Computing and evolving variants of computational depth

James Irvin Lathrop
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/rtd
Part of the Computer Sciences Commons

Recommended Citation
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6” x 9” black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Computing and evolving variants of computational depth

by

James Irvin Lathrop

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirement for the degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science
Major Professor: Jack H. Lutz

Iowa State University
Ames, Iowa
1997

Copyright © James Irvin Lathrop, 1997. All rights reserved.
This is to certify that the doctoral dissertation of

James Irvin Lathrop

has met the dissertation requirements of Iowa State University

Signature was redacted for privacy.

Committee Member

Signature was redacted for privacy.

Committee Member

Signature was redacted for privacy.

Committee Member

Signature was redacted for privacy.

Committee Member

Signature was redacted for privacy.

Major Professor

Signature was redacted for privacy.

For the Major Program

Signature was redacted for privacy.

For the Graduate College
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>PRELIMINARIES</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>RECURSIVE COMPUTATIONAL DEPTH</td>
<td>60</td>
</tr>
<tr>
<td>4</td>
<td>COMPRESSION DEPTH</td>
<td>110</td>
</tr>
<tr>
<td>5</td>
<td>GENETIC ALGORITHMS AND COMPRESSION DEPTH</td>
<td>160</td>
</tr>
<tr>
<td>6</td>
<td>CONCLUSION</td>
<td>178</td>
</tr>
<tr>
<td></td>
<td>BIBLIOGRAPHY</td>
<td>181</td>
</tr>
</tbody>
</table>
CHAPTER 1. INTRODUCTION

Measures of the complexities of objects are widely used in both theory and applications in order to model, predict, and classify objects. Information theory gives us several methods for measuring the information content of objects. The two most widely used information measures – entropy and algorithmic information (Kolmogorov complexity) – are used to solve problems in several scientific fields, including data compression, data prediction, image processing, and computational complexity.

Even though these two measures of information content are invaluable in many areas of research, they do not capture the essence of what many people perceive to be complex. The canonical example of this phenomenon is an object composed entirely of random information. Under these widely used measures of information content, objects composed of random information have maximal information content; however, these objects lack the intricate structure, found in complex objects such as DNA sequences or reference books, that allows its information to be used efficiently.

Many researchers have defined measures of complexity that attempt to capture specific types of organization or information. Typical proposals for such a measure are specific and designed to measure an object’s complexity under a restricted
model or domain. (See [16, 28, 29, 69, 67] for example.) However, Bennett [7, 8] has defined a complexity measure based on programs for universal Turing machines that does capture the desired complexity criteria and is universal for all objects that can be digitally encoded.

In the 1980's, Bennett introduced computational depth as a formal measure of the amount of computational history that is evident in an object's structure. In particular, Bennett identified the classes of weakly deep and strongly deep for (infinite) binary sequences, and showed that the halting problem is strongly deep. Juedes, Lathrop, and Lutz [38] subsequently extended this result by defining the class of weakly useful sequences, and proving that every weakly useful sequence is strongly deep.

Under Bennett's definition, the computational depth [7, 8] of a binary data string is roughly the amount of time required to generate the string from a description of the string that is of nearly minimal length. (A significance parameter $s$ is used to define "nearly minimal" in chapter 2.) A description of an object contains all the essential information required to algorithmically reproduce the object; a minimal description contains no redundancy or structure. (If a description did contain structure, this structure could be used to compress it.) If an object cannot be quickly derived from its minimal description, then the object is organized (contains redundancy) in an essential way. This organization is quanti-
fied by the amount of time required to generate the string from its nearly minimal
description. Thus, computational depth is the amount of organization embedded
in a string by a computation.

For (infinite, binary) sequences, Bennett [7, 8] introduced two interesting depth
conditions, strong depth and weak depth. A sequence $S$ is strongly deep if, for
every computable time bound $t : \mathbb{N} \to \mathbb{N}$ and every constant $c \in \mathbb{N}$, for all but
finitely many $n \in \mathbb{N}$, the $n$-bit prefix $S[0..n-1]$ of $S$ has depth greater than $t(n)$
at significance level $c$. If we regard a description $\pi$ from which $S[0..n-1]$ can be
derived in at most $t(n)$ computation steps as a $t(n)$-compression of $S[0..n-1]$,
then this says that, for all computable time bounds $t$ and constants $c$, for all but
finitely many $n$, every $t(n)$-compression of $S[0..n-1]$ is itself compressible by
more than $c$ bits. Thus a sequence is strongly deep if no computable time bound
suffices to compress infinitely many of its prefixes to within a constant number of
bits of the optimal compression.

To put the matter more fancifully, no matter how (computably) much time is
spent looking for inner structure (i.e., basis for compression) in a strongly deep
sequence, an unbounded quantity of such inner structure remains undiscovered.
A strongly deep sequence is thus analogous to a great work of literature for which
no number of readings suffices to exhaust its value.

Computational depth appears to be an ideal complexity measure for determi-
ing whether an object contains intricate structure. For example, Bennett [7, 8] notes that objects with simple structures such as strings consisting of all zeros or strings composed of random bits are not strongly deep. However, Bennett also shows that the characteristic sequence of the halting problem is strongly deep, reflecting its very intricate and useful structure. Further evidence that computational depth measures structural organization is given by Juedes, Lathrop, and Lutz [38] who have shown that, if an object can be used to speed up the computations of a significant collection of recursive sequences, then that object must be strongly deep.

Unfortunately, an essential feature in the definition of computational depth is Kolmogorov complexity, an uncomputable quantity. This makes computational depth itself uncomputable, and although valuable in theoretical contexts, its non-computability renders it useless for actual complexity measurements. In this thesis, we explore two alternative computable measures of complexities motivated by computational depth. First, by adding a variable to Bennett’s notion of computational depth, we introduce recursive computational depth, a parameterized version of computational depth that may be used to define a computable version of computational depth. Second, we utilize efficient compression algorithms to define compression depth [45], a quickly computable quantity that gives a complexity measure with properties analogous to computational depth.
Key to results in this thesis is the notion of recursive computational depth. Simply put, the recursive computational depth of a string is the amount of time required to generate the string from a nearly minimal time-bounded description of the string. (Here, the time-bounded minimal description refers to the shortest description that produces the object within the specified time bound. A latency parameter $l$ is used to define the time bound and, as with computational depth, a significance parameter $s$ is used to define “nearly minimal.” Precise definitions appear in chapter 3.) Thus, we use the parameter $l$ to restrict the definition of “minimal size.” Since the smallest program to compute a string $x$ in time $l$ is computable, it follows that for all $l, s \in \mathbb{N}$, the recursive computational depth of a string with significance parameter $s$ and latency parameter $l$ is computable.

In the terminology used above to describe strong depth, a sequence $S$ is recursively strongly deep (briefly, rec-strongly deep) if, for every computable time bound $t$ and constant $c$, there exists a computable time bound $l$ such that, for all but finitely many $n$, every $t(n)$-compression of $S[0..n - 1]$ is itself $l(n)$-compressible by more than $c$ bits. It is the existence of this computable time bound $l$ that distinguishes rec-strong depth from strong depth. Returning to the more fanciful language used earlier, no matter how (computably) much time is spent looking for inner structure in a rec-strongly deep sequence, and no matter how much additional structure (any constant number of bits) one wishes to find, there is always
a greater (computable) amount of time that suffices to find that much more structure. A rec-strongly deep sequence is thus analogous to a great work of literature with the property that, no matter how many times it has been read, there is a greater number of readings from which one can derive significantly more value.

Using Bennett's terminology, a rec-strongly deep sequence $S$ shows evidence of a nontrivial causal (computational) history in the constructive, incremental sense that every explanation of $S$ that can be realized by an effective process of computable duration is significantly less plausible than some other explanation of $S$ that can also be realized by an effective process of some greater computable duration. In contrast, a sequence that is strongly deep but not rec-strongly deep has an explanation that (i) can be realized by an effective process of computable duration, and (ii) is as plausible as any other explanation that can be realized by an effective process of computable duration. Although such a sequence does have a more plausible explanation, there is no constructive evidence of this fact.

Using recursive computational depth, we investigate refinements of Bennett's notions of weak and strong depth, called recursively weak depth (introduced by Fenner, Lutz and Mayordomo [19]) and recursively strong depth (introduced here). These refinements naturally capture Bennett's idea that deep objects are those which "contain internal evidence of a nontrivial causal history." The fundamental properties of recursive computational depth are developed, and it is shown
that the recursively weakly (respectively, strongly) deep sequences form a proper subclass of the class of weakly (respectively, strongly) deep sequences. The above-mentioned theorem of Juedes, Lathrop, and Lutz is then strengthened by proving that every weakly useful sequence is recursively strongly deep. It follows from these results that not every strongly deep sequence is weakly useful, thereby answering a question posed by Juedes [37].

Even though recursive computational depth is computable, the computation of this quantity is exponential in the length of the string and the latency. The time required to compute this quantity is large, even for very small latencies. Hence, recursive computational depth is not feasibly computable and thus is not more useful than computational depth for the purpose of actually computing a complexity measure for real data.

The definition of computational depth uses time-bounded Kolmogorov complexity to measure the time required to compute $x$ from its smallest representation. However, $K(x)$ (the smallest description of $x$) is not computable, and any reasonable approximation to $K(x)$ (computing the time-bounded minimal description for some large time-bound) requires so much computation time as to render it unusable. As noted above, recursive computational depth is also infeasible. One way to proceed is to consider the "reverse" of time-bounded Kolmogorov complexity by formulating a complexity measure based on the time required to compress $x$
to its shortest description. Using compression as the basis for a depth complexity measurement gives the following approach to defining the compression depth of a string $x$.

Compression depth, introduced by Lathrop [45] is a feasibly computable depth measure based on the Lempel-Ziv compression algorithm. Since the computation of compression depth is efficient, it is useful for making actual measurements of the complexities of specific objects. Lathrop [45] demonstrates the usefulness of this notion by investigating the compression depth of cellular automata in classes that have been defined and investigated by Wolfram [75] and Langton [44].

The compression depth of a string $x$ is the amount of resource required by a compression algorithm to compress $x$ to within very few bits of the shortest compression achievable by that algorithm. (In the terminology of Bennett [7, 8], a string with high compression depth is said to be cryptic.) Unlike Bennett's notion of computational depth, the resource is not required to be time, but may be any resource whose restriction impairs the performance of the compression algorithm, thereby parameterizing the amount of compression in terms of the resource. Intuitively, a string has a large compression depth if, as more resources are allowed, the compression algorithm utilizes these resources to find more subtle redundancy and further compresses the string. Lathrop [45] uses the well-known Lempel-Ziv (LZ) compression algorithm and restricts the size of the dictionary to achieve a
parameterization of the Lempel-Ziv compression algorithm. By computing the compression of a string $x$ at many different resource levels (dictionary sizes), we thus compute an analogy of computational depth that may be used to measure the “organizational” complexity of a string $x$.

Lathrop [45] demonstrates two applications of compression depth, using cellular automata as a testing ground. First, his experiments show that Wolfram Class I and Class II cellular automata (automata that give rise to simple structures) are shallow, having low compression depth. Wolfram class III cellular automata (automata that give rise to “random” structures) are also shallow. Wolfram class IV automata produce patterns that are complex to the human eye and appear to evolve a rich structure. Lathrop’s experiments show that many Class IV automata also have large compression depth, confirming that compression depth appears to measure some type of structure or complexity found in these types of cellular automata. Second, Lathrop [45] uses compression depth to analyze behavior of cellular automata under Langton’s $\lambda$ parameter. Results from this analysis show that compression depth (organization or complexity) does not necessarily arise at a critical value for $\lambda$, but rather arises in a range of values for $\lambda$. These studies provide experimental evidence that compression depth is useful for analyzing the structural complexity of data.

In this thesis, we formalize and extend this work with compression depth.
Using the parameterized version of the Lempel-Ziv compression, we show that analogues of the main properties of computational depth also hold for compression depth. Strongly (compression) deep sequences are defined using Lempel-Ziv compression depth and it is shown that Lempel-Ziv random sequences are shallow in the sense of this definition. We also show that Lempel-Ziv randomness is stronger (more random) than normality (a finite-state notion of randomness), and that simple sequences are also shallow.

Also in this thesis, we extend Lathrop's [45] experimental work by examining the role of compression depth in genetic algorithms. In this situation, we examine the complexities of "genes" during their "evolution." Since a genetic algorithm can be viewed as a computational process, and the initial genes are random, it is possible that the compression complexity rises as more fit (more evolved) genes replace less fit (less evolved) genes. This gives a basis for the experimental work in this thesis.

Darwin [17] studied the relationship between natural selection and complexity of organisms, eventually linking less complex older organisms to more complex recent organisms through the process of evolution. Natural selection is also used in computer science to search for better (more fit) answers embedded in large search spaces. Using genetic algorithms and compression depth, we experimentally show that "natural selection" in computer science also produces complexity and
organization as the *population* ages.

Genetic algorithms are commonly used to solve optimization problems where other search algorithms fail. First introduced by Holland [31, 32], genetic algorithms are an important tool for finding solutions to a variety of computational problems. Applications of genetic algorithms are now described in literally thousands of research papers, e.g., [60, 27, 23, 18, 26, 25]. A detailed list of applications and references can be found in [24].

Basically, a genetic algorithm operates by initially producing a random sampling (*population*) of possible solutions (*genes*) and then, using *genetic operators*. forms new possible solutions in the population. Once this is completed, a *fitness* function is used to remove weaker (less fit and farther from the solution) genes from the population. This procedure is iterated, producing a new generation of genes in the population each time, until a solution is found, presumably with perfect fitness.

The key to this process is the genetic operators used to generate new genes in the population. Taken from biology, crossover and mutation provide the means by which the solution space is searched. In this thesis, we use *crossover* and *mutation* genetic operators together with a genetic algorithm to explore how genetic algorithms can form complex objects that are deep. Since genetic algorithms work on the same principles as natural selection, the genetic algorithm itself corresponds
to a form of evolution. We propose that (depending on the fitness function) the computation performed by a genetic algorithm (i.e. the evolution) forms complex objects.

In this thesis we use genetic algorithms combined with finite-state machines to play iterated prisoner's dilemma. Since the fitness function in this game does not directly attempt to evolve solutions with high compression depth (the fitness function is based on how well a player does playing a game against all other players in the population, see below), we conclude that solutions to hard problems may require subtle organization, i.e., high compression depth.

Prisoner's dilemma, first introduced by Flood and Dresher [20], is a two-player game commonly used to model a variety of biological and social interactions. (See [57] for example). Iterated prisoner's dilemma, a repeated version of the game, was introduced by Axelrod [4, 5] to capture the players' ability to learn about their opponents. In this thesis we use compression depth to measure the complexities of players' behaviors by computing the compression depths of their behaviors as they are evolved using a genetic algorithm. We show that for finite-state control strategies, the strategies evolved over the course of time (measured by the number of generations in the genetic algorithm) produce behaviors that are increasingly complex in terms of compression depth. This leads to the conclusion that evolution (in genetic algorithms) can produce subtle organization and intricate complexity
in genes in order to achieve the overall goal of the genetic algorithm.

This thesis is divided into six chapters. Chapter 2 defines basic notations and definitions used throughout this dissertation, and reviews the basic ideas of Kolmogorov complexity, measure, category, and computational depth. Chapter 3, which reports joint work with Lutz [46], introduces and investigates recursive computational depth. Chapter 4, part of which is joint work with Strauss [47], may be read independently of chapter 3. In this chapter, we present a formalization of compression depth using the Lempel-Ziv algorithm. Strongly (compression) deep sequences are defined in analogy with computational depth. By defining a notion of Lempel-Ziv randomness, results analogous to results found in [8] are proven. Chapter 5 reviews iterated prisoner's dilemma and gives experimental evidence that genetic algorithms produce deep strategies for this game. Finally, Chapter 6 presents conclusions drawn from the results of this dissertation and gives some possible directions for future research.
CHAPTER 2. PRELIMINARIES

This chapter presents common notations and definitions used throughout this dissertation, and presents a brief introduction to randomness, computational depth, and the tools needed to read chapters 3, 4 and 5. Other notations and definitions used in this dissertation are presented where they first appear.

2.1 Notation and Conventions

A (binary) string, usually represented by a lower case character, is a finite sequence of symbols from the set \{0, 1\}. The set of all strings over \{0, 1\} is denoted by \{0, 1\}^*. For a string \(x \in \{0, 1\}^*\), the length of \(x\) is denoted by \(|x|\). The empty string, \(\lambda\), has length 0.

A (binary) sequence, usually represented by an upper case character, is an infinite sequence of symbols from the set \{0, 1\}. The set of all sequences over \{0, 1\} is denoted by \(\mathcal{C}\). The length of a sequence \(S\) is \(\infty\).

For \(x\) a string and \(y\) a string or sequence, the string (or sequence) \(x \cdot y\) denotes the concatenation of the string \(x\) with the string or sequence \(y\). \(x^n\) denotes the \(n\)-fold concatenation of the string \(x\) with itself, and \(x^\infty\) is the sequence consisting of the infinite concatenation of \(x\) with itself. For \(x\) a string or sequence, the substring \(x[i..j]\) denotes a string consisting of the \(i^{th}\) through \(j^{th}\) bits of the string.
or sequence \( x \), where \( 0 \leq i \leq j \leq |x| - 1 \). The \( i \text{th} \) bit of \( x \) is \( x[i] = x[i..i] \).

We let \( \prec \) be the standard (total) ordering of binary strings, first by length and then lexicographically. Thus \( \lambda \prec 0 \prec 1 \prec 00 \prec 01 \prec \cdots \). The string \( s_n \) is the \( n \text{th} \) string in the lexicographic ordering of all strings \( x \in \{0,1\}^* \). For example, 
\[
\begin{align*}
    s_0 &= \lambda, \\
    s_1 &= 0, \\
    s_2 &= 1, \\
    s_3 &= 00, \\
    s_4 &= 01,
\end{align*}
\]
even.

A string \( x \) is a prefix of a string or sequence \( y \), denoted by \( x \subseteq y \), if and only if \( |x| \leq |y| \) and \( x = y[0..|x| - 1] \). A string \( x \) is a proper prefix of \( y \), denoted by \( x \subset y \), if and only if \( x \subseteq y \) and \( |x| < |y| \).

A prefix code (also called an instantaneous code) is a set of strings such that no string is a proper prefix of any other string in the set. The probability of a prefix code \( I \subseteq \{0,1\}^* \) is
\[
    \Pr(I) = \sum_{x \in I} 2^{-|x|}.
\]

The self-delimited version of a string \( x \in \{0,1\}^* \), denoted \( sd(x) \), is defined as
\[
    sd(x) = bd(x)01,
\]
where \( bd(x) \) is the string \( x \) with every bit doubled. (E.g., \( bd(010) = 001100 \) and \( sd(010) = 00110001 \).)
For $x, y \in \{0, 1\}^*$ the self-delimited pairing of $x$ and $y$, denoted $< x, y >$, is

$$< x, y > = sd(x) \cdot sd(y) \cdot 01,$$

and in general, the $n$-self-delimited sequencing of $n$ strings, $x_0, x_1, \ldots, x_{n-1}$, denoted $(x_0, x_1, \ldots, x_{n-1})$, is

$$(x_0, x_1, \ldots, x_{n-1}) = sd(x_0) \cdots sd(x_1) \cdots \cdots sd(x_{n-1}) \cdot 01.$$

For a set $X$, we use the notation $\mathcal{P}(X)$ to represent the set of all subsets of $X$.

