ∎

We will show that $(\Sigma^*, \preceq)$ is a wqo, where $\preceq$ is the subsequence ordering on Σ^*. Since SUBSEQ(A) is closed under this ordering, Lemma A.3 will yield that SUBSEQ(A) is regular.

A.2 SUBSEQ is a WQO

Lemma A.4 *Let $\preceq$ be the subsequence ordering on Σ^*. $(\Sigma^*, \preceq)$ is a wqo.*

Proof: Assume not. Then there exist (perhaps many) sequences $x_1, x_2, \ldots$ from Σ^* such that $x_i \not\preceq x_j$ for all $i < j$. We call these *bad sequences*.

We define a particular bad sequence $y_1, y_2, \ldots$ inductively.

1. y_1 is the shortest element that appears as the first element of any bad sequence. (If there is a tie then pick one of them arbitrarily).
2. y_{n+1} is the shortest element that appears as the $(n+1)$st element of any bad sequence that began $y_1, \ldots, y_n$. (If there is a tie then pick one of them arbitrarily).

Let $y_i = y_i'\sigma_i$ where $\sigma_i \in \Sigma$. (Note that none of the y_i are empty since the empty string is a subsequence of every string.)

Let $Y' = \{y_1', y_2', \ldots\}$.

Claim A.5 *$(Y', \preceq)$ is a wqo.*

Proof: Assume not. Then there is a bad sequence of elements from Y'. Let the bad sequence be

$$y'_{k_1}, y'_{k_2}, \ldots.$$

By removing finitely many elements of the sequence, we can assume that

$$k_1 < k_2, k_3, k_4, \ldots.$$

Consider the sequence

$$\text{SEQ} = y_1, y_2, \ldots, y_{k_1-1}, y'_{k_1}, y'_{k_2}, y'_{k_3} \ldots.$$

We show that SEQ is a bad sequence by considering three possible cases.

Case 1: There exists $i < j < k_1$ such that $y_i \preceq y_j$. This cannot happen since $y_1, y_2, \ldots$ is a bad sequence.

Case 2: There exists $i < j$ such that $y'_{k_i} \preceq y'_{k_j}$. This cannot happen since $y'_{k_1}, y'_{k_2}, \ldots$ is a bad sequence by assumption.

Case 3: There exists $i \leq k_1 - 1$ and a $j \geq 1$ such that $y_i \preceq y'_{k_j}$. Then $y_i \preceq y_{k'_j}\sigma_{k_j} = y_{k_j}$. Since $i < k_j$ this cannot happen since $y_1, y_2, \ldots$ is a bad sequence.

Consider the finite sequence

$$y_1, y_2, \ldots, y_{k_1-1}.$$

y_{k_1} is defined as the shortest string such that there is a bad sequence that begins

$$y_1, y_2, \ldots, y_{k_1-1}, y_{k_1}.$$

However, we just showed that

$$\text{SEQ} = y_1, y_2, \ldots, y_{k_1-1}, y'_{k_1}, y'_{k_2}, y'_{k_3} \ldots$$

is a bad sequence. Note that SEQ begins with $y_1, \ldots, y_{k_1-1}$ but its next element is y'_{k_1}, and that $|y'_{k_1}| < |y_{k_1}|$. This contradicts the definition of y_{k_1}. ∎

So we know that Y' is a wqo.

Look at the sequence $y_1, y_2, \ldots$.

Since Σ is finite, there exists a $\sigma \in \Sigma$ such that there are an infinite number of i with $y_i = y'_i\sigma$. Let them be, in order,

$$y_{m_1}, y_{m_2}, \ldots \text{ which is } y'_{m_1}\sigma, y'_{m_2}\sigma, \ldots.$$

Now consider the sequence $y'_{m_1}, y'_{m_2}, \ldots$ of elements from Y'.

Since Y' is a wqo there exists $m_i < m_j$ such that $y'_{m_i} \preceq y'_{m_j}$. Note that $y'_{m_i}\sigma \preceq y'_{m_j}\sigma$, so $y_{m_i} \preceq y_{m_j}$. This contradicts $y_1, y_2, \ldots$ being a bad sequence. ∎

Theorem A.6 (Higman [19]) *If $A \subseteq \Sigma^*$, then* SUBSEQ(A) *is regular.*

Proof: Let $\preceq$ be the subsequence ordering. The language SUBSEQ(A) is closed under $\preceq$. By Lemma A.4, $\preceq$ is a wqo. By Lemma A.3, there exists $z_1, \ldots, z_k \in \Sigma^*$ such that

$$\text{SUBSEQ}(A) = \{x : z_1 \not\preceq x \wedge \cdots \wedge z_k \not\preceq x\}.$$

This set is clearly regular. ∎

B A Proof Without Well Quasi-Orderings

The proof of Theorem A.6 in the last section, derived from Higman's paper [19], makes use of the theory of well quasi-orders. The current proof is completely different and makes no mention of well quasi-orders. Although it is no more constructive than the previous proof, the new proof provides a different insight into the relationships between A and SUBSEQ(A) for various A.