The Boolean value of a condition $\phi$ is

$$[\phi] = \begin{cases} 
1 & \text{if } \phi \text{ is true} \\
0 & \text{if } \phi \text{ is false.} 
\end{cases}$$

We say that a condition $\varphi(n)$ holds infinitely often (i.o.) if it holds for infinitely many $n \in \mathbb{N}$. We say that a condition $\varphi(n)$ holds almost everywhere (a.e.) if it holds for all but finitely many $n \in \mathbb{N}$.

A language is a set of strings. The characteristic sequence of a language $A$ is the sequence $R$ such that

$$R[n] = [s_n \in A]$$
for all $n \in \mathbb{N}$. In this thesis we do not differentiate between a language and its characteristic sequence.

All logarithms in this thesis are base-2 logarithms.

## 2.2 Models of Computation

In this thesis we utilize two models of computation for measuring the depth (organizational complexity) of strings and sequences. We formally define these models below.

### 2.2.1 Finite-State Transducers

We use finite-state transducers to model the behavior of artificial agents in chapter 5. The model used here is similar to that found in [33] where on each transition, the finite-state outputs a string. We define a finite-state transducer formally as follows.

**Definition 2.1.** A *finite-state transducer* is a 6-tuple $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where

1. $Q$ is the finite set of states,
2. $\Sigma$ is the input alphabet,
3. $\Delta$ is the output alphabet,
4. $\delta$ is the input transition function mapping $Q \times \Sigma$ into $Q$, 
5. $\lambda$ is the output function mapping $Q$ into $\Delta$,
6. $q_0 \in Q$ is the start state.
(5) $\lambda$ is the output transition function mapping $Q \times \Sigma$ into $\Delta^*$, and

(6) $q_0 \in Q$ is the initial state.

As is standard, finite-state transducers operate by reading the symbols of the input string sequentially. As each symbol is read, the transition function determines the next state and the string output by the machine using the current state and current symbol being read. The output of a finite-state transducer is simply the concatenation of all the strings output as the input is read.

2.2.2 Self-delimiting Turing Machines

The Turing machine model of computation is widely accepted as the formal model used in the general theory of computation. The basic Turing machine model consists of a finite-state control together with a read head that can read and write to an infinite tape.

For technical reasons, in this thesis we use self-delimiting Turing machines [49, 50, 66, 11]. This Turing machine model forms the basis of algorithmic information theory, a measure of the amount of information in a string based on program-size complexity. (This is defined precisely below.) Essentially, the self-delimiting model requires that in order for a computation to succeed the program tape must scan the last bit of the input string $x$ when the finite-control halts (if it halts). Thus, the length of the program must be encoded in the program in order for the self-delimiting Turing machine to halt on the last bit. Formally, self-delimiting Turing
machines are defined as follows.

**Definition 2.2.** A *self-delimiting Turing machine* is a 4-tuple $M = (Q, \delta, q_0, F)$ where

1. $Q$ is the finite set of states,
2. $\delta$ is the transition function mapping $Q \times \{0, 1\} \times \{0, 1, B\} \times \{0, 1, B\}$ into $Q \times \{0, 1, B\} \times \{0, 1, B\} \times \{0, 1, B\} \times \{L, R\} \times \{L, R\} \times \{S, R\}$, where $B$ is a blank symbol and $L, R, S$ denote if the tape head should move left, right or stay stationary, respectively,
3. $q_0 \in Q$ is the start state,
4. $F \subseteq Q$ is the set of final states.

As shown in Figure 1, a self-delimiting Turing machine is initialized with the program on the one-way, read-only, infinite program tape, the other tapes are initialized to all blanks. The semantics of self-delimiting Turing machines are similar to the semantics of Turing machines. In one move, the self-delimiting Turing machine

- scans the symbols under the read head of the program tape and the read heads of the two work tapes,
- changes to the next state based on the current state and the symbols read,
• write symbols to the work tapes and the output tape, and

• moves the scanning heads of the work tapes and perhaps moves the scanning head of the program tape.

The distinguishing features of the self-delimiting model are the following.

(1) The program-tape head only moves right or does not move.

(2) The output-tape head is moved to the right automatically whenever a symbol is output.

(3) A computation is valid if the machine halts and the program-tape scanning head is scanning the last bit of the program.

(4) If the computation is valid, the output of the machine is the string written on the output tape.

Item (3) above is key for defining algorithmic information theory. By forcing the program-tape head to scan the last bit of the input for valid computations, we force the program to contain information about the length of the program. It is easy to see then that the set of valid programs (programs that perform valid computations) on a self-delimiting Turing machine is a prefix-code. If \( \pi \in \{0,1\}^\ast \) is a valid program, then no extension \( \pi \cdot x \ (x \in \{0,1\}^\ast - \{\lambda\}) \) can be a valid program since the self-delimiting Turing machine would still halt on the last bit of \( \pi \).
Figure 1: Example of self-delimiting Turing machine as seen before the first step
For a program $\pi$ and a (self-delimiting) Turing machine $M$, we say that $M(\pi)$ converges, and we write $M(\pi) \downarrow$, if the program $\pi$ is a valid computation for the (self-delimiting) Turing machine $M$. Otherwise we say that $M(\pi)$ diverges, and we write $M(\pi) \uparrow$.

It is well-known that there are (self-delimiting) Turing machines $U$ that are universal in the sense that, for every (self-delimiting) Turing machine $M$, there exists a program $\pi_M \in \{0, 1\}^*$ such that, for all $\pi \in \{0, 1\}^*$,

$$U(\pi_M \pi) = M(\pi).$$

(This condition means that $M(\pi) \downarrow$ if and only if $U(\pi_M \pi) \downarrow$, in which case the output of $U(\pi_M \pi)$ is the same as the output of $M(\pi)$.) Furthermore, there are universal (self-delimiting) Turing machines $U$ that are efficient, in the sense that, for each (self-delimiting) Turing machine $M$ there is a constant $c \in \mathbb{N}$ (which depends on $M$) such that, for all $\pi \in \{0, 1\}^*$,

$$time_U(\pi_M \pi) \leq c(1 + time_M(\pi) \log time_M(\pi)).$$

In this dissertation, all Turing machines are self-delimiting unless otherwise specified, and we fix a universal, efficient self-delimiting Turing machine $U$.

Oracle Turing machines (OTM) are used to define Turing machine computa-
tion relative to an oracle language (sequence). (Recall that a language is easily represented by a sequence and vice-versa.) Self-delimiting oracle Turing machines operate the same as their non-oracle counterparts, except that they have access to an oracle \( A \in \mathbb{C} \). Through the use of a query tape, a (self-delimiting) oracle Turing machine may query a bit of the oracle \( A \), and use this information in a computation.

For an oracle \( A \subseteq \{0, 1\}^* \) and oracle Turing machine \( M \), the output of \( M \) with oracle \( A \) on program input \( \pi \in \{0, 1\}^* \) is denoted \( M^A(\pi) \), provided that \( M^A(\pi) \downarrow \).

Given a recursive time bound \( t : \mathbb{N} \to \mathbb{N} \), we say that an oracle Turing machine \( M \) is \( t \)-time-bounded if, given any input \( n \in \mathbb{N} \) and oracle \( S \in \mathbb{C} \), \( M \) outputs a bit \( M^S(n) \) in at most \( t(|s_n|) \) time.

### 2.3 Reductions

It is often useful to consider whether a language can be derived from another language. This is formalized by the notion of reducibility. If a language \( A \) can be derived from language \( B \) then we say that \( A \) reduces to \( B \). However, it is usually more important to understand how \( A \) can be derived from \( B \). For this we define the following reducibilities used in this thesis.

**Definition 2.3.** Given a (self-delimiting) oracle Turing machine \( M \), a sequence \( A \in \mathbb{C} \) is Turing reducible to a sequence \( B \in \mathbb{C} \) via \( M \), and we write \( A \leq_T B \) via
$M$, if for all $n \in \mathbb{N}$, $A[n] = M^B[n]$. We write $A \leq_T B$ if there is some oracle Turing machine $M$ such that $A \leq_T B$ via $M$.

It is often the case that we wish to bound the running time of the machine $M$ in the reduction. In this case we define the time-bounded Turing reducibility as follows.

**Definition 2.4.** Given a recursive time bound $t : \mathbb{N} \rightarrow \mathbb{N}$ and an oracle Turing machine $M$, a sequence $A \in \mathcal{C}$ is $\text{DTIME}(t(n))$ Turing reducible to a sequence $B \in \mathcal{C}$, and we write $A \leq_{\text{DTIME}(t)} T B$ via $M$, if for all $n \in \mathbb{N}$, $A[n] = M^B[n]$ in time $t(n)$. We write $A \leq_{\text{DTIME}(t)} T B$ if there is some oracle Turing machine $M$ such that $A \leq_{\text{DTIME}(t)} T B$ via $M$.

It is sometimes the case that general oracle Turing machines offer too much computational power for the required reduction. It is often the case that a restriction is placed on the way in which the OTM may query the oracle.

**Definition 2.5.** As in [62], we define a truth-table condition (briefly, a tt-condition) to be an ordered pair $\tau = ((n_1, \ldots, n_k), g)$, where $k, n_1, \ldots, n_k \in \mathbb{N}$ and $g : \{0, 1\}^k \rightarrow \{0, 1\}$. We write TTC for the set of all tt-conditions. The tt-value of a sequence $S \in \mathcal{C}$ under a tt-condition $\tau = ((n_1, \ldots, n_k), g)$ is the bit

$$\tau^S = g(S[n_1] \cdots S[n_k]).$$
A truth-table reduction (briefly, a tt-reduction) is a computable function \( F : \mathbb{N} \to \) TTC. A tt-reduction \( F \) naturally induces a function \( \hat{F} : C \to C \) defined by

\[
\hat{F}(S)[n] = F(n)^S
\]

for all \( n \in \mathbb{N} \). In general, we identify a tt-reduction \( F \) with the induced function \( \hat{F} \), writing \( \hat{F} \) for either function. For \( A, B \in C \), \( A \) is truth-table reducible (briefly, tt-reducible) to \( B \), and we write \( A \leq_{tt} B \), if there is a tt-reduction \( F \) such that \( A = F(B) \).

It is easy to verify that tt-reductions are equivalent to time-bounded Turing reductions in the following sense.

Observation 2.6. For \( A, B \in C \), \( A \leq_{tt} B \) if and only if there exists a computable time-bound \( t : \mathbb{N} \to \mathbb{N} \) such that \( A \leq_{T}^{\text{DTIME}(t)} B \).

Definition 2.7. A uniform reducibility is a computable function \( F : \mathbb{N} \times \mathbb{N} \to \) TTC.

If \( F \) is a uniform reducibility, then we use the notation \( F_k(n) = F(k, n) \), thereby regarding \( F \) as a computable sequence \( F_0, F_1, F_2, \ldots \) of tt-reductions.

Definition 2.8. If \( F \) is a uniform reducibility and \( A, B \in C \), then \( A \) is \( F \)-reducible to \( B \), and we write \( A \leq_{F} B \), if there exists \( k \in \mathbb{N} \) such that \( A = F_k(B) \).
The following fact is well-known and easy to verify.

**Observation 2.9.**

1. For every computable function \( t : \mathbb{N} \rightarrow \mathbb{N} \), there is a uniform reducibility \( F \) such that, for all \( A, B \in \mathcal{C} \),

\[
A \leq_F B \iff A \leq_{\text{DTIME}(t)}^T B.
\]

2. For every uniform reducibility \( F \), there is a computable function \( t : \mathbb{N} \rightarrow \mathbb{N} \) such that, for all \( A, B \in \mathcal{C} \),

\[
A \leq_F B \implies A \leq_{\text{DTIME}(t)}^T B
\]

In this thesis we define a **finite-state reduction** as follows.

**Definition 2.10.** Given a finite-state transducer \( M \), a sequence \( A \in \mathcal{C} \) is **finite-state reducible to a sequence** \( B \in \mathcal{C} \), and we write \( A \leq_{\text{FST}} B \) via \( M \), if for all \( n \in \mathbb{N} \), there exists \( m \in \mathbb{N} \) such that \( A[0..n-1] = M(B[0..m-1]) \).

### 2.4 Algorithmic Information Theory

Turing machines are often the formal model researchers utilize when an algorithmic or deterministic process is required. If accepted as true, the Church-Turing
thesis [14, 71] tells us that any mechanical finite-size rule based machine can be simulated by a Turing machine. Thus, it is not surprising that algorithmic information theory utilizes self-delimiting Turing machines in order to quantify the amount of algorithmic information in an object or string.

**Definition 2.11.** The set of programs for a Turing machine $M$ is

$$\text{PROG}_M = \left\{ \pi \in \{0,1\}^* \mid M(\pi) \downarrow \right\}.$$  

As noted earlier, the set of programs that halt on a self-delimiting Turing machine form a prefix code. Thus $\text{PROG}_M$ is a prefix code, and by Kraft's inequality,

$$\sum_{\pi \in \text{PROG}_M} 2^{-|\pi|} \leq 1.$$  

**Definition 2.12.** The set of programs for a string $x \in \{0,1\}^*$ relative to a Turing machine $M$ is

$$\text{PROG}_M(x) = \left\{ \pi \in \{0,1\}^* \mid M(\pi) = x \right\}.$$  

**Definition 2.13.** Given a time bound $t : \mathbb{N} \rightarrow \mathbb{N}$, the set of $t$-fast programs for $x$ relative to $M$ is

$$\text{PROG}^t_M(x) = \left\{ \pi \in \text{PROG}_M(x) \mid \text{time}_M(\pi) \leq t(|x|) \right\}.$$
(Note that in the above definition that the time bound is computed in terms of the output length \(|x|\).) Since for all \(x \in \{0,1\}^*\) and all time bounds \(t\), \(\text{PROG}_M(x) \subseteq \text{PROG}_M\) and \(\text{PROG}^t_M(x) \subseteq \text{PROG}_M\), \(\text{PROG}_M(x)\) and \(\text{PROG}^t_M(x)\) are also prefix codes. We write \(\text{PROG}, \text{PROG}(x)\), and \(\text{PROG}^t(x)\) for \(\text{PROG}_U, \text{PROG}_U(x),\) and \(\text{PROG}^t_U(x)\), respectively.

Central to the ideas of algorithmic information theory, Kolmogorov complexity provides the measure by which information is quantified. Kolmogorov complexity, also called program-size complexity, was discovered independently by Solomonoff [68], Kolmogorov [41], and Chaitin [9]. Self-delimiting Kolmogorov complexity is a technical improvement of the original formulation that was developed independently, in slightly different forms, by Levin [49, 50], Schnorr [66], and Chaitin [11]. The advantage of the self-delimiting version is that it gives precise characterizations of algorithmic probability and randomness.

**Definition 2.14.** Let \(x \in \{0,1\}^*\), let \(t : \mathbb{N} \rightarrow \mathbb{N}\) be a time bound, and let \(M\) be a Turing machine.

1a. The *(self-delimiting)* Kolmogorov complexity of \(x\) relative to \(M\) is

\[
K_M(x) = \min \left\{|\pi| \mid \pi \in \text{PROG}_M(x)\right\}
\]
1b. The (self-delimiting) Kolmogorov complexity of $x$ is

$$K(x) = K_U(x).$$

The quantity $K(x)$ is also called the algorithmic entropy, or algorithmic information content, of $x$.

2a. The $t$-time-bounded (self-delimiting) Kolmogorov complexity relative to $M$ is

$$K_M^t(x) = \min \left\{ |\pi| \mid \pi \in \text{PROG}_M(x) \right\}.$$

2b. The $t$-time-bounded (self-delimiting) Kolmogorov complexity, or $t$-time-bounded algorithmic entropy, of $x$ is

$$K^t(x) = K_U^t(x).$$

3a. The algorithmic probability of $x$ relative to $M$ is

$$m_M(x) = \Pr(\text{PROG}_M(x)).$$
3b. The *algorithmic probability* of \( x \) is

\[
m(x) = m_U(x).
\]

4a. The *t-time-bounded algorithmic probability of \( x \) relative to \( M \) is*

\[
m^t_M(x) = \Pr(\text{PROG}^t_M(x)).
\]

4b. The *t-time-bounded algorithmic probability of \( x \) is*

\[
m^t(x) = m^t_U(x).
\]

(Here we use the convention that \( \min \emptyset = \infty \))

We now present useful results in information theory that relate algorithmic probability theory to self-delimiting Kolmogorov complexity. The lemmas below are time-bounded versions of the same lemmas that appear in [38], and trivial modifications to the proofs there yield the proofs for the time-bounded versions. They are presented here without proof.

**Lemma 2.15.** For all recursive functions \( l_1 : \mathbb{N} \to \mathbb{N} \) and all recursive functions \( t : \mathbb{N} \to \mathbb{N} \) there exist a recursive function \( l_2 : \mathbb{N} \to \mathbb{N} \) and constant \( c \in \mathbb{N} \) such
that, for all $x \in \{0, 1\}^*$ and all $\pi \in \text{PROG}^t(x)$,

$$K^{t_2}(x) \leq K^t(\pi) + c$$

**Theorem 2.16.** Let $t : \mathbb{N} \to \mathbb{N}$ be recursive.

(a) For all $x \in \{0, 1\}^*$,

$$-\log m^t(x) \leq K^t(x).$$

(b) There exist a recursive function $t_1 : \mathbb{N} \to \mathbb{N}$ and a constant $c \in \mathbb{N}$ such that, for all $x \in \{0, 1\}^*$

$$K^{t_1}(x) < \log m^t(x) + c.$$

**Lemma 2.17.** There exist a recursive function $t_1 : \mathbb{N} \to \mathbb{N}$ and a constant $c$ such that, for all $\pi \in \{0, 1\}^*$, if $\pi$ computes a finite prefix code $I$, then for all $x \in I$,

$$K^{t_1}(x) \leq |x| + \log \Pr(I) + |\pi| + c.$$  

**Corollary 2.18.** For every recursive function $t_1 : \mathbb{N} \to \mathbb{N}$ and recursive function $t_2 : \mathbb{N} \to \mathbb{N}$, there exists a function $t_3 : \mathbb{N} \to \mathbb{N}$ and a constant $c \in \mathbb{N}$ such that, for all $y \in \{0, 1\}^*$ and all $\pi \in \text{PROG}^{t_1}(y)$,

$$K^{t_3}(\pi) \leq |\pi| + \log m^{t_2}(y) + K^{t_1}(y) + c.$$
Lemma 2.19 (Chaitin [11]). There is a constant \(c \in \mathbb{N}\) such that, for all \(n, k \in \mathbb{N}\),

\[
\left| \left\{ x \in \{0,1\}^n \mid K(x) \leq n + K(n) - k \right\} \right| < 2^{n+c-k}.
\]

We also use the following result on the noncomputability of \(K(n)\).

Theorem 2.20 (Kolmogorov, reported in [76]). If \(g : \mathbb{N} \to \mathbb{N}\) is partial recursive and unbounded, then there exist infinitely many \(n \in \mathbb{N}\) such that \(K(n) < g(n)\).

2.5 Measure

Definition 2.21. A martingale is a function \(d : \{0,1\}^* \to [0, \infty)\) such that, for all \(w \in \{0,1\}^*\),

\[
d(w) = \frac{d(w0) + d(w1)}{2}.
\]

The following inequality of Kolmogorov is easily verified.

Lemma 2.22. If \(d\) is a martingale and \(0 \leq \alpha \in \mathbb{R}\), then

\[
\Pr_A[(\exists w \subseteq A)d(w) \geq \alpha \cdot d(\lambda)] \leq \frac{1}{\alpha}.
\]

In particular, for all \(w \in \{0,1\}^*\), \(d(w) \leq 2^{|w|}d(\lambda)\).
Definition 2.23. A **recursive martingale** (briefly, a rec-martingale) is a martingale \( d \) for which there exists a total recursive function \( \hat{d} : \mathbb{N} \times \{0, 1\}^* \rightarrow \mathbb{Q} \) such that, for all \( r \in \mathbb{N} \) and \( w \in \{0, 1\}^* \),

\[
|\hat{d}(r, w) - d(w)| \leq 2^{-r}.
\]

Definition 2.24. A **lower semicomputable martingale** is a martingale \( d \) for which there exists a total recursive function \( \hat{d} : \mathbb{N} \times \{0, 1\}^* \rightarrow \mathbb{Q} \) with the following two properties.

(i) For all \( r \in \mathbb{N} \) and \( w \in \{0, 1\}^* \), \( \hat{d}(r, w) \leq \hat{d}(r + 1, w) \).

(ii) For all \( w \in \text{strings} \), \( \lim_{r \rightarrow \infty} \hat{d}(r, w) = d(w) \).

Definition 2.25. A martingale \( d \) **succeeds** on a sequence \( S \in \mathcal{C} \) if

\[
\limsup_{n \rightarrow \infty} d(S[0..n - 1]) = \infty.
\]

Definition 2.26. The **success set** of a martingale \( d \) is

\[
S^\infty[d] = \left\{ S \in \mathcal{C} \middle| d \text{ succeeds on } S \right\}.
\]

Definition 2.27. Let \( X \subseteq \mathcal{C} \).
1. $X$ has recursive measure 0, and we write $\mu_{\text{rec}}(X) = 0$, if there is a recursive martingale $d$ such that $X \subseteq S^\infty[d]$.

2. $X$ has recursive measure 1, and we write $\mu_{\text{rec}}(X) = 1$, if $\mu_{\text{rec}}(X^c) = 0$.

3. $X$ has measure 0 in REC, and we write $\mu(X|\text{REC}) = 0$, if $\mu_{\text{rec}}(X \cap \text{REC}) = 0$.

4. $X$ has measure 1 in REC, and we write $\mu(X|\text{REC}) = 1$, if $\mu(X^c|\text{REC}) = 0$.

**Definition 2.28.** Given a function $g : \mathbb{N} \to [0, \infty)$ and a computable time bound $t : \mathbb{N} \to \mathbb{N}$, we define the classes

$$K_i.o. [g(n)] = \{ S \in C \mid K(S[0..n-1]) < g(n) \text{ i.o.} \}$$

and

$$K_i.o. [t, g(n)] = \{ S \in C \mid K^t(S[0..n-1]) < g(n) \text{ i.o.} \}.$$

**Theorem 2.29** (Lutz [53]). For every computable time bound $t : \mathbb{N} \to \mathbb{N}$ and every real number $0 < \alpha < 1$,

$$\mu \left( K_i.o. [\alpha n] \big| \text{REC} \right) = 0.$$
2.6 Randomness

2.6.1 Martin-Löf randomness

Algorithmic randomness was originally defined by Martin-Löf [54], using constructive versions of ideas from measure theory. Subsequently, Schnorr [64] gave the following equivalent definition in terms of martingales.

**Definition 2.30.** A sequence $S$ is *algorithmically random*, and we write $S \in \text{RAND}$, if there is no lower semicomputable martingale that succeeds on it.

Levin [49, 50], Schnorr [66], and Chaitin [10] showed that algorithmic randomness can be characterized in terms of Kolmogorov complexity. Intuitively, if a string $x$ has small Kolmogorov complexity, i.e., $K(x)$ is much less than $|x|$, then there is a short program $\pi$ such that $|\pi| = K(x)$ and $U(\pi)$ outputs the string $x$. Thus $x$ must contain some redundancy, or pattern, that is exploited by the program $\pi$ to generate $x$. Since a random string contains no such pattern, it must have a Kolmogorov complexity that is essentially as large as its length.

**Theorem 2.31** ([49, 50, 66, 10]). A sequence $S \in \mathbb{C}$ is algorithmically random if and only if there is a constant $c \in \mathbb{N}$ such that

$$K(S[0..n-1]) > n - c \text{ a.e.}$$
2.6.2 Recursive Randomness

Recursive randomness has been investigated by Schnorr [64, 65], van Lambalgen [72], Lutz [53], Wang [74], and others.

**Definition 2.32.** A sequence $S \in C$ is rec-random (recursively random), and we write $S \in \text{RAND}(\text{rec})$, if there is no rec-martingale that succeeds on $S$.

In Chapter 3 we use the following uniform, recursive version of the first Borel-Cantelli lemma.

**Theorem 2.33 (Lutz [53]).** Assume that

$$d : \mathbb{N} \times \mathbb{N} \times \{0, 1\}^* \rightarrow [0, \infty)$$

is a computable function with the following two properties.

(i) For each $k, n \in \mathbb{N}$, the function $d_{k,n}$ defined by $d_{k,n}(w) = d(k, n, w)$ is a martingale.

(ii) The series $\sum_{n=0}^{\infty} d_{k,n}(\lambda)$ ($k = 0, 1, \ldots$) are uniformly rec-convergent.

Then

$$\mu_{\text{rec}} \left( \bigcup_{k=0}^{\infty} \bigcap_{m=0}^{\infty} \bigcup_{n=m}^{\infty} S^1[d_{k,n}] \right) = 0.$$

The following easy consequence of Theorem 2.33 is also used in Chapter 3.
**Corollary 2.34** (Lutz [53]). Assume that $S \in \text{RAND}(\text{rec})$ and let

$$d : \mathbb{N} \times \{0, 1\}^* \to [0, \infty)$$

be a computable function with the following two properties.

(i) For each $n \in \mathbb{N}$, the function $d_n$ defined by $d_n(w) = d(n, w)$ is a martingale.

(ii) The series $\sum_{n=0}^{\infty} d_n(\lambda)$ is rec-convergent.

Then there are only finitely many $n \in \mathbb{N}$ such that $S \in S^1[d_n]$.

An exact rec-martingale is a martingale $d$ with rational values (i.e., $d : \{0, 1\}^* \to \mathbb{Q} \cap [0, \infty)$) that is exactly computable. The following lemma gives a convenient sufficient condition for rec-randomness. It follows immediately from the definition of rec-randomness, the recursive equivalence of martingale success and strong martingale success [72], and the Exact Computation Lemma [39, 55].

**Lemma 2.35.** Let $S \in C$. If for every exact rec-martingale $d$ satisfying $d(\lambda) = 1$ there exist $c_d \in \mathbb{N}$ and infinitely many prefixes $w \subseteq S$ such that $d(w) \leq c_d$, then $S$ is rec-random.

### 2.6.3 Normal Numbers

Normal numbers, also known as $\infty$-distributed $b$-ary sequences [40], were first introduced in 1909 by Borel [40]. Although Borel’s notation is somewhat different,
it is equivalent (when \( b = 2 \)) to the following definition.

**Definition 2.36.** The density of a string \( x \in \{0, 1\}^* \) in a sequence \( S \in \mathbb{C} \) is

\[
\lim_{n \to \infty} \frac{\text{number of occurrences of } x \text{ in } S[0..n-1]}{n},
\]

provided that this limit exists.

**Definition 2.37 (Borel [40]).** A sequence \( S \in \mathbb{C} \) is *normal* if for every \( x \in \{0, 1\}^* \) the density of \( x \) in \( S \) is \( 2^{-|x|} \).

### 2.7 Category

We now turn to the fundamentals of Baire category. Baire category gives a topological notion of smallness, usually defined in terms of “countable unions of nowhere dense sets” [58, 59, 63]. Here it is more convenient to define Baire category in terms of certain two-person, infinite games of perfect information, called Banach-Mazur games.

A Banach-Mazur game is a two-player, infinite game in which the players construct a sequence \( S \in \mathbb{C} \) by taking turns extending a prefix of \( S \). There is a “payoff set” \( X \subseteq \mathbb{C} \) such that player I wins a play of the game if \( S \in X \), and player II wins if \( S \notin X \). A *strategy* for a Banach-Mazur game is a function \( \sigma : \{0, 1\}^* \to \{0, 1\}^* \) such that, for all \( w \in \{0, 1\}^* \), \( w \nsubseteq \sigma(w) \), i.e., \( \sigma(w) \) is a proper extension of
w. A play of a Banach-Mazur game is an ordered pair \((\alpha, \beta)\) of strategies. For \(t \in \mathbb{N}\), the \(t^{th}\) partial result of a play \((\alpha, \beta)\) is the string \(R_t(\alpha, \beta) \in \{0, 1\}^*\) defined by the following recursion.

(i) \(R_0(\alpha, \beta) = \lambda\).

(ii) For all \(i \in \mathbb{N}\), \(R_{2i+1}(\alpha, \beta) = \alpha(R_{2i}(\alpha, \beta))\).

(iii) For all \(i \in \mathbb{N}\), \(R_{2i+2}(\alpha, \beta) = \beta(R_{2i+1}(\alpha, \beta))\).

(Player I uses strategy \(\alpha\), and player II uses strategy \(\beta\).) The result of a play \((\alpha, \beta)\) is the unique sequence \(R(\alpha, \beta) \in \mathbb{C}\) such that, for all \(t \in \mathbb{N}\), \(R_t(\alpha, \beta) \subseteq R(\alpha, \beta)\).

We write \(G[X; S_I, S_{II}]\) for the Banach-Mazur game with payoff set \(X\) in which Player I is required to use a strategy from the set \(S_I\) of strategies and player II is required to use a strategy from the set \(S_{II}\) of strategies. In this dissertation, the sets of strategies that we are interested in are the set \(\text{rec}\), consisting of all computable strategies, and the set \(\text{all}\), consisting of all strategies. We write \(G[X]\) for \(G[X; \text{all}, \text{all}]\).

A winning strategy for player I in a Banach-Mazur game \(G[X; S_I, S_{II}]\) is a strategy \(\alpha \in S_I\) such that, for every strategy \(\beta \in S_{II}\), \(R(\alpha, \beta) \in X\). A winning strategy for player II in a Banach-Mazur game \(G[X; S_I, S_{II}]\) is a strategy \(\beta \in S_{II}\) such that, for every strategy \(\alpha \in S_I\), \(R(\alpha, \beta) \notin X\).

**Definition 2.38** (Mazur and Banach [59]). Let \(X \subseteq \mathbb{C}\).
1. $X$ is *meager* if there is a winning strategy for player II in the Banach-Mazur game $G[X]$.

2. $X$ is *comeager* if $X^c$ is meager.

A meager set is sometimes called a *set of first category*, or a *set of first category in the sense of Baire*.

As an easy example, let $\text{FIN}$ be the set of all characteristic sequences of finite subsets of $\mathbb{N}$, i.e.,

\[ \text{FIN} = \left\{ S \in \mathcal{C} \mid S \text{ has only finitely many } 1's \right\}. \]

Then the strategy $\tau$ defined by $\tau_m(w) = w1$ is a winning strategy for Player II in $G[\text{FIN}]$, so $\text{FIN}$ is meager.

The proof that the above definition is equivalent to the "standard textbook" definition of the meager sets is due to Banach and may be found in [58] or [59]. It is clear that every subset of a meager set is meager and that every countable set $X \subseteq \mathcal{C}$ is meager. In fact, it is well-known that every countable union of meager sets is meager [59]. On the other hand, for every $w \in \{0, 1\}^*$, the strategy

\[ \sigma_m(u) = \begin{cases} w & \text{if } u \not\subset w \\ u0 & \text{otherwise} \end{cases} \]
is a winning strategy for Player I in $G[C_\omega]$, so no cylinder is meager. (This is the Baire Category Theorem [59].) These facts justify the intuition that meager sets are “topologically small,” or (negligible) small in the sense of Baire category. Thus, if a set $X \subseteq C$ is comeager, we say that its elements are “topologically abundant,” or that $X$ is large in the sense of Baire category, or that $X$ contains almost every sequence in the sense of Baire category.

**Definition 2.39** (Lutz [52]). Let $X \subseteq C$.

1. $X$ is rec-meager if there is a winning strategy for player II in the Banach-Mazur game $G[X; \text{all, rec}]$.

2. $X$ is rec-comeager if $X^c$ is rec-meager.

**Definition 2.40** (Lisagor [51], Lutz [52]). Let $X \subseteq C$.

1. $X$ is meager in REC if $X \cap \text{REC}$ is rec-meager.

2. $X$ is comeager in REC if $X^c$ is meager in REC.

For $X \subseteq C$, the implications

$$X \text{ is rec-meager} \implies X \text{ is meager}$$

$$\implies X \text{ is meager in REC}$$
are clear. It is also clear that every subset of a meager set is meager and that every countable set $X \subseteq \mathbb{C}$ is meager. In fact, it is well known that every countable union of meager sets is meager [59]. On the other hand, the Baire Category Theorem [59] says that no cylinder is meager. These facts justify the intuition that meager sets are *negligibly small in the sense of Baire category*. Thus, if a set $X \subseteq \mathbb{C}$ is comeager, we say that $X$ contains *almost every sequence in the sense of Baire category*.

The situation is analogous for sets that are meager in REC. Every subset of a set that is meager in REC is clearly meager in REC. Lisagor [51] has also shown that every *recursive union* (a natural, effective notion of countable union) of sets that are meager in REC is meager in REC and, more importantly, that no cylinder is meager in REC. These facts justify the intuition that, if $X \in \mathbb{C}$ is a set that is meager in REC, then $X \cap \text{REC}$ is a *negligibly small subset* of REC *in the sense of Baire category*. Similarly, if $X$ is comeager in REC, then $X$ contains *almost every sequence in REC in the sense of Baire category*.

It is well-known [59, 51] that a set may be large in the sense of measure but small in the sense of Baire category, or vice versa.
2.8 Computational Depth

Computational depth attempts to measure the organization, and therefore usefulness, of a finite binary string. In this section, we give a brief introduction to computation depth and the relevant theorems to this thesis. The interested reader may read the papers by Bennett [7, 8] or Juedes, Lathrop, and Lutz [38] for more in-depth and detailed analysis of computational depth and its properties.

Roughly speaking, the computational depth (called "logical depth" by Bennett [7, 8]) of an object is the amount of time required for an algorithm to derive the object from its shortest description. (Precise definitions appear below.) Since this shortest description contains all the information in the object, the depth thus represents the amount of "computational work" that has been "added" to this information and "stored in the organization" of the object. (Depth is closely related to Adleman's notion of "potential" [1] and Koppel's notion of "sophistication" [42].)

2.8.1 Computational Depth For Finite Strings

**Definition 2.41** (Bennett [7, 8]). Let \( x \in \{0, 1\}^* \) be a string, and let \( s \in \mathbb{N} \) be a significance parameter. The depth of the string \( x \) at significance level \( s \), is the number

\[
\text{depth}_s(x) = \max \left\{ t \mid K(x) \leq K^t(x) - s \right\},
\tag{2.1}
\]
where we use the convention that $\max 0 = 0$. For any given significance level $s$, a string $x$ is called $t$-deep if $\text{depth}_s(x) \geq t$, and $t$-shallow otherwise.

Note that in Bennett's papers, Bennett [7, 8] commonly uses the following definition of computational depth. We will later see that definitions are quantitatively similar.

**Definition 2.42.** Let $x \in \{0, 1\}^*$ and let $s \in \mathbb{N}$, then the computational depth of $x$ at significance level $s$ is

$$\text{depth}_s(x) = \min \left\{ t \in \mathbb{N} \mid (\exists \pi \in \text{PROG}^t(x)) \right\} |\pi| < K(\pi) + s$$

Also, using standard techniques in algorithmic information theory, the definition of $\text{depth}_s(x)$ can be used to define computational depth in terms of algorithmic probability. In the next section we see that these three definitions are easily extended to infinite sequences and are quantitatively nearly the same.

Figure 2 shows the relationship between depth and $K^t(x)$ for a hypothetical string $x$.

It is easy to see that the above definition (Definition 2.41) gives us a complexity measure with the property that both simple and random strings are shallow. For example, consider a string $x$ such that $K(x) \geq |x|$. (A simple counting argument shows that, for all $n$, at least one string of length $n$ has this property. In fact,
a number of researchers [43] have independently shown that, for all sufficiently large \( n \), at least \( 2^{n-c} \) of the strings of length \( n \) have this property, where \( c \) is a constant that does not depend on \( n \).) Since there is a very fast program of length \(|x| + 2 \log |x| + C\) that simply prints the string \( x \), and since \( K(x) \leq K^t(x) \) for all \( t \), the depth of \( x \) cannot be any greater than the time it takes to print \( x \) at any significance level greater than \( 2 \log |x| \). On the other hand, if \( x \) is simply \( 0^n \), then \( x \) contains at most \( \log |x| \) bits of information, and hence \( K(x) \leq 2 \log |x| \).

Since there is a fast (linear time) program that contains the binary encoding of

![Graph](image)

**Figure 2:** Graph of \( K^t(x) \) and its relationship to computational depth for a hypothetical string \( x \).
the length of $x$ that simply loops and outputs $0^n$, and since $K(x) \leq K^t(x)$ for all $t$, the depth of $x$ can be no greater than the time it takes for the program to output $0^n$ at significance levels greater than $2 \log |x|$. 

In contrast with the two examples described above, the characteristic sequence of the halting language, denoted $\chi_H$, is an example of a sequence that has a high depth measure. (This was proven by Bennett [7, 8] and generalized by Juedes, Lathrop, and Lutz [38].) Consider the first $n$ bits of this sequence, namely the string $\chi_H[0..n - 1]$. This string can be recovered exactly from a program that encodes the length of the string and the number of ones contained in the string. Such a program can easily be written with length at most $2 \log \log n + 2 \log n$. Thus, $\chi_H[0..n - 1]$ contains roughly the same amount of algorithmic information as the string $0^n$; however, the high depth of $\chi_H[0..n-1]$ implies that its information is "buried," or stored more "deeply" in the string, thereby requiring much more computation time to produce it from its minimal description. In effect, the time-bounded Kolmogorov complexity of $\chi_H[0..n - 1]$ drops as $t$ is increased, but it does not drop quickly.

2.8.2 Computational Depth For Infinite Sequences

In this section we extend computational depth to (infinite) sequences, and thus we are able to "measure" analytically the computational depth complexity of languages such as the halting language or the binary expansion of $\pi$. We begin
by extending the finite notions of computational depth to infinite versions by looking at the computational depth of their finite prefixes. We thus define the following depth classes.

**Definition 2.43.** For $t, g : \mathbb{N} \rightarrow \mathbb{N}$ and $n \in \mathbb{N}$, we define the sets

\[
D^t_g(n) = \left\{ S \in \mathbb{C} \mid (\forall \pi \in \text{PROG}^t(S[0..n - 1])) K(\pi) \leq |\pi| - g(n) \right\}
\]

\[
D^t_g = \left\{ S \in \mathbb{C} \mid S \in D^t_g(n) \text{ a.e.} \right\}
\]

\[
\tilde{D}^t_g(n) = \left\{ S \in \mathbb{C} \mid K(S[0..n - 1]) \leq K^t(S[0..n - 1]) - g(n) \right\}
\]

\[
\tilde{D}^t_g = \left\{ S \in \mathbb{C} \mid S \in \tilde{D}^t_g(n) \text{ a.e.} \right\}
\]

\[
\tilde{D}^t_g(n) = \left\{ S \in \mathbb{C} \mid m(S[0..n - 1]) \geq 2^{g(n)}m^t(S[0..n - 1]) \right\}
\]

\[
\tilde{D}^t_g = \left\{ S \in \mathbb{C} \mid S \in \tilde{D}^t_g(n) \text{ a.e.} \right\}
\]

The next Lemma shows that the depth classes $\tilde{D}^t_g$ and $\tilde{D}^t_g$ are quantitatively nearly the same as $D^t_g$ in the sense that they are related by constant significance parameters and recursive time bounds. The following results below are given without proof. The interested reader is referred to [38] for complete proofs.