Clearly, SUBSEQ(SUBSEQ(A)) = SUBSEQ(A) for any A, since $\preceq$ is transitive. We'll say that A is $\preceq$-*closed* if A = SUBSEQ(A). So Theorem A.6 is equivalent to the statement that if a language A is $\preceq$-closed then A is regular. The remainder of this appendix is to prove Theorem A.6 directly, without recourse to well quasi-orders.

B.1 Intuition Behind the Proof

In this section, we fix A to be any $\preceq$-closed language.

As a warm-up to the general proof, we'll describe how it works in the special cases of unary languages and binary languages.

The case of unary languages (where $\Sigma = \{ 0 \}$, say) is particularly easy. If $A \subseteq 0^*$ is nonempty and $\preceq$-closed, then either

1. $A = 0^*$, or else
2. A is finite, in which case there is an $n \geq 0$ such that $A = (0 \cup \lambda)^n$.

Case 1 holds if and only if A is infinite. In either case, A is obviously regular.

Now let's look at the binary case, where $\Sigma = \{ 0, 1 \}$. Suppose $A \subseteq \Sigma^*$ is $\preceq$-closed. More generally, suppose that $A \subseteq R$ for some regular expression R. We reduce the problem of showing that A is regular to the problem of showing that some set $B \subseteq S$ is regular, where S is a regular expression that is a "refinement" of R. We do this repeatedly, successively refining the regular expression, until we end up with a regular expression whose language is finite, whence the language in question is also finite and thus regular.

We start off as in the unary case. We first ask whether $A = \Sigma^*$, and, if not, what information that gives us. If $A = \Sigma^*$, then obviously $(01)^i \in A$ for infinitely many $i \geq 0$. A crucial observation is that the converse of this also holds: If $(01)^i \in A$ for infinitely many i, then $A = \Sigma^*$. Why? Because *every* string $x \in \Sigma^*$ is a subsequence of $(01)^i$ for all $i \geq |x|$, and in addition, A is $\preceq$-closed.

If $A = \Sigma^*$, then we're done, so suppose $A \neq \Sigma^*$. Then $(01)^i \in A$ for only finitely many i. This means that there is a fixed finite n such that every string in A has at most n many occurrences of the substring 01. With a bit of thought, it can be seen that this is equivalent to

$$A \subseteq 1^*(0^*1^*)^n0^*.$$

Let $R = 1^*(0^*1^*)^n0^*$ be the regular expression on the right-hand side. R is our first refinement. We can partition R into a finite number of pairwise disjoint regular sets

$$R = R_0 \cup R_1 \cup \cdots \cup R_n,$$

where for each j,

$$R_j = 1^*(0^*011^*)^j0^*$$

is the set of strings x in which 01 occurs *exactly* j times.

It now suffices to show that the set $A_j = A \cap R_j$ is regular for each $j \leq n$. Fix j. R_j is our second refinement. Again we ask whether $A_j = R_j$, and if not, what information that gives us. Obviously, if $A = R_j$, then for infinitely many k we have $1^k(0^k1^k)^j0^k \in A$. The converse of this is also true: For each $x \in R_j$, there are unique natural numbers $k_0, k_1, \ldots, k_{2j+1}$ such that $k_1, \ldots, k_{2j} > 0$ and

$$x = 1^{k_0}0^{k_1} \cdots 1^{k_{2j}}0^{k_{2j+1}}.$$

Clearly, $x \preceq 1^k(0^k1^k)^j0^k$ for any $k \geq \max(k_0, \ldots, k_{2j+1})$. Thus if $1^k(0^k1^k)^j0^k \in A$ for infinitely many k, then x is a subsequence of some string in A, and hence $x \in A$. Since x is an arbitrary element of R_j, this gives $R_j \subseteq A$ and thus $A_j = R_j$.

Now suppose $A_j \neq R_j$. Then $1^k(0^k1^k)^j0^k \in A$ for only finitely many k. This means that there is some fixed finite m such that if x is any string in A_j and $x = 1^{k_0}0^{k_1}\cdots 1^{k_{2j}}0^{k_{2j+1}}$ as above, then $k_i \leq m$ for *at least one* $i \in \{0, \ldots, 2j+1\}$. Therefore, we have

$$A_j \subseteq S_0 \cup S_1 \cup \cdots \cup S_{2j+1},$$

where

$$\begin{aligned}
S_0 &= R_j \cap (1 \cup \lambda)^m 0^* 1^* \cdots 1^* 0^*, \\
S_1 &= R_j \cap 1^* (0 \cup \lambda)^m 1^* \cdots 1^* 0^*, \\
S_2 &= R_j \cap 1^* 0^* (1 \cup \lambda)^m \cdots 1^* 0^*, \\
&\vdots \\
S_{2j} &= R_j \cap 1^* 0^* 1^* \cdots (1 \cup \lambda)^m 0^*, \\
S_{2j+1} &= R_j \cap 1^* 0^* 1^* \cdots 1^* (0 \cup \lambda)^m.
\end{aligned}$$

That is, S_i is the set of all strings $x \in R_j$ where the corresponding k_i is at most m.

So now it suffices to show that the set $A_{j,i} = A_j \cap S_i$ is regular for each $i \leq 2j+1$. Fix i. For example, suppose $i = 2j+1$. The regular expression S_{2j+1} is our third refinement. Then we have

$$S_{2j+1} = 1^*00^*11^* \cdots 11^*(0 \cup \lambda)^m = T_0 \cup T_1 \cup \cdots \cup T_m,$$

where for each $\ell \leq m$,

$$T_\ell = 1^*00^*11^* \cdots 11^*0^\ell.$$

So it suffices to show that the set $A_{j,i,\ell} = A_{j,i} \cap T_\ell$ is regular for each $\ell \leq m$. (The situation is similar for values of i other than $2j+1$.)

Fix $\ell \leq m$. Following our routine, we ask whether $A_{j,i,\ell} = T_\ell$, and, if not, what information do we get. Now just like before, if there are infinitely many k such that $1^k(0^k1^k)^j0^\ell \in A_{j,i,\ell}$, then $A_{j,i,\ell} = T_\ell$ and we are done. Otherwise, we proceed with $A_{j,i,\ell}$ in the same manner as we did above with A_j, except that now we have one fewer (i.e., only $2j+1$ many) k_i values to worry about, so we are making progress.

If we continue on in this way, the number of k_i values we need to check for boundedness will decrease, and we will eventually get to the point where we have some set $A_{j,i,\ell,\ldots} \subseteq S$, where S is a regular expression whose language is *finite*, and so $A_{j,i,\ell,\ldots}$ must be regular. This ends the refinement process.

The general proof for any alphabet, which we now give, uses this idea of successive refinements. An important aspect of the proof is that the refinement relation that we define on our regular expressions is well-founded, and so every succession of refinements must end eventually.

B.2 Preliminaries

We let $\mathbb{N} = \omega = \{0, 1, 2, \ldots\}$ be the set of natural numbers. We will assume WLOG that all symbols are elements of $\mathbb{N}$ and that all alphabets are finite, nonempty subsets of $\mathbb{N}$. We can also assume WLOG that all languages are nonempty.

Definition B.1 For any alphabet $\Sigma = \{ n_1 < \cdots < n_k \}$, we define the *canonical string* for Σ,

$$\sigma_\Sigma := n_1 \cdots n_k,$$

the concatenation of all symbols of Σ in increasing order. If $w \in \Sigma^*$, we define the number

$$\ell_\Sigma(w) := \max\{ n \in \mathbb{N} : (\sigma_\Sigma)^n \preceq w \}.$$

Given a string $x = x_1 \cdots x_m \in \Sigma^*$, where $m \leq n$, we can map each x_i into the i'th copy of σ_Σ within the string $(\sigma_\Sigma)^n$. Thus we have the following observation:

Observation B.2 Every string in Σ^* of length at most n is a subsequence of $(\sigma_\Sigma)^n$. Thus for any string w and $x \in \Sigma^*$, if $|x| \leq \ell_\Sigma(w)$, then $x \preceq w$.

Our regular expressions (regexps) are built from the atomic regexps λ and $a \in \mathbb{N}$ using union, concatenation, and Kleene closure in the standard way (we omit $\emptyset$ as a regexp since all our languages are nonempty). For regexp r, we let $L(r)$ denote the language of r. We consider regexps as syntactic objects, distinct from their corresponding languages. So for regexps r and s, by saying that $r = s$ we mean that r and s are syntactically identical, not just that $L(r) = L(s)$. For any alphabet $\Sigma = \{ n_1, \ldots, n_k \} \subseteq \mathbb{N}$, we let Σ also denote the regexp $n_1 \cup \cdots \cup n_k$ as usual, and in keeping with our view of regexps as syntactic objects, we will hereafter be more precise and say, e.g., "$A \subseteq L(\Sigma^*)$" rather than "$A \subseteq \Sigma^*$."