**Lemma 2.44** (Bennett [7, 8, 38]). If $t : \mathbb{N} \rightarrow \mathbb{N}$ is recursive, then there exist constants $c_0, c_1, c_2 \in \mathbb{N}$ and a recursive function $t_1 : \mathbb{N} \rightarrow \mathbb{N}$ such that the following
six conditions hold for all $g : \mathbb{N} \to \mathbb{N}$ and all $n \in \mathbb{N}$.

1. $D_{g+co}(n) \subseteq \tilde{D}_g(n)$
2. $\tilde{D}_{g+c_1}(n) \subseteq \tilde{D}_g(n)$
3. $\tilde{D}_{g+c_2}(n) \subseteq D_g(n)$
4. $D_{g+co} \subseteq \tilde{D}_g$
5. $\tilde{D}_{g+c_1} \subseteq \tilde{D}_g$
6. $\tilde{D}_{g+c_2} \subseteq D_g$

### 2.8.3 Strong Computational Depth

Using depth classes $D_t^t$, and quantifying over $t$ and $c$, Bennett defines *strong computational depth*. Bennett [8, 38] proves that neither recursive nor random sequences can be strongly deep, and then proves a *slow growth law* for computational depth. Informally, this law says that a recursively time-bounded process cannot increase the computational depth of an object too much. In particular, the set of all strongly deep sequences is closed upward under tt-reductions. We now briefly review these results.

**Definition 2.45** (Bennett [7, 8, 38]). A sequence $S$ is *strongly deep* and we write $S \in \text{strDEEP}$, if for every recursive time bound $t : \mathbb{N} \to \mathbb{N}$ and every constant $c \in \mathbb{N}$, it is the case that $S \in D_t^t$.

Intuitively, a sequence $S \in \mathbb{C}$ is in $D_t^t(n)$ if every $t$-fast program $\pi$ for $x[0..n-1]$ can be compressed by at least $g(n)$ bits. Note that, if $t(n) \leq \tilde{t}(n)$ and $g(n) \leq \tilde{g}(n)$, then $D_t^t(n) \subseteq D_g(n)$. Thus, if $t(n) \leq \tilde{t}(n)$ a.e. and $g(n) \leq \tilde{g}(n)$ a.e., then $D_t^t(n) \subseteq$
49

$D^l(n)$ In particular, if $g(n) = c$ and $\tilde{g}(n) = \tilde{c}$, then we have the situation depicted in Figure 3. As seen in this figure, strongly deep sequences are those whose prefixes can be compressed an ever-increasing amount if given an ever-increasing amount of time.

Using the depth classes defined previously, Bennett [8, 38] showed that they all can be used to give equivalent definitions of strong depth.

**Theorem 2.46** (Bennett [7, 8, 38]). For $S \in C$, the following four conditions are equivalent.

1. $S$ is strongly deep.
2. For every recursive time bound $t : \mathbb{N} \rightarrow \mathbb{N}$ and every constant $c \in \mathbb{N}$,
   
   \[
   \text{depth}_c(S[0..n - 1]) > t(n) \text{ a.e.}
   \]

3. For every recursive time bound $t : \mathbb{N} \rightarrow \mathbb{N}$ and every constant $c \in \mathbb{N}$. $S \in \tilde{D}^l_c$.
4. For every recursive time bound $t : \mathbb{N} \rightarrow \mathbb{N}$ and every constant $c \in \mathbb{N}$. $S \in \tilde{D}^l_c$.

Intuitively, strongly deep sequences contain subtle redundancy that can only be exploited to compress the sequence if large amounts of time are used. The next result shows that random sequences cannot be strongly deep.

**Theorem 2.47** (Bennett [7, 8, 38]). There exists a constant $c$ and computable time bound $t(n) \in O(n \log n)$ such that $D^l_c \cap \text{RAND} = \emptyset$. 

Figure 3: The classes $D^t_\varepsilon$, $D^\ddagger_\varepsilon$ in the case where $t(n) \leq \tilde{t}(n)$ a.e. and $x \leq \tilde{c}$. 
Corollary 2.48 (Bennett [7, 8, 38]).

\[ \text{RAND} \cap \text{strDEEP} = \emptyset. \]

Thus not only are sequences in RAND not strongly deep, but they are extremely shallow in the sense that RAND does not even meet the largest (shallowest) of depth classes.

2.8.4 Slow-Growth Theorem

Theorem 2.49 (Bennett [7, 8, 38]). Let \( R, S \in C. \) If \( R \leq_u S \) and \( S \) is strongly deep, then \( R \) is strongly deep.

Intuitively, the above theorem says that computational depth cannot be created quickly. A sequence that is not strongly deep cannot be converted to a strongly deep sequence in any recursive amount of time. This theorem allows us to quickly prove the following.

Theorem 2.50 (Bennett [7, 8, 38]).

\[ \text{REC} \cap \text{strDEEP} = \emptyset. \]

Thus, recursive sequences are also shallow in the sense that no recursive sequence is strongly deep. This confirms our intuition that recursive sequences are
simple, and are easily compressible in recursive amounts of time. However, unlike sequences in RAND, for every $c \in \mathbb{N}$ and recursive time bound $t$, there is a recursive sequence $R$ such that $R \in D^t_c$. In fact, a much stronger result is true [38]. For every recursive function $t : \mathbb{N} \rightarrow \mathbb{N}$ and every constant $c \in \mathbb{N}$, $D^t_c$ has measure 1 in REC. Thus, while random sequences do not meet any depth class stronger than $D^2_c$, the class of recursive sequences “nearly touches” the class strDEEP.

2.8.5 Weakly Useful Sequences and Strong Depth

We now turn to the main result in [38] which relates strongly deep sequences to those that are useful. Here “useful” means that the sequence may be used to solve a non-negligible subset of recursive problems.

**Definition 2.51.** A sequence $S \in C$ is **weakly useful** if there is a recursive time bound $t : \mathbb{N} \rightarrow \mathbb{N}$ such that $\text{DTIME}^t(S)$ does not have measure 0 in REC.

Using this definition, Juedes, Lathrop and Lutz show that every weakly useful sequence is strongly deep, thus confirming the intuition that strongly deep sequences have structure, and that this structure can be used in to solve nontrivial problems.

**Theorem 2.52.** Every weakly useful sequence is strongly deep

Since the halting language can be used to compute every sequence in REC in time linear with respect to program that computes the recursive sequence, the
above theorem gives us an easy proof that the halting language is strongly deep.

**Corollary 2.53.** The halting language and the diagonal halting language are strongly deep.

### 2.8.6 Weak Computational Depth

Bennett [8] also defines the notion of weak computational depth. Intuitively, a sequence is weakly deep if it cannot be derived from any random sequence in a recursive amount of time. Thus, recursive sequences are not weakly deep since they can be computed directly without the use of a random sequence, and a random sequence is not weakly deep since the identity function is recursive.

**Definition 2.54.** A sequence $S \in \mathcal{C}$ is *weakly deep* and we write $S \in \text{wkDEEP}$, if there is no sequence $R \in \text{RAND}$ such that $S \leq_{tt} R$.

We use the following notation for convenience.

**Definition 2.55.**

$$
\text{REC}_{tt}(\text{RAND}) = \left\{ S \in \mathcal{C} \left| (\exists R \in \text{RAND}) S \leq_{tt} R \right. \right\}
$$

Thus,

$$
\text{wkDEEP} = \text{REC}_{tt}(\text{RAND})^c,
$$
and it is easily seen that

\[ \text{wkDEEP} \cap \text{REC} = \text{wkDEEP} \cap \text{RAND} = \emptyset \]

We now present without proof the relationship between weak computational depth and strong computational depth. Juedes, Lathrop and Lutz [38] show that weak depth is not the same as strong depth by the following theorems.

**Theorem 2.56** (Bennett [8, 38]). \( \text{strDEEP} \subseteq \text{wkDEEP} \).

**Theorem 2.57** ([38]). The set wkDEEP is co-meager.

**Corollary 2.58** ([38]). The set \( \text{wkDEEP} - \text{strDEEP} \) is comeager.

**Corollary 2.59** (Bennett [8, 38]).

\[ \text{strDEEP} \nsubseteq \text{wkDEEP} \]

Figure 4 [38] pictorially shows the relationships between the classes REC, RAND, strDEEP, wkDEEP, and \( \text{REC}_{tt}(\text{RAND}) \).

We now turn to recursively weak depth, which was introduced by Fenner, Lutz, and Mayordomo [19]. Recall from section 2.3 the definitions of tt-reductions and the set \( \text{TTC} \) of all tt-conditions.
Figure 4: A classification of binary sequences.
**Definition 2.60.** A *uniform reducibility* is a computable function $F : \mathbb{N} \times \mathbb{N} \rightarrow \text{TTC}$.

If $F$ is a uniform reducibility, then we use the notation $F_k(n) = F(k, n)$, thereby regarding $F$ as a computable sequence $F_0, F_1, F_2, \ldots$ of tt-reductions.

**Definition 2.61.** If $F$ is a uniform reducibility and $A, B \in \mathbb{C}$, then $A$ is $F$-reducible to $B$, and we write $A \leq_F B$, if there exists $k \in \mathbb{N}$ such that $A = F_k(B)$.

The following fact is well-known and easy to verify.

**Observation 2.62.**

1. For every computable function $t : \mathbb{N} \rightarrow \mathbb{N}$, there is a uniform reducibility $F$ such that, for all $A, B \in \mathbb{C}$,

   $A \leq_F B \iff A \leq_{\text{DTIME}(t)} B.$

2. For every uniform reducibility $F$, there is a computable function $t : \mathbb{N} \rightarrow \mathbb{N}$ such that, for all $A, B \in \mathbb{C}$,

   $A \leq_F B \implies A \leq_{\text{DTIME}(t)} B.$
Definition 2.63. If $F$ is a uniform reducibility and $A \in C$, then the upper $F$-span of $A$ is the set

$$F^{-1}(A) = \{ B \in C \mid A \leq_F B \}.$$ 

Definition 2.64 ([19]). Let $F$ be a uniform reducibility. A sequence $S \in C$ is recursively $F$-deep (briefly, rec-$F$-deep), and we write $S \in \text{rec-$F$-DEEP}$, if

$$\mu_{\text{rec}}(F^{-1}(S)) = 0.$$ 

Definition 2.65 ([19]). A sequence $S \in C$ is recursively weakly deep (briefly, rec-weakly deep), and we write $S \in \text{rec-wkDEEP}$, if, for every uniform reducibility $F$, $S$ is rec-$F$-deep.

If $S$ is a recursive sequence, then it is easy to see that there is a uniform reducibility $F$ such that $F^{-1}(S) = C$. (Intuitively, the reduction decides $S$ without using the oracle.) It is this immediate from the definition that no recursive sequence is rec-weakly deep.

The notion of rec-weak depth is analogous to the notion of weak depth. in the sense that as is easily seen a sequence $S \in C$ is weakly deep if and only if, for every uniform reducibility $F$, the upper span $F^{-1}(S)$ has constructive measure 0. The following is also true.

Observation 2.66. No rec-weakly deep sequence is tt-reducible to a rec-random
sequence.

**Proof.** Assume that \( S \leq_{tt} R \in \text{RAND}(rec) \). Then there is a uniform reducibility \( F \) such that \( R \in F^{-1}(S) \). Since \( R \) is rec-random, this implies that \( \mu_{rec}(F^{-1}(S)) \neq 0 \), whence \( S \) is not rec-weakly deep.

We do not know whether the converse of this observation holds, i.e., whether a sequence that is not tt-reducible to any rec-random sequence must be rec-weakly deep. As it is, however, Observation 4.8, together with the fact that \( \text{RAND} \subseteq \text{RAND}(rec) \) tells us that every rec-weakly deep sequence is weakly deep.

**Observation 2.67** (Fenner, Lutz, and Mayordomo [19]). \( \text{rec-wkDEEP} \subseteq \text{wkDEEP} \).

If \( F \) is any uniform reducibility such that the relation \( \leq_F \) is reflexive, then by Observation 4.7, the set \( \text{rec-F-DEEP} \) must be disjoint from \( \text{RAND}(rec) \), and hence must have measure 0 in \( \mathbb{C} \). However, the measure of \( \text{rec-F-DEEP} \) in \( \text{REC} \) is a different matter.

**Theorem 2.68** (Fenner, Lutz, and Mayordomo [19]). If \( F \) is a uniform reducibility, then

\[
\mu \left\{ \text{rec-F-DEEP} \middle| \text{REC} \right\} = 1,
\]

i.e., almost every sequence in \( \text{REC} \) is recursively \( F \)-deep.
Intuitively, the above definition says that a sequence contains structure (depth) if it cannot be derived from a random sequence in a recursive amount of time. Hence, weakly shallow sequences are those that are recursive, (the reduction simply ignores the oracle and computes the recursive sequence) or nearly random in the sense that they can be derived recursively from some random sequence. Thus, weakly deep sequences are neither simple (recursive) nor nearly random.
CHAPTER 3. RECURSIVE COMPUTATIONAL
DEPTH

In this chapter we investigate the notion of recursive computational depth, a stronger notion of Bennett's computational depth. By adding an additional variable, we define depth^l, a recursive (computable) version of computational depth, and investigate its properties. Similar to Bennett [8] we extend this stronger notion of depth to infinite (binary) sequences and define the classes, \( D_g^l \), \( D_g^{l, \text{rec}} \), and \( \text{rec-strDEEP} \). We show that a slow-growth law, similar to that defined by Bennett [8] for computational depth, also holds for recursive depth. We then investigate the relationship between the classes \( \text{strUSEFUL} \), \( \text{wkUSEFUL} \), \( \text{strDEEP} \), \( \text{rec-wkDEEP} \), \( \text{rec-strDEEP} \), and \( \text{wkDEEP} \). In particular, we show that a weakly useful sequence must be rec-strongly deep, and that there are strongly deep sequences that are not rec-strongly deep. Thus, there are weakly useful sequences that are not strongly deep, answering a question posed by Juedes [37].

3.1 Recursive Depth Classes

As noted in section 2.8.1, the value \( \text{depth}_c(w) \) — the computational depth of a string \( w \) at significance level \( c \) — is not computable from \( w \) and \( c \). The following definition remedies this at the expense of introducing an additional variable.
Definition. For \( w \in \{0, 1\}^* \) and \( c, l \in \mathbb{N} \), the recursive computational depth of \( w \) at significance level \( c \) with latency \( l \) is

\[
\text{depth}_c^l(w) = \min \left\{ t \in \mathbb{N} \mid \exists \pi \in \text{PROG}^l(w) \mid |\pi| < K^l(\pi) + c \right\}.
\]

That is, \( \text{depth}_c^l(w) \) is the minimum amount of time required to obtain \( w \) from a program \( \pi \) that cannot itself be obtained in time \( l \) from a program that is \( c \) or more bits shorter than \( \pi \). It is clear that \( \text{depth}_c^l(w) \) is computable from \( w, c, \) and \( l \); this is why it is called the recursive computational depth. Two other properties of \( \text{depth}_c^l(w) \) are immediately evident. For each \( w \in \{0, 1\}^* \) and \( c \in \mathbb{N} \), \( \text{depth}_c^l(w) \) is nondecreasing in \( l \), and \( \lim_{l \to \infty} \text{depth}_c^l(w) = \text{depth}_c(w) \). For each \( w \in \{0, 1\}^* \) and \( l \in \mathbb{N} \), the value \( \text{depth}_c^l(w) \) is, like \( \text{depth}_c(w) \), nonincreasing in \( c \).

As noted above, recursive computational depth is rendered computable by the addition of the latency parameter \( l \). It is easy to see that for all \( x \in \text{strings} \), and for all \( s \in \mathbb{N} \), there exists an \( l \in \mathbb{N} \) such that

\[
\text{depth}_s(x) = \text{depth}_{s}^l(x).
\]

On the other hand, it is not immediately evident that computational depth and recursive computational depth are different. Ideally, the following conjecture is
true.

**Conjecture 3.1.** For all $l, s \in \mathbb{N}$, there is a string $x \in \{0,1\}^*$ such that

$$\text{depth}_s(x) = 2^{|x|}$$

and

$$\text{depth}_l^s(x) = 0.$$

However, we do show in section 3.3 that if recursive computational depth is extended to infinite sequences, a strong separation exists between strong depth sequences [8, 38] and recursive strong depth sequences defined below. Thus, $l$ can be thought of as a restriction on the amount of computational power available for compressing the string $x$, and if no restriction is placed on $l$, then recursive computational depth is equivalent to computational depth.

We now investigate recursive depth classes by extending the definition of recursive computational depth to sequences. We then restrict the latency to recursive functions which yields a notion of recursive depth classes and recursive strong depth. Analogous to computational depth, these classes may be defined using either Kolmogorov complexity or algorithmic probability. We show that these
definitions are robust in the sense that the classes defined are equivalent up to a constant and recursive time bounds.

We begin by defining the recursive analogs of the depth classes $D_g(n)$ and $D_g'$ discussed in section 2.8.2

**Definition 3.2.** For $t, g : \mathbb{N} \to \mathbb{N}$ and $n \in \mathbb{N}$, define the sets

$$D_{g, t}^I(n) = \left\{ S \in \mathbb{C} \mid \text{depth}_{g(n)}^t(S[0..n - 1]) > t(n) \right\}$$

and

$$D_{g, t}^I = \bigcup_{m=0}^{\infty} \bigcap_{n=m}^{\infty} D_{g, t}^I(n) = \left\{ S \in \mathbb{C} \mid (\forall \infty n) S \in D_{g, t}^I(n) \right\}.$$

Note that

$$D_{g, t}^I(n) = \left\{ S \in \mathbb{C} \mid (\forall \pi \in \text{PROG}^t(S[0..n - 1])) K^t(n)(\pi) \leq |\pi| - g(n) \right\}.$$

(It is crucial here that the left-hand side of the inequality is $K^t(n)(\pi)$, not $K^t(\pi)$, i.e., that the time bound is $t(n)$, not $l(|\pi|)$.)

**Definition 3.3.** Let $t, g : \mathbb{N} \to \mathbb{N}$. A sequence $S \in \mathbb{C}$ is **recursively $t$-deep at significance level $g$**, and we write $S \in D_{g, t}^I$, if there is a computable function
$l : N \to \mathbb{N}$ such that $S \in D^t_g \cdot l$. That is,

$$D^t_g \cdot \text{rec} = \bigcup_{l \in \text{rec}} D^t_g \cdot l.$$ 

It is clear that, for all $t, g, l : N \to \mathbb{N}$ with $l$ computable,

$$D^t_g \cdot l \subseteq D^t_g \cdot \text{rec} \subseteq D^t_g.$$ 

To define recursive strong depth, we substitute $D^t_g \cdot \text{rec}$ for $D^t_g$ in the definition of strong depth.

**Definition 3.4.** A sequence $S \in C$ is *recursively strongly deep* (or, briefly, rec-strongly deep), and we write $S \in \text{rec-strDEEP}$, if for every computable time bound $t : N \to \mathbb{N}$ and every constant $c \in \mathbb{N}$, $S \in D^t_g \cdot \text{rec}$. That is,

$$\text{rec-strDEEP} = \bigcap_{c \in N} D^t_g \cdot \text{rec}.$$ 

**Observation 3.5.** It is clear that $\text{rec-strDEEP} \subseteq \text{strDEEP}$.

**Proof.** This follows immediately from the fact that each $D^t_g \cdot \text{rec} \subseteq D^t_g$.

Since $\text{REC} \cap \text{strDEEP} = \emptyset$, it follows immediately from Observation 3.5 that no recursive sequence can be rec-strongly deep.
Recall that a sequence $S$ is strongly deep if, for every computable time bound $t$ and constant $c$, all but finitely many prefixes of $S$ can be described at least $c$ bits more succinctly without a time bound than with the time bound $t$. In contrast, a sequence $S$ is rec-strongly deep if, for every computable time bound $t$ and constant $c$, there exists a *computable* time bound $l$ such that all but finitely many prefixes of $S$ can be described at least $c$ bits more succinctly *with the time bound* $l$ than with the time bound $t$. Very informally, a sequence is strongly deep if it has more regularity than can be explained by a causal (computational) history of any computable duration. For a sequence to be rec-strongly deep, it must also be the case that, for every computable duration $t$ there is a larger computable duration $l$ such that more of the sequence's regularity can be explained by a causal history of duration $l$ than can be explained by a causal history of duration $t$.