Definition B.3 A regexp r is *primitive syntactically $\preceq$-closed* (PSC) if r is one of the following two types:

Bounded: $r = a \cup \lambda$ for some $a \in \mathbb{N}$;

Unbounded: $r = \Sigma^*$ for some alphabet Σ.

The *rank* of such an r is defined as

$$\mathrm{rank}(r) := \begin{cases} 0 & \text{if } r \text{ is bounded,} \\ |\Sigma| & \text{if } r = \Sigma^*. \end{cases}$$

Definition B.4 A regexp R is *syntactically $\preceq$-closed* (SC) if $R = r_1 \cdots r_k$, where $k \geq 0$ and each r_i is PSC. For the $k = 0$ case, we define $R := \lambda$ by convention. If w is a string, we define an *R-partition* of w to be a list $\langle w_1, \ldots, w_k \rangle$ of strings such that $w_1 \cdots w_k = w$ and $w_i \in L(r_i)$ for each $1 \leq i \leq k$. We call w_i the ith *component* of the R-partition.

Observation B.5 If regexp R is SC, then $L(R)$ is $\preceq$-closed.

Observation B.6 For SC R and string w, $w \in L(R)$ iff some R-partition of w exists.

Definition B.7 Let $r = \Sigma^*$ be an unbounded PSC regexp. We define $\mathrm{pref}(r)$, the *primitive refinement* of r, as follows: if $\Sigma = \{\, a \,\}$ for some $a \in \mathbb{N}$, then let $\mathrm{pref}(r)$ be the bounded regexp $a \cup \lambda$; otherwise, if $\Sigma = \{\, n_1 < n_2 < \cdots < n_k \,\}$ for some $k \geq 2$, then we let

$$\mathrm{pref}(r) := (\Sigma - \{\, n_1 \,\})^*(\Sigma - \{\, n_2 \,\})^* \cdots (\Sigma - \{\, n_k \,\})^*. \tag{2}$$

In the definition above, note that $\mathrm{pref}(r)$ is SC although not necessarily PSC. Also note that $L((\mathrm{pref}(r))^*) = L(r)$. This leads to the following definition, analogous to Definition B.1:

Definition B.8 Let r be an unbounded PSC regexp, and let $w \in L(r)$ be a string. Define

$$m_r(w) := \min\{\, n \in \mathbb{N} : w \in L((\mathrm{pref}(r))^n) \,\}.$$

There is a nice connection between Definitions B.1 and B.8, given by the following Lemma:

Lemma B.9 *For any unbounded PSC regexp $r = \Sigma^*$ and any string $w \in L(r)$,*

$$m_r(w) = \begin{cases} \ell_\Sigma(w) & \text{if } |\Sigma| = 1, \\ \ell_\Sigma(w) + 1 & \text{if } |\Sigma| \geq 2. \end{cases}$$

Proof: First, if $|\Sigma| = 1$, then $\Sigma = \{\, a \,\}$ for some $a \in \mathbb{N}$, and so $\mathrm{pref}(r) = a \cup \lambda$, and $\sigma_\Sigma = a$. Then clearly,

$$m_r(w) = |w| = \ell_\Sigma(w).$$

Second, we suppose that $\Sigma = \{\, n_1 < \cdots < n_k \,\}$ with $k \geq 2$. We then have $\sigma_\Sigma = n_1 \cdots n_k$, and $\mathrm{pref}(r) = \Sigma_1^* \cdots \Sigma_k^*$ from (2), where we set $\Sigma_i = \Sigma - \{\, n_i \,\}$ for $1 \leq i \leq k$. Let $m = m_r(w)$. We want to show that $m = \ell_\Sigma(w) + 1$. By Definition B.8, we have $w \in L((\mathrm{pref}(r))^m)$, and so by Observation B.6 there is some $(\mathrm{pref}(r))^m$-partition

$$P = \langle w_{11}, \ldots, w_{1k}, w_{21}, \ldots, w_{2k}, \ldots, w_{m1}, \ldots, w_{mk} \rangle$$

of w. Owing to the structure of $(\mathrm{pref}(r))^m$, we know that each $w_{ij} \in L(\Sigma_j^*)$.