We now prove a recursive analog of Theorem 2.47, stating that rec-strongly deep sequences cannot be rec-random.

**Lemma 3.6.** Let $t, g, l : \mathbb{N} \rightarrow \mathbb{N}$ be computable. If $\Pr(D_g^{t,l}(n)) = 0$, then $\mu_{\text{rec}}(D_g^{t,l}) = 0$.

**Proof.** Assume the hypothesis. Then $\Pr\left(\bigcup_{m=0}^{\infty} \bigcap_{n=m}^{\infty} D_g^{t,l}(n)\right) = 0$, so for each $m \in \mathbb{N}$, $\Pr\left(\bigcap_{n=m}^{\infty} D_g^{t,l}(n)\right) = 0$. Thus, for each $m, k \in \mathbb{N}$, there exists $r \in \mathbb{N}$ such that $\Pr\left(\bigcap_{n=m}^{r} D_g^{t,l}(n)\right) \leq 2^{-k}$. Since $\Pr\left(\bigcap_{n=m}^{r} D_g^{t,l}(n)\right)$ is computable from $m$ and
it follows that the function \( r : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \) defined by

\[
r(m, k) = \text{the least } r \in \mathbb{N} \text{ such that } \Pr \left( \bigcap_{n=m}^r D_{g}^{t_i}(n) \right) \leq 2^{-k}
\]
is computable. For each \( m, k \in \mathbb{N} \), define \( d_{m,k} : \{0,1\}^* \to [0,1] \) by

\[
d_{m,k}(w) = \Pr \left( \bigcap_{n=m}^{r(m,k)} D_{g}^{t_i}(n) \mid C_w \right),
\]
and define \( d : \{0,1\}^* \to [0,\infty) \) by

\[
d(w) = \sum_{m=0}^{\infty} \sum_{k=0}^{\infty} 2^{-m} d_{m,k}(w).
\]
(Note that each \( d_{m,k}(\lambda) \leq 2^{-k} \), so \( d(\lambda) \leq 4 \).) It is routine to check that each \( d_{m,k} \) is an exact rec-martingale, whence \( d \) is a rec-martingale.

Let \( S \in D_{g}^{t_i} \), and let \( a \in \mathbb{N} \) be arbitrary. Fix \( m \in \mathbb{N} \) such that \( S \in \bigcap_{n=m}^{\infty} D_{g}^{t_i}(n) \), and let \( r = r(m,2^m \cdot a) \). Let \( w = S[0..r - 1] \). Then

\[
C_w \subseteq \bigcap_{n=m}^{r} D_{g}^{t_i}(n),
\]
so for all $0 \leq k \leq 2^m \cdot a$,

$$C_w \subseteq \bigcap_{n=m}^{r(m.k)} D_g^{i.t}(n),$$

whence $d_{m,k}(w) = 1$. It follows that

$$d(w) \geq 2^{-m} \sum_{k=0}^{r} d_{m,k}(w) = 2^{-m}(1 + 2^m \cdot a) > a.$$ 

Since $a \in \mathbb{N}$ is arbitrary here, this shows that $S \in S^\infty[d]$.

The preceding paragraph establishes that $D_g^{i.t} \subseteq S^\infty[d]$, whence $\mu_{\text{rec}}(D_g^{i.t}) = 0$.

\[\square\]

**Lemma 3.7.** There exist a computable function $t(n) = O(n \log n)$ and a constant $c \in \mathbb{N}$ such that, for every computable function $l : \mathbb{N} \rightarrow \mathbb{N}$, $\mu_{\text{rec}}(D_c^{i.t}) = 0$.

**Proof.** Let $t$ and $c$ be as in Theorem 2.47, and let $l : \mathbb{N} \rightarrow \mathbb{N}$ be computable.

Then $D_c^{i.t} \subseteq D_c^i$, so $\text{RAND} \cap D_c^{i.t} \subseteq \text{RAND} \cap D_c^i = \emptyset$, so $\text{Pr}(D_c^{i.t}) = 0$. It follows by Lemma 3.6 that $\mu_{\text{rec}}(D_c^{i.t}) = 0$. \[\square\]

**Theorem 3.8.** $\text{RAND(\text{rec})} \cap \text{rec-strDEEP} = \emptyset$. In fact, there exist a computable function $t(n) = O(n \log n)$ and a constant $c \in \mathbb{N}$ such that $\text{RAND(\text{rec})} \cap D_c^{i.t_{\text{rec}}} = \emptyset$.

**Proof.** Let $t$ and $c$ be as in Lemma 3.6. To see that $\text{RAND(\text{rec})} \cap D_c^{i.t_{\text{rec}}} = \emptyset$, let
Fix a computable function \( l : \mathbb{N} \to \mathbb{N} \) such that \( S \in \mathbb{D}_{c, l}^t \). Then, by Lemma 3.7, \( \mu_{rec}(\mathbb{D}_{c, l}^t) = 0 \), so \( \mu_{rec}(\{ S \}) = 0 \), so \( S \not\in \text{RAND}(\text{rec}) \). \( \square \)

As with strong depth, it is useful to have characterizations of rec-strong depth in terms of the time-bounded Kolmogorov complexities and algorithmic probabilities of prefixes. To this end, we define recursive analogs of the classes \( \mathbb{D}^t \) and \( \mathbb{D}^t_{g, l} \) of \cite{38}.

**Definition 3.9.** For \( t, g, l : \mathbb{N} \to \mathbb{N} \) and \( n \in \mathbb{N} \), we define the sets

\[
\mathbb{D}^t_{g, l}(n) = \{ S \in C \mid K^t(S[0..n-1]) \leq K^t(S[0..n-1]) - g(n) \}.
\]

\[
\mathbb{D}^t_{g, l} = \{ S \in C \mid (\forall n) S \in \mathbb{D}^t_{g, l}(n) \}.
\]

\[
\mathbb{D}^t_{g, l, \text{rec}} = \{ S \in C \mid (\exists l \in \text{rec}) S \in \mathbb{D}^t_{g, l}(n) \}.
\]

\[
\mathbb{D}^t_{g, l}(n) = \{ S \in C \mid m^t(S[0..n-1]) \geq 2^{g(n)} m^t(S[0..n-1]) \}.
\]

\[
\mathbb{D}^t_{g, l} = \{ S \in C \mid (\forall n) S \in \mathbb{D}^t_{g, l}(n) \}.
\]

\[
\mathbb{D}^t_{g, l, \text{rec}} = \{ S \in C \mid (\exists l \in \text{rec}) S \in \mathbb{D}^t_{g, l}(n) \}.
\]

The following lemma yields a proof that these definitions of recursive depth are robust in the sense that they all may be used to define rec-strong depth.

**Lemma 3.10.** If \( t, l : \mathbb{N} \to \mathbb{N} \) are computable, then there exist constants \( c_0, c_1, c_2 \in \mathbb{N} \) and computable functions \( t_1, l_1, l_2 : \mathbb{N} \to \mathbb{N} \) such that the follow-
ing nine conditions hold for all \( g : \mathbb{N} \to \mathbb{N} \) and \( n \in \mathbb{N} \).

1. \( D_{g+c_0}^{l,l}(n) \subseteq \tilde{D}_g^{l,l}(n) \)
2. \( \tilde{D}_g^{l,l}(n) \subseteq \tilde{D}_g^{l,l}(n) \)
3. \( \tilde{D}_g^{l,l}(n) \subseteq D_g^{l,l}(n) \)
4. \( D_{g+c_0}^{l,l} \subseteq \tilde{D}_g^{l,l} \)
5. \( \tilde{D}_g^{l,l} \subseteq \tilde{D}_g^{l,l} \)
6. \( \tilde{D}_g^{l,l} \subseteq D_g^{l,l} \)
7. \( D_{g+c_0}^{l,\text{rec}} \subseteq \tilde{D}_g^{l,\text{rec}} \)
8. \( \tilde{D}_g^{l,\text{rec}} \subseteq \tilde{D}_g^{l,\text{rec}} \)
9. \( \tilde{D}_g^{l,\text{rec}} \subseteq D_g^{l,\text{rec}} \)

**Proof.**

Proof of 1. Assume that \( S \in D_{g+c_0}^{l,l}(n) \). Then

\[
(\forall \pi \in \text{PROG}'(S[0..n-1])) \ K'(\pi) \leq |\pi| - g(n) - c_0.
\]

Let \( \tilde{\pi} \) be the shortest program in \( \text{PROG}'(S[0..n-1]) \). From the above inequality, it follows that

\[
K'(\tilde{\pi}) + c_0 \leq |\tilde{\pi}| - g(n).
\]

Let \( c_0 \) be \( c \), \( l \) be \( l_1 \), and \( l_1 \) be \( l_2 \) be as in Lemma 2.15. Then

\[
K'(S[0..n-1]) \leq K'(\tilde{\pi}) + c_0
\]

\[
\leq |\tilde{\pi}| - g(n)
\]

\[
= K'(S[0..n-1]) - g(n),
\]
whence $S \in \widehat{D}_g^{t+1}(n)$. It follows immediately that (4) $D_g^{t+1} \subseteq \widehat{D}_g^{t+1}$ and (7)
$D_g^{t+1, \text{rec}} \subseteq \widehat{D}_g^{t+1, \text{rec}}$

Proof of 4. Assume that $S \in \widehat{D}_g^{t+1}(n)$ then,

$$K^t(S[0..n-1]) \leq k^t(S[0..n-1] - g(n) - c_l.$$ 

By substituting $t$ for $t$ in Theorem 2.16 we have

$$- \log m^t(S[0..n-1]) \leq K^t(S[0..n-1])$$

$$\leq k^t(S[0..n-1] - g(n) - c_l$$

$$\leq - \log m^t(S[0..n-1]) + c_l - g(n) - c_l$$

$$\leq - \log m^t(S[0..n-1]) - g(n),$$

so

$$\log m^t(S[0..n-1]) \geq \log m^t(S[0..n-1]) + g(n),$$

and thus

$$m^t(S[0..n-1]) \geq m^t(S[0..n-1]) 2^{g(n)}.$$
Therefore $S \in \tilde{D}_{g,r}^{l_{1},l}(n)$. It follows immediately that (5) $\tilde{D}_{g+c_{1}}^{l_{1},l} \subseteq \tilde{D}_{g}^{l_{1},l}$ and (8) $\tilde{D}_{g+c_{1}}^{l_{1},\text{rec}} \subseteq \tilde{D}_{g}^{l_{1},\text{rec}}$

Proof of 3. Assume that $S \in \tilde{D}_{g+c_{2}}^{l_{1},l}(n)$. Let $c_{2} = c_{3} + c_{4}$ with $c_{3}$ and $c_{4}$ defined as below. Then,

$$- \log m^t(S[0..n-1] \leq - \log m^t(S[0..n-1]) - g(n) - c_{3} - c_{4}$$

By Theorem 2.16b there is a recursive function $t_{1} : \mathbb{N} \rightarrow \mathbb{N}$ and constant $c_{3} \in \mathbb{N}$ such that

$$K^{t_{1}}(S[0..n-1]) \leq - \log m^t(S[0..n-1]) + c_{3}$$

whence

$$K^{t_{1}}(S[0..n-1]) - c_{3} \leq - \log m^t(S[0..n-1]) - g(n) - c_{4} - c_{3}.$$ 

Rewriting, we have

$$\log m^t(S[0..n-1]) \leq - K^{t_{1}}(S[0..n-1]) - g(n) - c_{4}. \quad (3.1)$$

Let $t$ be $t_{2}$, $t_{1}$ be $t_{1}$, and $l_{1}$ be $t_{3}$ as in Corollary 2.18. Then there exists a recursive
time bound $l_1 : \mathbb{N} \to \mathbb{N}$ such that for all $\pi \in \text{PROG}^t(S[0..n - 1])$.

$$K^{l_1}(\pi) \leq |\pi| + \log m^t(S[\ldots]) + K^{l_1}(S[0..n - 1]) + c_4.$$ Using (3.1) with the above inequality yields

$$K^{l_1}(\pi) \leq |\pi| + K^{l_1}(S[0..n - 1]) + c_4 - K^{l_1}(S[0..n - 1]) - g(n) - c_4$$

$$= |\pi| - g(n).$$

Thus, $S \in D^{l_1}_g(n)$. It follows immediately that (6) $\bar{D}^{l_1}_g \subseteq D^{l_1}_g$ and (9) $\bar{D}^{l_1}_{g+c_3} \subseteq D^{l_1}_{g+c_3}$. □

**Lemma 3.11.** For all $t, l, g : \mathbb{N} \to \mathbb{N}$,

(1) $D^{l_1}_g(n) \subseteq D^l_g(n)$

(2) $\bar{D}^{l_1}_g(n) \subseteq \bar{D}^l_g(n)$

(3) $\bar{D}^{l_1}_{g+c_3} \subseteq \bar{D}^l_g(n)$

**Proof.**

(1) Assume that $S \in D^{l_1}_g(n)$, then for all $\pi \in \text{PROG}^t(S[0..n - 1])$

$$K(\pi) \leq K^{l_1}(\pi) \leq |\pi| - g(n).$$
(2) Assume that $S \in \tilde{D}_{g}^{t, l}(n)$, then

$$K(S[n..]) \leq K^{t}(S[0..n-1]) \leq K^{t}(S[0..n-1]) - g(n).$$

(3) Assume that $S \in \tilde{D}_{g}^{t, l}(n)$, then

$$m(S[0..n-1] \geq m^{t}(S[0..n-1]) \geq m^{t}(S[0..n-1])2^{g(n)}.$$

\[ \square \]

**Corollary 3.12.**

1. $D_{g}^{t, l} \subseteq D_{g}^{t}$.
2. $\tilde{D}_{g}^{t, l} \subseteq \tilde{D}_{g}^{t}$.
3. $\tilde{D}_{g}^{t, l} \subseteq \tilde{D}_{g}^{t}$.

Using Lemma 3.10 we now show that set rec-strDEEP is the same under the various definitions of recursive depth defined in this thesis.

**Theorem 3.13.** For $S \in C$, the following four conditions are equivalent.

1. $S$ is recursively strongly deep.
2. For every recursive time bound $t : \mathbb{N} \rightarrow \mathbb{N}$ and every constant $c \in \mathbb{N}$, there exists a recursive time bound $l : \mathbb{N} \rightarrow \mathbb{N}$, $\text{depth}_l c > t(n)$ a.e.
(3) For every recursive time bound \( t : \mathbb{N} \rightarrow \mathbb{N} \) and every constant \( c \in \mathbb{N} \), there exists a recursive time bound \( l : \mathbb{N} \rightarrow \mathbb{N} \), \( S \in \widetilde{D}_c^t \).

(4) For every recursive time bound \( t : \mathbb{N} \rightarrow \mathbb{N} \) and every constant \( c \in \mathbb{N} \), there exists a recursive time bound \( l : \mathbb{N} \rightarrow \mathbb{N} \), \( S \in \widetilde{D}_c^t \).

**Proof.** The equivalence of (1) and (2) follows immediately from definitions. The equivalence of (1), (3) and (4) follows immediately from Lemma 3.10.

\[\square\]

### 3.2 Class Inclusions

In this section, we establish the basic relations that hold among the weak and strong depth classes defined in Chapter 2. For this and later purposes, we need a technical lemma. This result, called the *deterministic slow-growth law for recursive computational depth*, places a quantitative upper bound on the ability of a time-bounded oracle Turing machine to amplify the depth of its oracle. This recursive version of the slow-growth law is a refinement of Lemma 5.5 of [38] presented in section 2 of this thesis.

As in [38], we need two special notations.

**Definition 3.14.** For any function \( s : \mathbb{N} \rightarrow \mathbb{N} \), we define the function \( s^* : \mathbb{N} \rightarrow \mathbb{N} \) by

\[
s^*(n) = 2^{s([\log n]) + 1}.
\]
Definition 3.15. For any unbounded, nondecreasing function \( f : \mathbb{N} \rightarrow \mathbb{N} \), we define the special-purpose “inverse” function \( f^{-1} : \mathbb{N} \rightarrow \mathbb{N} \) by

\[
f^{-1}(n) = \max \{ m \mid f(m) < n \}.
\]

Definition 3.16. Also as in [38], say that a function \( s : \mathbb{N} \rightarrow \mathbb{N} \) is time-constructible if there exist a constant \( c_s \in \mathbb{N} \) and a Turing machine that, given the standard binary representation \( w \) of a natural number \( n \), computes the standard binary representation of \( s(n) \) in at most \( c_s \cdot s(|w|) \) steps. Using standard techniques [6, 33], it is easy to show that, for every computable function \( r : \mathbb{N} \rightarrow \mathbb{N} \), there is a strictly increasing, time-constructible function \( s : \mathbb{N} \rightarrow \mathbb{N} \) such that, for all \( n \in \mathbb{N} \), \( r(n) \leq s(n) \).

Lemma 3.17 (Recursive Slow Growth Law, version I). Let \( s : \mathbb{N} \rightarrow \mathbb{N} \) be strictly increasing and time-constructible, with the constant \( c_s \in \mathbb{N} \) as witness. For each \( s \)-time-bounded oracle Turing machine \( M \), there is a constant \( c_M \in \mathbb{N} \) with the following property. Given nondecreasing functions \( t, g, l : \mathbb{N} \rightarrow \mathbb{N} \), define the
functions $\tau, \tilde{\tau}, \tilde{g}, \tilde{t} : \mathbb{N} \rightarrow \mathbb{N}$ by

$$
\tau(n) = t(s^*(n+1)) + 4s^*(n+1) + 2(n+1)c_s([\log(n+1)]) +
2ns^*(n+1)s([\log(n+1)]),
$$

$$
\tilde{\tau}(n) = \text{c}_M(1 + \tau(n)[\log \tau(n)]),
$$

$$
\tilde{g}(n) = g(s^*(n+1)) + \text{c}_M.
$$

$$
\tilde{t}(n) = \text{c}_M(1 + \tilde{\tau}(n))\log \tilde{\tau}(n)
$$

For all $A, B \in \mathbb{C}$, if $B \leq^T_{\text{DTIME}(s)} A$ via $M$ and $B \in \mathcal{D}_{\tilde{g}}^{\tilde{t}}$, then $A \in \mathcal{D}_{\tilde{g}}^{\tilde{t}}$.

**Proof.** Let $s$ and $M$ be as in the statement of the lemma. Let $M'$ be a Turing machine that, with program $\pi \in \{0,1\}^*$. operates as follows.

**begin** $M'(\pi)$

$u = U(\pi)$

$n = (s^*)^{-1}(|u|)$

for $i = 0$ to $n - 1$ do

output $M^u_{\omega^0}(i)$ (Note that $M$ is hard coded into this machine.)