Suppose $(\sigma_\Sigma)^\ell \preceq w$ for some $\ell \geq 0$. We show first by the pigeonhole principle that $\ell < m$, and hence $m \geq \ell_\Sigma(w)+1$. Since $(\sigma_\Sigma)^\ell \preceq w$ and $|(\sigma_\Sigma)^\ell| = \ell k$, there is some monotone nondecreasing map $p : \{\, 1, \ldots, \ell k \,\} \to \{\, 1, \ldots, mk \,\}$ such that the t^{th} symbol of $(\sigma_\Sigma)^\ell$ occurs in the $p(t)^{\text{th}}$ component of P. Now we claim that $p(t) \neq t$ for all $1 \leq t \leq \ell k$, which can be seen as follows: writing $t = qk + s$ for some unique q and s where $1 \leq s \leq k$, we have that the t^{th} symbol of $(\sigma_\Sigma)^\ell$ is n_s, but the t^{th} component of P is $w_{(q+1)s} \in L(\Sigma_s^*)$, and $n_s \notin \Sigma_s$. Thus the t^{th} symbol in $(\sigma_\Sigma)^\ell$ does not occur in the t^{th} component of P, and so $t \neq p(t)$. Now it follows from the monotonicity of p that $p(t) > t$ for all t. In particular, $\ell k < p(\ell k) \leq mk$, and so $\ell < m$. This shows that $m \geq \ell_\Sigma(w) + 1$.

We next show that $m \leq \ell_\Sigma(w) + 1$. We build a particular $(\mathrm{pref}(r))^m$-partition

$$P_{\text{greedy}} = \langle w_{11}, \ldots, w_{1k}, w_{21}, \ldots, w_{2k}, \ldots, w_{m1}, \ldots, w_{mk} \rangle$$

of w by a greedy algorithm that defines each successive component of P_{greedy} to be as long as possible, given the previous components. For the purposes of the algorithm, we define, for integers $1 \leq i \leq m$ and $1 \leq j \leq k$,

$$(i, j)' = \begin{cases} (i, j+1) & \text{if } j < k, \\ (i+1, 1) & \text{otherwise.} \end{cases}$$

This is the successor operation in the lexicographical ordering on the pairs (i, j) with $1 \leq j \leq k$ and $1 \leq i \leq m$:

$$(i_1, j_1) < (i_2, j_2) \text{ if either } i_1 < i_2 \text{ or } i_1 = i_2 \text{ and } j_1 < j_2.$$

The idea is that we "match" the longest prefix of w that we can by the regexp Σ_j^* for various j.

$(i, j) \leftarrow (1, 1)$.
While $i \leq m$ do
    Let w_{ij} be the longest prefix of w that is in Σ_j^*.
    Remove prefix w_{ij} from w.
    $(i, j) \leftarrow (i, j)'$.
End while

Since *some* $(\text{pref}(r))^m$-partition of w exists, this algorithm will clearly also produce a $(\text{pref}(r))^m$-partition of w, i.e., the while-loop terminates with $w = \lambda$. Furthermore, w does not become λ until the end of the $(m, 1)$-iteration of the loop at the earliest; otherwise, the algorithm would produce a $(\text{pref}(r))^{m-1}$-partition of w, contradicting the minimality of m. Finally, for all (i, j) lexicographically between $(1, 1)$ and $(m - 1, k)$ inclusive, just after we remove w_{ij}, the remaining string w starts with n_j. This follows immediately from the greediness (maximum length) of the choice of w_{ij}. This implies that the next component after w_{ij} starts with n_j. Therefore, we have σ_Σ is a subsequence of each of the $m - 1$ strings

$$w_{12}w_{13}w_{14}\cdots w_{21}, \quad w_{22}w_{23}w_{24}\cdots w_{31}, \quad \ldots, \quad w_{(m-1)2}w_{(m-1)3}w_{(m-1)4}\cdots w_{m1},$$

and so $(\sigma_\Sigma)^{m-1} \preceq w$, which proves that $m \leq \ell_\Sigma(w) + 1$. ∎

Definition B.10 Let $R = r_1 \cdots r_k$ and S be two SC regexps, where each r_i is PSC. We say that S is a *one-step refinement* of R if S results from either

- removing some bounded r_i from R, or
- replacing some unbounded r_i in R by $(\text{pref}(r_i))^n$ for some $n \in \mathbb{N}$.

We say that S is a *refinement* of R (and write $S < R$) if S results from R through a sequence of one or more one-step refinements.

One may note that if $S < R$, then $L(S) \subseteq L(R)$, although it is not important to the main proof.