**halt**

**end**

Since $U$ is an efficient universal Turing machine, there exist a program prefix
\( \pi_{M'} \in \{0, 1\}^* \) and constant \( C_{M'} \in \mathbb{N} \) such that, for all \( \pi \in \{0, 1\}^* \)

\[
U(\pi_{M'}\pi) = M'(\pi)
\]

and

\[
time_U(\pi_{M'}\pi) \leq C_{M'} \left( 1 + \text{time}_{M'}(\pi) \log \text{time}_{M'}(\pi) \right).
\]

Let \( M'' \) be a Turing machine that, with program \( \pi^* \in \{0, 1\}^* \), simulates \( U(\pi^*) \)

and outputs \( \pi \) if and only if \( U(\pi^*) = \pi_{M'}\pi \) with \( \text{time}_{M''}(\pi^*) \leq \text{time}_U(\pi^*) \). Since \( U \) is universal and efficient, there is a program prefix \( \pi_{M''} \in \{0, 1\}^* \) such that, for all \( \pi^* \in \{0, 1\}^* \)

\[
U(\pi_{M''}\pi^*) = M''(\pi^*)
\]

with

\[
time_U(\pi_{M''}\pi^*) \leq C_{M''} \cdot \left( 1 + \text{time}_{M''}(\pi^*) \log \text{time}_{M''}(\pi^*) \right).
\]

Let

\[
C_M = \max \{ C_{M'}, C_{M''}, |\pi_{M'}| + |\pi_{M''}| \}.
\]

Define \( \tau, \hat{\ell}, \hat{g} \) and \( \hat{l} \) as in the statement of the lemma and assume that \( A, B \in \mathcal{C} \)

satisfy \( B \preceq_{\text{TIME}(s)} A \) via \( M \) and \( B \in \mathcal{D}^{\hat{g}}_{\hat{l}} \). Fix \( n_0 \in \mathbb{Z}^+ \) such that \( B \in \mathcal{D}^{\hat{g}}_{\hat{l}}(n) \)

for all \( n \geq n_0 \) and let \( m_1 = s^*(n_0) + 1 \).

The following two claims are verified at the end of the proof.
CLAIM 1. For all \( m > s^*(1) \) and \( \pi \in \{0, 1\}^* \), if \( \pi \in \text{PROG}^t(A[0..m-1]) \), then 
\( \pi_M \pi \in \text{PROG}^t(B[0..n-1]) \), where \( n = (s^*)^{-1}(m) \).

CLAIM 2. For all \( m \geq m_1 \) and all \( \pi \in \text{PROG}^t(A[0..m-1]) \),

\[
K^t(\pi) \leq |\pi| - \hat{\gamma}(n) + c_M
\]

where \( n = (s^*)^{-1}(m) \).

Given these two claims, we now finish the proof as follows. Let \( m \geq m_1 \), and let \( \pi \in \text{PROG}^t(A[0..m-1]) \). Then, by Claim 2 and the monotonicity of \( g \),

\[
K^t(\pi) \leq |\pi| - \hat{\gamma}(s^*)^{-1}(m) + c_M \\
= |\pi| - g(s^*(s^*)^{-1}(m) + 1)) \\
\leq |\pi| - g(m).
\]

Thus \( A \in D^g_{\hat{\gamma}}(m) \). Since this holds for all \( m \geq m_1 \), it follows that \( A \in D^g_{\hat{\gamma}} \).

We now verify Claim 1 and Claim 2. Claim 1 is proven in [38] and is repeated here for convenience.

PROOF OF CLAIM 1: Assume that \( m \geq m_1 \) and \( \pi \in \text{PROG}^t(b[0..m-1]) \). Let 
\( u = B[0..m-1] \) and \( n = (s^*)^{-1}(m) \). Since \( m \geq m_1 \), we must have \( s^*(n) < m \).
Since \( M \) is \( s \)-time-bounded, this implies that \( M^{u[0..\infty]}(i) = M^B(i) = A[i] \) for all
0 ≤ i < n. (All queries in these computations must be made to bits A[j] for j < |u|.) Thus

\[ U(\pi_{M'}, \pi) = M'(\pi) = B[0..n - 1]. \]

With program π, \( M' \) requires at most \( t(m) \) steps to compute \( u \), at most \( 4m \) additional steps to compute \( |u| \) in binary, at most \( 2(n + 1)c_s(\lceil \log(n + 1) \rceil) \) steps to compute \( n \), and at most \( 2nms(\lceil \log(n + 1) \rceil) \) steps to execute the for-loop. Since \( s^*(n + 1) ≥ m \), and \( t \) is nondecreasing, it follows that \( time_{M'}(\pi) ≤ t(n) \), so

\[ time_U(\pi_{M'}, \pi) ≤ t(n). \]

Thus \( \pi_{M'} \pi \in \text{PROG}^i(B[0..n - 1]) \). This proves Claim 1.

**Proof of Claim 2.** Let \( m ≥ m_1 \). Let \( \pi \in \text{PROG}^i(A[0..m - 1]) \), and let \( n = (s^*)^{-1}(m) \). Since \( m > s^*(n_0) \) it must be the case that

\[ n = (s^*)^{-1}(m) ≥ n_0, \]

and thus \( B \in D_{s}^{\pi_{M'}}(n) \). Since \( m ≥ m_1 = s^*(n_0) + 1 > s^*(1) \), Claim 1 tells us that

\[ \pi_{M'} \pi \in \text{PROG}^i(A[0..n - 1]). \]
Since $B \in D_{\tilde{g}}^t(n)$,

\[
K^t(\pi_{M'}^t \pi) \leq |\pi_{M'}^t \pi| - \tilde{g}(n)
= |\pi| - \tilde{g}(n) + |\pi_{M'}|
\]

Let $\pi^*$ be a shortest element of $\text{PROG}^t(\pi_{M'})$. Then,

\[
U(\pi^*) = \pi_{M'}^t \pi \text{ in } t(|\pi_{M'}^t \pi|) \text{ steps.}
\]

Thus,

\[
U(\pi_{M''} \pi^*) = M''(\pi^*) = \pi
\]

with time bound

\[
time_U(\pi_{M''} \pi^*) \leq c_{M''} \cdot (1 + time_{M''}(\pi^*) \log time_{M''}(\pi^*))
\]

\[
= c_{M''} \cdot (1 + time_U(\pi^*) \log time_U(\pi^*))
\]

\[
\leq c_{M''} \cdot (1 + l(|\pi_{M'} \pi|) \log l(|\pi_{M'} \pi|))
\]

\[
\leq c_{M''} \cdot (1 + l(\tilde{l}(n)) \log l(\tilde{l}(n)))
\]

\[
= \tilde{l}(n).
\]
Thus,

\[ K^t(\pi) \leq |\pi_{M''}^*| = K^t(\pi_{M'}, \pi) + |\pi_{M''}| \]

\[ \leq |\pi| - \tilde{g}(n) + c_M. \]

Therefore we have

\[ K^t(\pi) \leq |\pi_{M''}^*| \]

\[ = K^t(\pi_{M'}, \pi) + |\pi_{M''}| \]

\[ \leq |\pi| - \tilde{g}(n) + c_M. \]

This proves Claim 2 and completes the proof of the lemma. \(\square\)

It is useful to have the following slightly weaker form of the Slow Growth Lemma.

**Lemma 3.18** (Slow Growth Lemma version II). Let \( s : \mathbb{N} \to \mathbb{N} \) be strictly increasing and time-constructible, with the constant \( c_s \in \mathbb{N} \) as witness. For each \( s \)-time-bounded oracle Turing machine \( M \), there is a constant \( c_M \in \mathbb{N} \) with the following property. Given nondecreasing functions \( t, g : \mathbb{N} \to \mathbb{N} \), define the functions
\(\tau, \tilde{t}, \tilde{g} : \mathbb{N} \to \mathbb{N}\) by

\[
\tau(n) = t(s^*(n + 1)) + 4s^*(n + 1) + 2(n + 1)c_s([\log(n + 1)]) + 2ns^*(n + 1)s([\log(n + 1)]),
\]

\[
\tilde{t}(n) = c_M(1 + \tau(n)[\log \tau(n)]),
\]

\[
\tilde{g}(n) = g(s^*(n + 1)) + c_M.
\]

For all \(A, B \in C\), if \(B \leq^\text{DTIME} A\) via \(M\) and \(B \in D^\text{rec}_g\), then \(A \in D^\text{rec}_g\).

**Proof.** The proof follows directly from Lemma 3.17. \(\square\)

An easy consequences of the Slow Growth Lemma is that the class of rec-strongly deep sequences is (like the class of strongly deep sequences [38]) closed upwards under \(tt\)-reductions. The proof of this Theorem is similar to the proof of Theorem 5.6 in [38] except Lemma 3.18 is used in place of Lemma 5.5 in [38]. The proof is given here for completeness.

**Theorem 3.19.** Let \(A, B \in C\). If \(B \leq^t A\) and \(B\) is rec strongly deep, then \(A\) is rec strongly deep.

**Proof.** Assume the hypothesis. To see that \(A\) is rec-strongly deep, fix a recursive function \(t : \mathbb{N} \to \mathbb{N}\) and a constant \(c \in \mathbb{N}\). It suffices to show that \(A \in D^t_c\). Since \(B \leq^t A\), there exist a strictly increasing, time-constructible function \(s : \mathbb{N} \to \mathbb{N}\) and an \(s\)-time-bounded oracle Turing machine \(M\) such that \(B \leq^\text{DTIME}(s) A\) via
Choose a constant $c_M$ for $M$ as in Lemma 3.18 and define $g : \mathbb{N} \to \mathbb{N}$ by $g(n) = c$ for all $n \in \mathbb{N}$. Then, in the notation of Lemma 3.18, $\hat{t}$ is recursive and $\hat{\eta}$ is constant. Since $B$ is strongly rec-deep, it follows that $B \in D^{\hat{\eta}}_{rec}$. It follows by Lemma 3.18 that $A \in D_{\eta}^{rec}$.

In analogy to [38], the Slow Growth Lemma also gives an alternate proof that no recursive sequence is rec-strongly deep.

We now come to the main result of this section. The following theorem gives the inclusion relations that hold among the weak, strong, rec-weak, and rec-strong depth classes defined in chapter 2 and this chapter.

**Theorem 3.20.** The following diagram of inclusions holds.

```
rec-wkDEEP
\subseteq
rec-strDEEP
\subseteq
wkDEEP
\subseteq
strDEEP
```

**Proof.** It was shown by Bennett [8] (see also [citeCDR]) that $strDEEP \subseteq wkDEEP$, and Observations 3.5 and 2.67 tell us that $rec-strDEEP \subseteq strDEEP$ and $rec-wkDEEP \subseteq wkDEEP$, respectively. All that remains, then is to show that $rec-strDEEP \subseteq rec-wkDEEP$. 
Let \( S \in \text{rec-strDEEP} \), and let \( F \) be a uniform reducibility. Fix a strictly increasing, time-constructible function \( s : \mathbb{N} \to \mathbb{N} \) such that, for all \( A, B \in \mathcal{C} \).

\[
A \leq_F B \iff A \leq_{\text{DTIME}(s)} B.
\]

Choose \( t, c \) as in Lemma 3.7. Define \( \widehat{t} \) and \( \widehat{g} \) as in Lemma 3.17, where \( g(n) = c \). Then \( \widehat{g}(n) \) is constant: say \( \widehat{g}(n) = \widehat{c} \). Now \( S \in D_{\widehat{c}}^{\widehat{t}, \text{rec}} \), so there is a computable function \( \ell : \mathbb{N} \to \mathbb{N} \) such that \( S \in D_{\widehat{c}}^{\ell, \text{rec}} \). Define \( \widehat{t} \) as in Lemma 3.17. Then Lemma 3.17 tells us that \( F^{-1}(S) \subseteq D_{\widehat{c}}^{\ell, \text{rec}} \). By Lemma 3.7, \( \mu_{\text{rec}}(D_{\widehat{c}}^{\ell, \text{rec}}) = 0 \), so \( \mu_{\text{rec}}(F^{-1}(S)) = 0 \), i.e., \( S \) is rec-\( F \)-deep. Since \( F \) is arbitrary here, this shows that \( S \in \text{rec-wkDEEP} \).

---

### 3.3 Class Separations

We now show that all four inclusions in Theorem 3.20 are proper. It is most efficient (and most informative) to prove this by proving the two non-inclusions

\[ \text{strDEEP} \not\subseteq \text{rec-wkDEEP} \]

and

\[ \text{rec-wkDEEP} \not\subseteq \text{strDEEP}. \]
We prove these in succession.

We prove that \( \text{strDEEP} \subseteq \text{rec-wkDEEP} \) by proving the much stronger fact that, in contrast with Theorems 2.47 and 3.8, strongly deep sequences can be recursive random. We do this by examining the Kolmogorov and the time-bounded Kolmogorov complexities of recursively random sequences.

We first prove that rec-random sequences have very high time-bounded Kolmogorov complexities.

**Theorem 3.21.** Assume that \( S \) is rec-random and that \( t, g : \mathbb{N} \to \mathbb{N} \) are computable functions with \( g \) nondecreasing and unbounded. Then, for all but finitely many \( n \in \mathbb{N} \),

\[
K^t(S[0..n-1]) > n - g(n).
\]

**Proof.** Assume the hypothesis. For each \( n \in \mathbb{N} \) and \( w \in \{0,1\}^* \), let

\[
\mathcal{E}_n = \left\{ A \left| K^t(A[0..n-1]) \leq n - g(n) \right. \right\}
\]

and

\[
d_n(w) = \Pr(\mathcal{E}_n \mid C_w).
\]
It suffices to show that the set

\[ J = \{ n \in \mathbb{N} \mid S \in \mathcal{E}_n \} \]

is finite.

It is easy to see that the function \((n, w) \mapsto d_n(w)\) is computable, and that each \(d_n\) is a martingale. Choose a constant \(c \in \mathbb{N}\) as in Lemma 2.19, and define \(m : \mathbb{N} \to \mathbb{N}\) by

\[ m(r) = \text{the least } m \in \mathbb{N} \text{ such that } g(m) \geq r + c. \]

Then \(m\) is computable, and for all \(r \in \mathbb{N}\),

\[
\sum_{n=m(r)}^{\infty} d_n(\lambda) = \sum_{n=m(r)}^{\infty} \Pr(\mathcal{E}_n) \\
\leq \sum_{n=m(r)}^{\infty} \Pr[K(A[0..n-1]) \leq n - g(n)] \\
< \sum_{n=m(r)}^{\infty} 2^{c-K(n)-g(n)} \\
\leq \sum_{n=m(r)}^{\infty} 2^{c-K(n)-g(m(r))} \\
\leq 2^{-r} \sum_{n=m(r)}^{\infty} 2^{-K(n)} \\
\leq 2^{-r}.
\]
Thus the series $\sum_{n=0}^{\infty} d_n(\lambda)$ is computably convergent. It follows by Corollary 2.34 that there are only finitely many $n \in \mathbb{N}$ such that $S \in S^1[d_n]$. Since, for all $n \in \mathbb{N}$.

$$n \in J \implies d_n(S[0..n-1]) = 1 \implies S \in S^1[d_n],$$

it follows that $J$ is finite.

The function $g$ above may be very slow-growing, e.g., an inverse Ackermann function. Theorem 3.21 thus says that, for every rec-random sequence $S$ and computable time bound $t$, all but finitely many of the prefixes of $S$ have $K'$-complexities that are nearly as large as their lengths.

We next show that the situation is very different in the absence of the time bound $t$.

**Definition 3.22.** A sequence $S \in C$ is ultracompressible if, for every computable, nondecreasing, unbounded function $g : \mathbb{N} \to \mathbb{N}$, there exists $n_g \in \mathbb{N}$ such that, for all $n \geq n_g$,

$$K(S[0..n-1]) < K(n) + g(n). \quad (3.2)$$

It is clear that every $n$-bit string $w$ must satisfy $K(w) \geq K(n) - O(1)$. A sequence $S$ is thus ultracompressible if, for every computable, nondecreasing, un-
bounded (but perhaps very slowly growing) function \( g \), for all but finitely many \( n \), the \( n \)-bit prefix of \( S \) has \( K \)-complexity that is within \( g(n) \) bits of the minimum possible \( K \)-complexity for an \( n \)-bit string.

We now show that a rec-random sequence can be ultracompressible. Similar results have been proven by Wang [74] and Ambos-Spies and Wang [2] for the monotone Kolmogorov complexities of rec-random sequences. The present result is slightly stronger than these results in that it gives a single rec-random sequence \( S \) that has property (3.2) for every computable, nondecreasing, unbounded function \( g \). The proof is based in part on a simpler, unpublished construction by Gasarch and Lutz [22] of a rec-random sequence that is not algorithmically random.

**Theorem 3.23.** There is a rec-random sequence that is ultracompressible.

**Proof.** Let \( g_0, g_1, g_2, \ldots \) be an enumeration of all computable, nondecreasing, unbounded functions \( g_k : \mathbb{N} \rightarrow \mathbb{N} \), and let \( d_0, d_1, d_2, \ldots \) be an enumeration of all exact rec-martingales \( d_k \) with \( d_k(\lambda) = 1 \). (Both enumerations are necessarily noneffective.) For each \( k \in \mathbb{N} \), fix a program prefix \( \pi_{d_k} \in \{0,1\}^* \) such that, for all \( w \in \{0,1\}^* \), \( U(\pi_{d_k} \text{sd}(w)) = d_k(w) \), where \( \text{sd}(w) \) is the self-delimiting encoding of \( w \) defined in section 2.1. For each \( k \geq -1 \), let \( \pi_k^{(d)} = \langle \pi_{d_0}, \ldots, \pi_{d_k} \rangle \), where \( \langle \ldots \rangle \) is the self-delimiting sequence encoding defined in section 2.1, and let \( a_k = |\pi_k^{(d)}| \).

Our objective is to exhibit a rec-random sequence \( S \) that is ultracompressible.
This sequence $S$ is specified by a sequence

$$w_{-1} \subseteq w_0 \subseteq w_1 \subseteq w_2 \subseteq \ldots \subseteq S$$

of prefixes $w_k$ that are defined inductively below. There is a single Turing machine that carries out all of the extensions from $w_k$ to $w_{k+1}$, given a suitable program at each stage $w_k$. We now describe this machine.

Fix a Turing machine $M$ that, given a program of the form $\pi = \pi_k^{(w)} \pi_{k+1}^{(d)} \pi_n$, where $k \geq -1$, $U(\pi_k^{(w)}) = \langle w_0, \ldots, w_k \rangle$, $U(\pi_n) = s_n$, and $n \geq |w_k|$, outputs the encoded list $\langle w_0, \ldots, w_k, w(k, n) \rangle$, where $w(k, n) \in \{0, 1\}^n$ is the string whose $i^{th}$ bit is given by the recursion

$$w(k, n)[i] = \begin{cases} w_k[i] & \text{if } 0 \leq i < |w_k| \\ [\tilde{d}_k(w(k, n)[0..i - 1]1) \leq \tilde{d}_k(w(k, n)[0..i - 1]0)] & \text{if } |w_k| \leq i < n, \end{cases}$$

where

$$\tilde{d}_k(w) = \sum_{j=0}^{k+1} 2^{-|j+|w|-1|}d_j(w)$$

and $w_{-1} = \lambda$. (If the program $\pi$ for $M$ is not of the above form, then $M(\pi)$, which may or may not be defined, is not used in this proof.)
In more intuitive terms, given such a program \( \pi \), \( M \) extends \( w_k \) one bit at a time, choosing the bit that minimizes the composite martingale \( \tilde{d}_k \) at each step of the extension. In particular, it is evident that

\[
\tilde{d}_k(w(k, n)) \leq d_k(w_k).
\] (3.3)

As defined below, the extended prefix \( w_{k+1} \) is precisely the string \( w(k, n_k) \) for a suitable value of \( n_k \). The rec-randomness of \( S \) is then ensured by (3.3), while the ultracompressibility of \( S \) is ensured by a judicious choice of \( n_k \).

Fix a constant \( c \in \mathbb{N} \) such that, for all \( k \in \mathbb{N} \) and all \( w_0, \ldots, w_k \in \{0, 1\}^* \),

\[
K(w_k) \leq K(\langle w_0, \ldots, w_k \rangle) + c;
\] (3.4)

and, for all \( x \in \{0, 1\}^* \),

\[
K(x) \leq K_M(x) + c.
\] (3.5)

Define the sequence

\[
w_{-1} \not\subseteq w_0 \not\subseteq w_1 \not\subseteq w_2 \not\subseteq \ldots
\]

inductively as follows. First, let \( w_{-1} = \lambda \). Next, assume that \( w_{-1} \not\subseteq \ldots \not\subseteq w_k \) have
been defined, where \( k \geq -1 \). For each \( n \geq |w_k| \), let

\[
\pi(k, n) = \pi_k^{(w)} \pi_{k+1}^{(d)} \pi_n,
\]

where \( \pi_k^{(w)} \) is a minimum-length program for \((w_0, \ldots, w_k)\) and \( \pi_n \) is a minimum-length program for \( s_n \), and let \( w(k, n) \) be the (unique) string such that

\[
M(\pi(k, n)) = (w_0, \ldots, w_k, w(k, n)).
\]

Note that, for all \( k \geq -1 \) and \( n \geq |w_k| \),

\[
K((w_0, \ldots, w_k, w(k, n))) \leq K(n) + K((w_0, \ldots, w_k)) + a_{k+1} + c.
\] (3.6)

This is because, by (3.5),

\[
K((w_0, \ldots, w_k, w(k, n))) = K(M(\pi(k, n))) \leq K_M(M(\pi(k, n))) + c \leq |\pi(k, n)| + c = K(n) + K((w_0, \ldots, w_k)) + a_{k+1} + c.
\]

Define \( \tilde{g} : \mathbb{N} \to \mathbb{N} \) by

\[
\tilde{g}(n) = \min_{0 \leq j \leq k+1} \left\lfloor \frac{g_j(n)}{2} \right\rfloor.
\]

Then \( \tilde{g} \) is computable and unbounded, so by Theorem 2.20 there exist infinitely
many $n \in \mathbb{N}$ such that $K(n) < \overline{g}(n)$. Thus we can fix $n_k > |w_k|$ such that

$$K(n_k) < \overline{g}(n_k)$$

(3.7)

and

$$K((w_0, \ldots, w_k)) + a_{k+1} + a_{k+2} + 3c < \overline{g}(n_k).$$

(3.8)

Let $w_{k+1} = w(k, n_k)$. This completes the definition of the sequence $w_{-1} \subset w_0 \subset w_1 \subset \ldots$. For all $0 < j < l + 1$, by (3.6), (3.7), and (3.8),

$$K ((w_0, \ldots, w_{l+1})) + a_{l+2} + 2c = K ((w_0, \ldots, w_l, w(l, n_l))) + a_{l+2} + 2c$$

$$\leq K (n_l) + K ((w_0, \ldots, w_l)) + a_{l+1} + a_{l+2} + 3c$$

$$< 2\overline{g}(n_l)$$

$$= 2\overline{g}(|w_{l+1}|)$$

$$\leq g_j (|w_{l+1}|).$$

It follows by the change of variable $k = l + 1$ that, for all $0 \leq j \leq k$,

$$K ((w_0, \ldots, w_k)) + a_{k+1} + 2c < g_j (|w_k|).$$

(3.9)
We next show that, for all $k \geq -1$,

\[ \tilde{d}_k(w_k) \leq 2 - 2^{-(k+1)} \quad (3.10) \]

We prove this by induction on $k$. It clearly holds for $k = -1$; assume that it holds for $k$. Then, by (3.3), Lemma 2.22, and (3.10),

\[
\tilde{d}_{k+1}(w_{k+1}) = \tilde{d}_k(w_{k+1}) + 2^{-(k+2+|w_{k+1}|)} d_{k+2}(w_{k+1}) \\
\leq \tilde{d}_k(w_k) + 2^{-(k+2)} \\
\leq 2 - 2^{-(k+1)} + 2^{-(k+2)} \\
= 2 - 2^{-(k+2)},
\]

so it holds for $k + 1$.

Now let $S$ be the unique sequence such that $w_k \subseteq S$ for all $k \in \mathbb{N}$. We show that $S$ is rec-random and ultracompressible.

To see that $S$ is rec-random, let $d$ be an exact rec-martingale with $d(\lambda) = 1$. Fix $j \in \mathbb{N}$ such that $d_j = d$. Then, for all $k > j$, (3.10) tells us that the prefix $w_k$
of $S$ satisfies

$$d(w_k) = 2^{j+|w_{j-1}|}2^{-((j+|w_{j-1}|)}d_j(w_k)$$

$$\leq 2^{j+|w_{j-1}|}d_k(w_k)$$

$$\leq 2^{j+|w_{j-1}|}(2 - 2^{-(k+1)})$$

$$< 2^{j+|w_{j-1}|+1}.$$ 

It follows by Lemma 2.35 that $S$ is rec-random.

Finally, to see that $S$ is ultracompressible, let $g : \mathbb{N} \to \mathbb{N}$ be computable, nondecreasing, and unbounded. Fix $j \in \mathbb{N}$ such that $g_j = g$, and let $n \geq |w_j|$. Fix $k \in \mathbb{N}$ such that $|w_k| \leq n < |w_{k+1}|$. Then, by (3.4), (3.6), (3.9), and the fact that $g$ is nondecreasing,

$$K(S[0..n-1]) = K(w(k,n))$$

$$\leq K((w_0, \ldots, w_k, w(k,n))) + c$$

$$\leq K(n) + K((w_0, \ldots, w_k)) + a_{k+1} + 2c$$

$$< K(n) + g_j(|w_k|)$$

$$\leq K(n) + g(n).$$

Hence $S$ is ultracompressible. □
We now note that rec-random sequences can be strongly deep.

**Theorem 3.24.** There is a rec-random sequence that is strongly deep.

**Proof.** By Theorem 3.23, there is a rec-random sequence $S$ that is ultracompressible. To see that $S$ is strongly deep, fix a computable function $t : \mathbb{N} \rightarrow \mathbb{N}$ and a constant $c \in \mathbb{N}$. By Theorem 2.46, it suffices to show that $S \in \hat{D}_c^t$.

Fix a real number $\alpha$ such that $0 < \alpha < 1$, and define $g : \mathbb{N} \rightarrow \mathbb{N}$ by

$$g(n) = \left\lfloor \frac{(1 - \alpha)n}{3} \right\rfloor.$$

Then $g$ is computable, nondecreasing, and unbounded, so by Theorem 3.21, there exists $n_1 \in \mathbb{N}$ such that, for all $n \geq n_1$,

$$K^t(S[0..n-1]) > n - g(n). \quad (3.11)$$

Also, since $S$ is ultracompressible, there exists $n_2 \in \mathbb{N}$ such that, for all $n \geq n_2$,

$$K(S[0..n-1]) < K(n) + g(n). \quad (3.12)$$

Finally, there exists $n_3 \in \mathbb{N}$ such that, for all $n \geq n_3$,

$$K(n) \leq g(n). \quad (3.13)$$
Let \( n_0 = \max\{n_1, n_2, n_3\} \). Then, for all \( n \geq n_0 \), (3.11), (3.12) and (3.13) tell us that

\[
K'(S[0..n - 1]) - K(S[0..n - 1]) > n - 3g(n) \geq \alpha n.
\]

Hence, \( S \in \hat{D}_{\alpha n} \subseteq \hat{D}_c \).

The rec-random sequence \( S \) given by the above proof is not only strongly deep, but is in the class \( \hat{D}_{\alpha n} \) for all computable time bounds \( t \). Since the real number \( \alpha \) may be arbitrarily close to 1, this says that \( S \) is strongly deep at very high significance levels (significance levels very close to \( n \) bits).

Theorem 3.24 contrasts sharply with Theorems 2.47 and 3.8. There is of course nothing paradoxical in this contrast. It is merely a consequence of the strong, quantitative separation of RAND(rec) from RAND given by Theorems 2.31 and 3.23.

We now have the first of the desired noninclusions.

**Corollary 3.25.** \( \text{strDEEP} \nsubseteq \text{rec-wkDEEP} \).

**Proof.** By Theorem 3.24, there is a sequence \( S \) that is rec-random and strongly deep. Since \( S \) is rec-random, Observation 2.66 tells us that \( S \) is not rec-weakly deep. \( \square \)

Our proof that \( \text{strDEEP} \nsubseteq \text{rec-wkDEEP} \) uses Baire category and Banach-
Mazur games. It is known that the set of strongly deep sequences is small in the sense of Baire category.

**Theorem 3.26** (Juedes, Lathrop, and Lutz [38]). The class strDEEP is meager.

We show that rec-wkDEEP \( \not\subseteq \) strDEEP by showing that rec-wkDEEP is comeager. Our proof of this fact is somewhat more involved than the proof by Juedes, Lathrop, and Lutz [38] that wkDEEP is comeager.

**Theorem 3.27.** For each uniform reducibility \( F \), the class rec-\( F \)-deep is rec-comeager, hence comeager in REC.

**Proof.** Let \( F \) be a uniform reducibility. For each \( n \in \mathbb{Z}^+ \), let \( a(n) = \frac{1}{6}n(n - 1)(2n - 1) \), so that \( a(n) + n^2 = a(n + 1) \). For each \( n \in \mathbb{Z}^+ \) and \( 0 \leq k < n \), let

\[
I_n(k) = \left\{ a(n) + kn + m \mid 0 \leq m < n \right\}.
\]

Note that the intervals

\[
I_1(0), I_2(0), I_2(1), I_3(0), I_3(1), I_4(2), I_4(0), \ldots
\]

partition \( \mathbb{N} \) into successive blocks, with each \( |I_n(k)| = n \).
For each $n \in \mathbb{Z}^+$, $0 \leq k < n$, $x \in \{0, 1\}^\leq n$, and $A \in \mathcal{C}$, say that $A$ agrees with $x$ on $I_n(k)$ if

$$A[a(n) + kn \ldots a(n) + kn + |x| - 1] = x.$$  

For each $n \in \mathbb{Z}^+$, $0 \leq k < n$, and $x \in \{0, 1\}^\leq n$, define the event

$$E_{k,n,x} = \{B \in \mathcal{C} \mid F_k(B) \text{ agrees with } x \text{ on } I_n(k)\}.$$  

For each $n \in \mathbb{Z}^+$ and $0 \leq k < n$, let $y_n(k)$ be the $n$-bit string whose $l^{th}$ bit is defined by the recursion

$$y_n(k)[l] = \left[\Pr(E_{k,n,z1}) < \Pr(E_{k,n,z0})\right]$$

for all $0 \leq l < n$, where $z = y_n(k)[0..l-1]$. This definition ensures that

$$\Pr(E_{k,n,y_n(k)[0..l]}) \leq \frac{1}{2}\Pr(E_{k,n,y_n(k)[0..l-1]}). \quad (3.14)$$

For each $n \in \mathbb{Z}^+$ and $0 \leq k < n$, define the event

$$E_{k,n} = E_{k,n,y_n(k)}.$$
Then, by (3.14), for all \( n \in \mathbb{Z}^+ \) and \( 0 \leq k < n \),

\[
\Pr(\mathcal{E}_{k,n}) \leq 2^{-n}.
\]  

(3.15)

Let

\[
Y = \left\{ A \in C \mid (\forall k)(\exists n) A \text{ agrees with } y_n(k) \text{ on } f_n(k) \right\}.
\]

It suffices to prove that

\[
Y \subseteq \text{rec-}F\text{-DEEP}
\]  

(3.16)

and

\[
Y \text{ is rec-comeager.}
\]  

(3.17)

We first prove (3.16). For each \( k, n \in \mathbb{N} \), define the function \( d_{k,n} : \{0,1\}^* \rightarrow [0,1] \) by

\[
d_{k,n}(w) = \begin{cases} 
\Pr(\mathcal{E}_{k,n} \mid C_w) & \text{if } 0 \leq k < n \\
0 & \text{otherwise}.
\end{cases}
\]
It is easy to check that each \(d_{k,n}\) is a martingale, and that the function \((k, n, w) \mapsto d_{k,n}(w)\) is total recursive (with rational values). Also, by (3.15),

\[
d_{k,n}(\lambda) \leq 2^{-n}
\]

for all \(k, n \in \mathbb{N}\). It follows by Theorem 2.33 that

\[
\mu_{\text{rec}} \left( \bigcup_{k=0}^{\infty} \bigcap_{m=0}^{\infty} \bigcup_{n=m}^{\infty} S^1[d_{k,n}] \right) = 0.
\]

To prove (3.16), let \(A \in Y\). Let \(B \in F^{-1}(A)\). Fix \(k \in \mathbb{N}\) such that \(A = F_k(B)\). Since \(A \in Y\), the set

\[
J_k = \{ n > k \mid A \text{ agrees with } y_n(k) \text{ on } I_n(k) \}
\]

is infinite. Let \(n \in J_k\). Then \(B \in \mathcal{E}_{k,n}\). In fact, since \(F_k\) is a tt-reduction, there is a prefix \(w \subseteq B\) such that \(C_w \subseteq \mathcal{E}_{k,n}\). Then \(d_{k,n}(w) = \Pr(\mathcal{E}_{k,n}|C_w) = 1\), so \(B \in S^1[d_{k,n}]\). Since \(J_k\) is infinite, this argument shows that

\[
F^{-1}(A) \subseteq \bigcup_{k=0}^{\infty} \bigcap_{m=0}^{\infty} \bigcup_{n=m}^{\infty} S^1[d_{k,n}].
\]

It follows from (3.19) and (3.20) that \(\mu_{\text{rec}}(F^{-1}(A)) = 0\), i.e., that \(A \in \text{rec-}F^{-1}\).
Finally, to prove (3.17), define a strategy $\beta$ for player II in the Banach-Mazur game $G[Y^c; \text{all; rec}]$ as follows. Given $w \in \{0,1\}^*$, fix the least $n \in \mathbb{Z}^+$ such that $a(n) \geq |w|$, and set

$$\beta(w) = w0^{a(n)-|w|}y_n(0)y_n(1)\cdots y_n(n-1).$$

It is clear that $\beta \in \text{rec}$ and, for every strategy $\alpha$ that player I might use, $R(\alpha, \beta) \in Y$. Hence, $\beta$ is a winning strategy for player II in $G[Y^c; \text{all.rec}]$. It follows that $Y^c$ is rec-meager, whence (3.17) holds. \hfill \Box

Theorem 3.28. The class $\text{rec-wkDEEP}$ is comeager.

Proof. The class $\text{rec-wkDEEP}$ is a countable intersection of classes $\text{rec-F-DEEP}$, each of which is rec-comeager, hence comeager, by Theorem 3.27. \hfill \Box

Corollary 3.29. $\text{rec-wkDEEP} \not\subset \text{strDEEP}$.

Proof. This follows immediately from Theorems 3.26 and 3.28. \hfill \Box

We now have the main result of this section.
Theorem 3.30. The following diagram of proper inclusions holds.

\[
\begin{array}{ccc}
\text{rec-wkDEEP} & \cup & \text{wkDEEP} \\
\cap & & \\
\text{rec-strDEEP} & & \text{strDEEP} \\
\cap & & \cup \\
\end{array}
\]

Proof. This follows immediately from Theorem 3.20, and Corollaries 3.25 and 3.29. □

By Theorem 3.30, there exist sequences that are strongly deep, but not rec-strongly deep. Let \( S \) be such a sequence. Since \( S \) is not rec-strongly deep, there exist a fixed computable time bound \( t_0 : \mathbb{N} \to \mathbb{N} \) and a fixed constant \( c_0 \in \mathbb{N} \) such that, for every computable time bound \( t : \mathbb{N} \to \mathbb{N} \), there are infinitely many prefixes of \( S \) that cannot be described \( c_0 \) bits more succinctly with the time bound \( t \) than with the time bound \( t_0 \). Nevertheless, since \( S \) is strongly deep, it must be the case that, for every constant \( c \in \mathbb{N} \) (even when \( c \) is much greater than \( c_0 \)), all but finitely many prefixes of \( S \) can be described at least \( c \) bits more succinctly without a time bound than with the time bound \( t_0 \). None of this additional succinctness (beyond \( c_0 \) bits) can be realized within any computable time bound; all of it
requires greater-than-computable running time. The depth of such a sequence $S$ appears not to come from so much from a nontrivial causal (computational) history as from something utterly noncomputational.

If $F$ is a uniform reducibility that is (like all standard reducibilities) reflexive, then the measure and category of the class rec-$F$-DEEP are of some interest. First, rec-$F$-DEEP must be disjoint from RAND(rec), so rec-$F$-DEEP must be a measure 0 subset of $C$. Also, by Theorem 3.27, rec-$F$-DEEP must be comeager. Thus, the class rec-$F$-DEEP is small in the sense of measure, but large in the sense of Baire category. This state of affairs is not unusual and would not be worth mention, were it not for the fact that the situation changes when we look at the measure and category of rec-$F$-DEEP in REC. By Theorems 2.68 and 3.27, rec-$F$-DEEP is large in REC in the senses of both measure and category. The class rec-$F$-DEEP is thus one concerning which measure and category agree in REC, but disagree in $C$.

3.4 Weakly Useful Sequences

Juedes, Lathrop, and Lutz [38] defined the class of weakly useful sequences and proved that every weakly useful sequence is strongly deep. Fenner, Lutz, and Mayordomo [19] subsequently proved that every weakly useful sequence is rec-weakly deep. In this section, we strengthen both these results by proving that
every weakly useful sequence is rec-strongly deep. Our argument closely follows that of [38], but it is short and central, so we present it in full.

**Definition** (Juedes, Lathrop, and Lutz [38]). A sequence $A \in C$ is strongly useful, and we write $A \in \text{strUSEFUL}$, if there is a computable time bound $A : \mathbb{N} \rightarrow \mathbb{N}$ such that $\text{REC} \subseteq \text{DTIME}^{A}(s)$. A sequence $A \in C$ is weakly useful, and we write $A \in \text{wkUSEFUL}$, if there is a computable time bound $A : \mathbb{N} \rightarrow \mathbb{N}$ such that $\text{DTIME}^{A}(s)$ does not have measure 0 in $\text{REC}$.

Thus a sequence is strongly useful if it enables one to solve all decidable sequences in some fixed, computable amount of time. A sequence is weakly useful if it enables one to solve all elements of a nonnegligible set of decidable sequences in some fixed, computable amount of time.

Recall that the diagonal halting problem is the sequence $K$ whose $n^{\text{th}}$ bit is

$$K[n] = [M_n(n) \text{ halts}],$$

where $M_0, M_1, \ldots$ is a standard enumeration of all deterministic Turing machines.

It is well-known that $K$ is polynomial-time many-one complete for the set of all recursively enumerable subsets of $\mathbb{N}$, so $K$ is strongly useful.

It is clear that every strongly useful sequence is weakly useful. Fenner, Lutz, and Mayordomo [19] used martingale diagonalization to construct a sequence that
is weakly useful but not strongly useful, so \( \text{strUSEFUL} \nsubseteq \text{wkUSEFUL} \).

Our proof that every weakly useful sequence is strongly deep uses the following theorem, which is a recursive strengthening of Theorem 5.8 of [38]. Recall the class \( K^{t}_{i.o.}[< g(n)] \) defined in section 2.28.