Lemma B.11 *The relation $<$ of Definition B.10 is a well-founded partial order on the set of SC regexps (of height at most ω^ω).*

Proof: Let $R = r_1 \cdots r_k$ be an SC regexp, and let $e_1 \geq e_2 \geq \cdots \geq e_k$ be the ranks of all the r_i, arranged in nonincreasing order, counting duplicates. Define the ordinal

$$\text{ord}(R) := \omega^{e_1} + \omega^{e_2} + \cdots + \omega^{e_k},$$

which is in Cantor Normal Form and is always less than ω^ω. If $R = \lambda$, then $\mathrm{ord}(R) := 0$ by convention. Let S be an SC regexp. Then it is clear that $S < R$ implies $\mathrm{ord}(S) < \mathrm{ord}(R)$, because the ord of any one-step refinement of R results from either removing some addend $\omega^0 = 1$ or replacing some addend ω^e for some positive e (the rightmost with exponent e) in the ordinal sum of $\mathrm{ord}(R)$ with the ordinal $\omega^{e-1} \cdot n$, for some $n < \omega$, resulting in a strictly smaller ordinal. From this the lemma follows. ∎

B.3 Main Proofs

The following lemma is key to proving Theorem A.6.

Lemma B.12 (Key Lemma) *Let $R = r_1 \cdots r_k$ be a SC regexp where at least one of the r_i is unbounded. Suppose $A \subseteq L(R)$ is $\preceq$-closed. Then either*

1. *$A = L(R)$ or*
2. *there exist refinements $S_1, \ldots, S_k < R$ such that $A \subseteq \bigcup_{i=1}^{k} L(S_i)$.*

Before proving Lemma B.12, we use it to prove the next lemma, from which Theorem A.6 follows immediately.

Lemma B.13 *Let $R = r_1 \cdots r_k$ be any SC regexp. If $A \subseteq L(R)$ and A is $\preceq$-closed, then A is regular.*

Proof: We proceed by induction on the refinement relation on SC regexps. Fix $R = r_1 \cdots r_k$, and suppose that $A \subseteq L(R)$ is $\preceq$-closed. If all of the r_i are bounded, then $L(R)$ is finite, and hence A is regular. Now assume that at least one r_i is unbounded and that the statement of the lemma holds for all $S < R$. If $A = L(R)$, then A is certainly regular, since R is a regexp. If $A \neq L(R)$, then by Lemma B.12 there are $S_1, \ldots, S_k < R$ with $A \subseteq \bigcup_{i=1}^{k} L(S_i)$. Each set $A \cap L(S_i)$ is $\preceq$-closed (being the intersection of two $\preceq$-closed languages) and hence regular by the inductive hypothesis. But then,

$$A = A \cap \bigcup_{i=1}^{k} L(S_i) = \bigcup_{i=1}^{k} (A \cap L(S_i)),$$

and so A is regular. ∎

Corollary B.14 (equivalent to Theorem A.6) *Let $A \subseteq L(\Sigma^*)$ be any language. If A is $\preceq$-closed, then A is regular.*

Proof: Apply Lemma B.13 with $R = \Sigma^*$. ∎

Proof of Lemma B.12: Fix $R = r_1 \cdots r_k$ and A as in the statement of the lemma. Whether Case 1 or Case 2 holds hinges on whether or not a certain quantity associated with each string in $L(R)$ is unbounded when taken over all strings in A.

For any string $w \in L(R)$ and any R-partition $P = \langle w_1, \ldots, w_k \rangle$ of w, define

$$M_P^{\mathrm{bd}}(w) := \min_{i:\ r_i \text{ is bounded}} |w_i|, \tag{3}$$

and define

$$M_P^{\text{unbd}}(w) := \min_{i:\ r_i \text{ is unbounded}} m_{r_i}(w_i). \tag{4}$$

In (3), for any bounded r_i, we have $w_i \in L(r_i)$ and thus $|w_i| \in \{\, 0, 1 \,\}$. (If there is no bounded r_i, we'll take the minimum to be 1 by default.) Thus we have

$$M_P^{\text{bd}}(w) = \begin{cases} 0 & \text{if } (\exists i)[r_i \text{ is bounded and } w_i = \lambda], \\ 1 & \text{otherwise.} \end{cases}$$

Now define

$$M(w) := \max_{P:\ P \text{ is an } R\text{-partition of } w} M_P^{\text{bd}}(w) \cdot M_P^{\text{unbd}}(w), \tag{5}$$

and note that

$$M(w) = \begin{cases} \max_P M_P^{\text{unbd}}(w) & \text{if } (\exists P)[M_P^{\text{bd}}(w) = 1], \\ 0 & \text{otherwise.} \end{cases}$$

Finally, define

$$M(A) := \{\, M(w) : w \in A \,\}. \tag{6}$$

We will show that if $M(A)$ is infinite, then Case 1 of the lemma holds. Otherwise, Case 2 holds.

Suppose that $M(A)$ is infinite. We already have $A \subseteq L(R)$ by assumption, so we show that $L(R) \subseteq A$. Let $x \in L(R)$ be arbitrary. Then since $M(A)$ is infinite, there is a $w \in A$ such that $|x| < M(w)$. For this w there is an R-partition $P = \langle w_1, \ldots, w_k \rangle$ of w such that $M_P^{\text{bd}}(w) = 1$ and $M_P^{\text{unbd}}(w) > |x|$. Let $\langle x_1, \ldots, x_k \rangle$ be any R-partition of x. We then have the following facts:

- For each i where r_i is bounded, we have $|w_i| = 1$, since $M_P^{\text{bd}}(w) = 1$. Hence, $x_i \preceq w_i$ because both x_i and w_i are in $L(r_i) = L(a \cup \lambda)$ for some $a \in \mathbb{N}$.
- For each i where r_i is unbounded, we have $|x| \leq m_{r_i}(w_i) - 1$, because $|x| < M_P^{\text{unbd}}(w)$. Writing $r_i = \Gamma^*$ for some alphabet Γ, we see that $m_{r_i}(w_i) - 1 \leq \ell_\Gamma(w_i)$ by Lemma B.9. We then have $|x_i| \leq |x| \leq \ell_\Gamma(w_i)$, and so $x_i \preceq w_i$ by Observation B.2.

So in any case, we have $x_i \preceq w_i$ for all $1 \leq i \leq k$. Thus $x \preceq w$. Since $w \in A$ and A is $\preceq$-closed, we have $x \in A$. Since $x \in L(R)$ was arbitrary, this proves that $A = L(R)$, which is Case 1 of the lemma.

Now suppose that $M(A)$ is finite. This means that there is a finite bound B such that $M(w) \leq B$ for all $w \in A$. So for any $w \in A$ and any R-partition $P = \langle w_1, \ldots, w_k \rangle$ of w, either $M_P^{\text{bd}}(w) = 0$ or $M_P^{\text{unbd}}(w) \leq B$. Suppose first that $M_P^{\text{bd}}(w) = 0$. Then there is some i where r_i is bounded and $w_i = \lambda$. Let S_i be the one-step refinement of R obtained by removing r_i from R. Then clearly, $w \in L(S_i)$. Now suppose $M_P^{\text{unbd}}(w) \leq B$, so that there is some unbounded r_j such that $m_{r_j}(w_j) \leq B$. This means that $w_j \in L((\text{pref}(r_j))^B)$ by Definition B.8. Let S_j be the one-step refinement obtained from R by replacing r_j with $(\text{pref}(r_j))^B$. Then clearly again, $w \in L(S_j)$. In general, we define, for all $1 \leq i \leq k$,

$$S_i = \begin{cases} r_1 \cdots r_{i-1} r_{i+1} \cdots r_k & \text{if } r_i \text{ is bounded,} \\ r_1 \cdots r_{i-1} (\text{pref}(r_i))^B r_{i+1} \cdots r_k & \text{if } r_i \text{ is unbounded.} \end{cases}$$

We have shown that there is always an i for which $w \in L(S_i)$. Since $w \in A$ was arbitrary, Case 2 of the lemma holds. ∎

Remark. We know that Theorem A.6 cannot be made effective, so it is worth while to pinpoint the noncomputability in the current proof. If we are trying to find a regular expression for SUBSEQ(A), the crucial part of the task is in determining whether or not the set $M(A)$ of (6) is finite, and if so, finding an upper bound B for it. (The rest of the task is effective.) The function $w \mapsto M(w)$ depends implicitly on the underlying SC regexp R, and is computable uniformly in w and R. This means, for example, that if we have a c.e. index for A, we can effectively find a c.e. index for $M(A)$, and so determining whether or not $M(A)$ is finite can be done with a single query to $\emptyset''$. (If $M(A)$ is finite, then finding B can be done by making multiple queries to $\emptyset'$.) We may have to repeat this whole process some finite number of times to find refinements of various regular expressions. We know from Corollary 4.10 that we cannot improve on this.

Acknowledgments

We thank Walid Gomma for proofreading and helpful comments beyond the call of duty. We also thank two anonymous referees for suggestions that strengthened results in Sections 4, 8, and 9.

References

[1] R. Beigel. *Query-Limited Reducibilities*. Ph.D. thesis, Stanford University, 1987. Also available as Report No. STAN-CS-88–1221.

[2] R. Beigel. Unbounded searching algorithms. *SIAM Journal on Computing*, 19(3):522–537, June 1990.

[3] R. Beigel and W. Gasarch. On the complexity of finding the chromatic number of a recursive graph II: The unbounded case. *Annals of Pure and Applied Logic*, 45(3):227–246, Dec. 1989.

[4] R. Beigel, W. Gasarch, J. Gill, and J. Owings. Terse, Superterse, and Verbose sets. *Information and Computation*, 103(1):68–85, Mar. 1993.

[5] J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Inf. Process. Lett.*, 5(3):82–87, Aug. 1976.

[6] L. Blum and M. Blum. Towards a mathematical theory of inductive inference. *Information and Computation*, 28:125–155, 1975.

[7] J. Case and C. H. Smith. Comparison of identification criteria for machine inductive inference. *Theoretical Computer Science*, 25:193–220, 1983.

[8] Y. L. Ershov, S. S. Goncharov, A. Nerode, and J. B. Remmel, editors. *Handbook of Recursive Mathematics*. Elsevier North-Holland, Inc., New York, 1998. Comes in two volumes. Volume 1 is Recursive Model Theory, Volume 2 is Recursive Algebra, Analysis, and Combinatorics.

[9] A. F. Fahmy and R. Roos. Efficient learning of real time one-counter automata. In *Proceedings of the 6th International Workshop on Algorithmic Learning Theory ALT'95*. LNCS 997:25–40, Springer-Verlag, 1995.

[10] S. Fenner and W. Gasarch. The complexity of learning SUBSEQ(A). In *Proceedings of the 17th International Conference on Algorithmic Learning Theory*. LNAI 4264:109–123, Springer-Verlag, 2006. Journal version in preparation.

[11] P. C. Fischer. Turing machines with restricted memory access. *Information and Control*, 9:364–379, 1966.

[12] P. C. Fischer, A. R. Meyer, and A. L. Rosenberg. Counter machines and counter languages. *Mathematical Systems Theory*, 2(3):265–283, 1968.

[13] W. Gasarch and K. S. Guimarães. Binary search and recursive graph problems. *Theoretical Computer Science*, 181:119–139, 1997.

[14] W. Gasarch and G. Martin. *Bounded Queries in Recursion Theory*. Progress in Computer Science and Applied Logic. Birkhäuser, Boston, 1999.

[15] E. M. Gold. Language identification in the limit. *Information and Computation*, 10(10):447–474, 1967.

[16] J. Hartmanis. Context-free languages and Turing machine computations. In *Proceedings of Symposia in Applied Mathematics*. Mathematical aspects of computer science. (J. T. Schwartz, ed.) 19:42–51, American Mathematical Society, Providence, RI, 1967.

[17] J. Hartmanis. On the succinctness of different representations of languages. *SIAM Journal on Computing*, 9, 1980.

[18] L. Hay. On the recursion-theoretic complexity of relative succinctness of representations of languages. *Information and Control*, 52, 1982.

[19] A. G. Higman. Ordering by divisibility in abstract algebras. *Proc. of the London Math Society*, 3:326–336, 1952.

[20] G. Metakides and A. Nerode. Effective content of field theory. *Annals of Mathematical Logic*, 17:289–320, 1979.

[21] M. L. Minsky. Recursive unsolvability of Post's Problem of "Tag." *Annals of Mathematics*, 74(3):437–453, 1961.

[22] A. A. Muchnik. On strong and weak reducibility of algorithmic problems. *Sibirskii Matematicheskii Zhurnal*, 4:1328–1341, 1963. In Russian.

[23] S. G. Simpson. *Subsystems of Second Order Arithmetic*. Springer-Verlag, 1999. Perspectives in mathematical logic series.

[24] S. G. Simpson, editor. *Reverse Mathematics 2001*. Association of Symbolic Logic, 2005. Perspectives in mathematical logic series.

[25] M. Sipser. *Introduction to the Theory of Computation (2nd Ed.)*. Course Technology, Inc., 2005.

[26] R. I. Soare. *Recursively Enumerable Sets and Degrees*. Perspectives in Mathematical Logic. Springer-Verlag, Berlin, 1987.

[27] R. I. Soare. Computability and recursion. *Bulletin of Symbolic Logic*, 27, 1996.

[28] L. G. Valiant and M. S. Paterson. Deterministic one-counter automata. *Journal of Computer and System Sciences*, 10:340–350, 1975.

[29] J. van Leeuwen. Effective constructions in well-partially-ordered free monoids. *Discrete Mathematics*, 21:237–252, 1978.