**Theorem 3.31.** If \( t : \mathbb{N} \rightarrow \mathbb{N} \) is computable and \( 0 < \alpha < \beta < 1 \), then

\[
\text{REC} \subseteq \hat{D}^{t}_{\text{rec}} \cup K^{t}_{i.o.}[< \beta n].
\]

**Proof.** Assume the hypothesis and let

\[
S \in \text{REC} - K^{t}_{i.o.}[< \beta n].
\]

We will show that \( S \in \hat{D}^{t}_{\text{rec}} \).

Since \( S \not\in K^{t}_{i.o.}[< \beta n] \), it must be the case that, for all but finitely many \( n \),

\[
K^{t}(S[0..n-1]) \geq \beta n.
\]

Since \( S \) is recursive, there is a Turing machine \( M' \) such that, for all \( n \in \mathbb{N} \),

\[
M'(sd(s_n)) = S[0..n-1], \text{ where } sd(s_n) \text{ is the self-delimiting version of } s_n, \text{ the } n^{\text{th}} \text{ string in the standard enumeration of } \{0,1\}^{*}.
\]
Let \( M' \) be a Turing machine that on input \( \pi \) performs the following computation.

\[
\text{begin } M'(\pi) \\
\quad \text{if } \pi \text{ not of the form } sd(n) \text{ then halt} \\
\quad \text{decode } n = sd^{-1}(\pi). \\
\quad \text{for } i = 0 \text{ to } n - 1 \text{ do} \\
\quad \quad \text{output } M(i) \text{ (Note that } M \text{ is hard coded into this machine.)} \\
\quad \text{halt} \\
\text{end}
\]

It is clear that \( M' \) is recursive, and that for all \( n \in \mathbb{N} \), \( M'(sd(s_n)) = S[0..n-1] \).

Now let \( \pi_M' \) be a program prefix for \( U \) such that for all \( \pi \in \{0,1\}^* \).

\[
U(\pi_M', \pi) = M'(\pi).
\]

In particular, we have

\[
U(\pi_M', sd(s_n)) = M'(sd(s_n)) = S[0..n-1].
\]

Let \( l : \mathbb{N} \to \mathbb{N} \) give the running time of \( U \) on these programs, i.e.,

\[
l(n) = time_U(\pi_M', sd(s_n)).
\]
Then \( l \) is computable and, for all but finitely many \( n \in \mathbb{N} \),

\[
K^l(S[0..n - 1]) \leq |\pi_{M'} sd(s_n)|
\]

\[
= 2[\log(n + 1)] + 2 + |\pi_{M'}|
\]

\[
< \beta n - \alpha n
\]

\[
\leq K^l(S[0..n - 1]) - \alpha n,
\]

so \( S \in \tilde{D}^{t, \text{rec}}_{\alpha n} \).

Corollary 3.32. For every computable time bound \( t : \mathbb{N} \to \mathbb{N} \) and every \( 0 < \gamma < 1 \),

\[
\mu \left( D_{\gamma n}^{t, \text{rec}} | \text{REC} \right) = 1.
\]

Proof. Let \( t : \mathbb{N} \to \mathbb{N} \) be computable, and let \( 0 < \gamma < \alpha < \beta < 1 \). Choose a computable time bound \( t_1 : \mathbb{N} \to \mathbb{N} \) for \( t \) and constants \( c_1, c_2 \in \mathbb{N} \) as in Lemma 3.10, so that for all \( n \in \mathbb{N} \),

\[
\tilde{D}_{\gamma n + c_2 + c_1}^{t_1, \text{rec}}(n) \subseteq \tilde{D}_{\gamma n + c_2}^{t_1, \text{rec}}(n) \subseteq D_{\gamma n}^{t_1, \text{rec}}(n).
\]
For all sufficiently large $n$, we have

$$\widehat{D}_{\alpha n}^{t_1, \text{rec}}(n) \subseteq \widehat{D}_{\gamma n + c_2 + c_1}^{t_1, \text{rec}}(n),$$

so $\widehat{D}_{\alpha n}^{t_1, \text{rec}} \subseteq D_{\gamma n}^{t, \text{rec}}$.

By Theorem 2.29 $K_{1, \text{o}}^{t_1}(< \beta n)$ has measure 0 in REC. Combined with Theorem 3.31, this implies that $\widehat{D}_{\alpha n}^{t_1, \text{rec}}$ has measure 1 in REC. Since $\widehat{D}_{\alpha n}^{t_1, \text{rec}} \subseteq D_{\gamma n}^{t, \text{rec}}$, it follows that $D_{\gamma n}^{t, \text{rec}}$ has measure 1 in REC. \hfill \Box

**Corollary 3.33.** For every computable time bound $t : \mathbb{N} \to \mathbb{N}$ and every constant $c \in \mathbb{N}$,

$$\mu\left(D_{c, \text{rec}}^{t, \text{rec}} \mid \text{REC}\right) = 1.$$

We now demonstrate the rec-strong depth of weakly useful sequences.

**Theorem 3.34.** Every weakly useful sequence is rec-strongly deep.

**Proof.** Let $A \in C$ be weakly useful. To see that $A$ is rec-strongly deep, let $t : \mathbb{N} \to \mathbb{N}$ be an arbitrary computable time bound, and let $c \in \mathbb{N}$ be arbitrary. It suffices to show that $A \in D_{c}^{t, \text{rec}}$. 

Since \( \mathcal{A} \) is weakly useful, there is a computable time bound \( s : \mathbb{N} \rightarrow \mathbb{N} \) such that \( \text{DTIME}^\mathcal{A}(s) \) does not have measure 0 in \( \text{REC} \). Since every computable function is bounded above by a strictly increasing, time-constructible function, we can assume without loss of generality that \( s \) is strictly increasing and time-constructible.

Let \( \tilde{t}(n) = n \cdot (1 + \tau(n)[\log \tau(n)]) \), where \( \tau \) is defined from \( t \) and \( s \) as in Lemma 3.18, and let \( \gamma = \frac{1}{s} \). Since \( \tilde{t} \) is recursive, Corollary 3.33 tells us that \( D^{\tilde{t},\text{rec}}_{\gamma n} \) has measure 1 in \( \text{REC} \). Since \( \text{DTIME}^\mathcal{A}(s) \) does not have measure 0 in \( \text{REC} \), it follows that \( D^{\tilde{t},\text{rec}}_{\gamma n} \cap \text{DTIME}^\mathcal{A}(s) \neq \emptyset \). Fix a sequence \( B \in D^{\tilde{t},\text{rec}}_{\gamma n} \cap \text{DTIME}^\mathcal{A}(s) \). Then there is an \( s \)-time-bounded oracle Turing machine \( M \) such that \( A \cup B \) via \( M \).

Fix a constant \( c_M \) as in Lemma 3.18. Define \( g(n) = c \) for all \( n \in \mathbb{N} \) and define the functions \( \tau, \tilde{t}, \text{and } g \) from \( t \) and \( g \) as in Lemma 3.18. Since \( \tilde{g} \) and \( c_M \) are constant, we have \( \tilde{t}(n) > \tilde{t}(n) \) and \( \gamma n > \tilde{g}(n) \) for all but finitely many \( n \), so \( B \in D^{\tilde{t},\text{rec}}_{\gamma n} \subseteq D^{\tilde{t},\text{rec}}_{\tilde{g}} \). It follows by Lemma 3.18 that \( A \in D^{\tilde{t},\text{rec}}_{c} \).

Juedes [37] asked whether every strongly deep sequence is weakly useful. We can now answer this question negatively.

**Corollary 3.35.** \( \text{wkUSEFUL} \nsubseteq \text{strDEEP} \)

**Proof.** This follows immediately from Theorems 3.30 and 3.34. \( \square \)
CHAPTER 4. COMPRESSION DEPTH

Compression depth is a computable complexity measure that provides a measure of the amount of structure (organization) in (finite) binary strings. Motivated by Bennett’s notion of computational depth [7, 8], compression depth is based on well-known compression algorithms that quickly compress data. While any lossless compression algorithm with the property that it can be parameterized may be used to define a compression depth complexity measure, this section focuses on the well-understood Lempel-Ziv compression algorithm, and thereby defines the LZ-compression depth of strings.

We further develop LZ-compression depth, extend the notion to (infinite) sequences, and define compression depth classes DLZ^L analogous to the depth classes D^L defined in section 2. Using this definition of compression depth, we then define strong compression depth. Using LZ-compression we define LZ random sequences and show that this notion of randomness is stronger than normality, a standard randomness criterion. We then show that a sequence that is strongly LZ deep cannot be LZ random. This result is joint work with Martin Strauss. Similarly, we define a notion of LZ simplicity and show that these “simple” sequences are also not strongly LZ deep.

We begin by formally defining compression depth and LZ compression depth.
Note that in this chapter we assume that strings and sequences are over the binary alphabet \{0, 1\}

**Definition 4.1.** A parameterized compression algorithm is an algorithm \(A\) that maps \(\mathbb{N} \times \{0, 1\}^*\) into \(\{0, 1\}^*\). For \(t \in \mathbb{N}\) and \(x \in \{0, 1\}^*\), \(A(t, x)\) is said to be the \(t\)-resource compression of \(x\), where \(t\) specifies the amount of resources available to the parameterized compression algorithm. (Note that the resource is not necessarily time.)

**Definition 4.2.** If \(A\) is a parameterized compression algorithm and \(t \in \mathbb{N}\), then the \(t\)-resource compression complexity of a string \(x \in \{0, 1\}^*\) is

\[
C_A^t(x) = \min \left\{ |A(q, x)| \middle| 0 \leq q \leq t \right\}
\]

(The minimum is taken in order to force \(C_A^t(x)\) to be nonincreasing in \(t\).)

**Definition 4.3.** The compression complexity of a string \(x \in \{0, 1\}^*\) relative to a parameterized compression algorithm \(A\) is

\[
C_A(x) = \min \left\{ |A(q, x)| \middle| q \geq 0 \right\}.
\]

**Definition 4.4.** Let \(A\) be a parameterized compression algorithm, and let \(s \in \mathbb{N}\).
The compression depth of the string $x$ at significance level $s$ is

$$\text{Cdepth}^s(x) = \max \{ t \mid C_A(x) \leq C_A^t(x) - s \},$$

where $\max 0 = 0$.

For any given significance level $s$ and parameterized compression algorithm $A$, a string $x$ is called $t$-compression-deep relative to $A$ at significance level $s$ if $\text{Cdepth}^s(x) \geq t$. Otherwise, $x$ is $t$-compression-shallow relative to $A$ at significance level $s$.

Since the compression algorithm is parameterized by $t$, the compression depth can be viewed graphically in the same manner as the computational depth. In Figure 5, the relationship between the compression depth of a string $x$ and the significance parameter is shown by plotting $C_A^t(x)$ versus $t$. Intuitively, a string has a large compression depth if, as more resources are allowed, the compression algorithms utilizes these resources to find more subtle redundancy and further compress the string.

We extend these definitions to infinite sequences as follows.

**Definition 4.5.** The $t$-resource compression ratio of a sequence $S \in \mathbb{C}$ relative
to a parameterized compression algorithm $A$ is

$$\rho_A(S) = \limsup_{n \to \infty} \left( \frac{C_A(S[0..n-1])}{n} \right).$$

**Definition 4.6.** The *compression ratio* of a sequence $S \in C$ relative to a parameterized compression algorithm $A$ is

$$\rho_A(S) = \limsup_{n \to \infty} \left( \frac{C_A(S[0..n-1])}{n} \right).$$

Figure 5: Graphical view of compression depth for a hypothetical parameterized compression algorithm $A$ and string $x$. 
There are many compression algorithms used to compress data. However, not all of them are suitable for use as a method for computing a compression depth. Particular properties must be present in a compression algorithm in order for it to be useful for computing compression depth. For the properties listed below, let $x$ be a string and let $A$ be a compression algorithm.

1. There must be a useful parameterized version of $A$.

2. The compression must be lossless. That is, there must exist a decompression algorithm $B$, such that, for all $t$, $B(A(t, x), t_1) = x$, where $t_1$ is the running-time of the decompression algorithm.

3. For all $t$, $A(t, x)$ must be feasibly computable.

There are a variety of compression algorithms that are used for many purposes. By evaluating these algorithms in terms of the requirements stated above, a suitable compression algorithm may be found that can be used to define the compression depth of strings. Note that requirement (2) above eliminates many sound and video compression algorithms. These algorithms often discard information in order to achieve more compression, resulting in a somewhat degraded recording image after decompression. Unfortunately, the information ignored in the compression process for these algorithms often forms the very structure that makes strings deep. Thus, these types of algorithms are unsuitable for generating
a depth measurement as described here.

Run-length encoding and Huffman encoding are both compression algorithms that do not yield a depth-like measurement, each for a different reason. Run-length encoding is a simple compression algorithm designed to compress picture data by encoding a long string of zeros or ones as a special code followed by the number of zeros or ones. However, this algorithm does not compress simple strings such as \((01)^n\). Therefore any parameterization of this compression algorithm is inadequate for the purpose of depth measurement.

Huffman encoding compresses data by using either the probability distribution or an approximation to the probability distribution over a fixed block size, and then exploiting strings with high probability to achieve compression. This technique does not yield a good depth measure for two reasons. First, the natural parameterization of the Huffman compression algorithm is block size. However, the string \((01101)^n\) will achieve much better compression with block sizes that are multiples of 10. It is desirable that compression not fluctuate greatly with small increments of resource. Secondly, unless the probability is agreed upon in advance, the encoder must also store the string substitution table with the compressed data in order for it to be decompressed. This can be very large, obscuring any compression of the string. (Note that for infinite sequences, a large constant-size table that prefixes the compressed string is inconsequential to the compression ratio.)
4.1 Lempel-Ziv Compression

Lempel-Ziv compression, first introduced by Lempel and Ziv [48], provides a good and efficient compression algorithm that can be parameterized without suffering from the blocking effects associated with Huffman encoding. Many variations of this original algorithm have since been introduced that run faster and with better compression. However, these improvements are small and the asymptotic performance of these algorithms is no better than the original Lempel-Ziv algorithm [48, 15].

In this thesis we utilize the original Lempel-Ziv (LZ) algorithm for simplicity. This section describes the original algorithm and gives two examples. A careful description of a new parameterized compression algorithm based on the original Lempel-Ziv algorithm that yields a good notion of compression depth follows. Finally, examples of compression depth using the modified Lempel-Ziv algorithm are illustrated using binary strings of various depths.

The following definitions are useful for defining the original Lempel-Ziv algorithm, as well as the parameterized version defined later in this section.

**Definition 4.7.** The prefix set of a string \( x \in \{0, 1\}^* \) is the set \( X = \{ y \mid y \subseteq x \} \).

**Definition 4.8.** A valid code is a set \( X \subseteq \{0, 1\}^* \) such that, for all \( x \in X \), the prefix set of \( x \) is a subset of \( X \).
Definition 4.9. A parsing of a string $x \in \{0,1\}^*$ is a partition of the string $x$ into phrases $x_1, x_2, x_3, \ldots, x_n$ such that $x_1 \cdot x_2 \cdot x_3 \cdot \ldots \cdot x_n = x$.

Definition 4.10 (Cover and Thomas [15]). A distinct parsing of a string $x \in \{0,1\}^*$ is a parsing of $x$ such that no phrase, except possibly the last phrase, is the same as an earlier phrase.

Definition 4.11. A valid distinct parsing of a string $x \in \{0,1\}^*$ is a distinct parsing of $x$ such that if $x_i$ is a phrase in the string $x$, then every prefix $y$ of $x_i$ appears before $x_i$ in the distinct parsing.

It is clear that every string $x$ has a unique valid distinct parsing and that the set of phrases in this valid distinct parsing is a valid code. This is also illustrated graphically in Figure 6.

1 11 0 10 00 111 001 1110

Figure 6: The valid distinct parsing of the string $x = 1110100111001110$.

The Lempel-Ziv compression algorithm uses the valid distinct parsing of a string to encode it by replacing each phrase with a code word representing a pointer and a bit. In this scheme, the pointer indicates the longest proper prefix of the phrase, and the bit is simply the last bit of the phrase. Together, these completely specify the phrase being encoded. Because every prefix word of a
phrase must also be a phrase that occurs earlier in the distinct parsing, the distinct parsing shown in Figure 6 can be augmented with arrows to show these pointer-pair codes as depicted in Figure 7. By assigning an address to each parse phrase, beginning at address 1, the pairs of pointers and bits are coded in binary to yield a final compressed string as illustrated in Figure 8.

A graph of the Lempel-Ziv compression lengths for $0^n$, for various values of $n$, is shown in Figure 9. This figure also shows that the strings $(00000000)^n$, $(00000001)^n$ and $(10101010)^n$ are also highly compressible. On the other hand, this figure also shows that a string chosen randomly according to the uniform distribution is not compressible. Thus, the Lempel-Ziv compression algorithm exhibits all the key properties required for defining a compression depth algorithm, providing it can be parameterized.

Figure 7: Example of valid distinct parsing with pointers
A Parameterized Lempel-Ziv Compression Algorithm

The Lempel-Ziv algorithm described above provides reasonable compression with modest computational requirements, but it also offers a natural parameterization. By restricting the number of phrases used from the distinct valid parsing, we can “cripple” the Lempel-Ziv algorithm, limiting its ability to compress data. If this restriction of the valid distinct parsing is performed properly, then simple strings such as 0^n compress to near-maximal even when the valid distinct parsing is so severely limited. Thus, the size of the limited valid distinct parsing forms the basis for a parameterized Lempel-Ziv compression algorithm and, ultimately, a measure of compression depth.

![Diagram of Lempel-Ziv compression]

Figure 8: Example of Lempel-Ziv compression.
To simplify the exposition (and implementation) of the parameterized Lempel-Ziv algorithm, we define a dictionary as a rooted binary tree used to define a set of strings. Each non-root node represents a nonempty string, corresponding to the path from the root node to that node. A left branch represents a zero bit and right branch represents a one bit. Figure 10 shows an example of a dictionary and the set of strings it represents.

The tree structure of the dictionary can be used to implement both the original Lempel-Ziv algorithm and a parameterized version. In the original algorithm, the dictionary represents the phrases seen so far as the input is scanned. By traversing

Figure 9: Lempel-Ziv compression of periodic and random strings.
the tree as each bit of the input is read, the next phrase in the string is determined. When this traversal leads to a leaf node, the next bit determines the parse point and a new leaf is appropriately added. This process is illustrated in Figure 11.

We parameterize the Lempel-Ziv algorithm by restricting the size of the dictionary. This is accomplished by only allowing the parameterized algorithm to add new strings to the dictionary when they are also in a master dictionary. Since the parameterized Lempel-Ziv algorithm may only add strings that are also in the master dictionary, the dictionary built by the parameterized Lempel-Ziv algorithm is bounded in size and structure by the master dictionary. Thus, by adding strings to the master dictionary, we increase a resource for compression, thereby giving a method for computing compression depth based on the Lempel-Ziv algorithm.

Figure 10: A dictionary and the set of strings it represents
Figure 11: Using a tree (dictionary) to generate the distinct parsing of a string

The process of parsing a string given a master dictionary is illustrated in Figure 12. The parse tree is obtained by labeling the node of the master dictionary with non-negative integers. Initially, all nodes are labeled 0. This label is then used to indicate whether the string represented by the node has been used in the parse. A non-zero label indicates which phrase in the parse the node represents. The label associated with the root node is always zero and meaningless. The parsing is performed in the same manner as the normal Lempel-Ziv algorithm except that only strings in the master dictionary may be added to the parse tree. (Note that the master dictionary must be at least of size 3 containing at least strings “0” and “1”. This is the smallest resource bound possible.)

In the example shown in Figure 12, the first bit (a one) is read and the right branch (corresponding to reading a one) of the root node is examined. If there is no right branch, or the right branch is labeled with a zero (as in this example), then a phrase has been found. The node corresponding to the phrase found (in this case the phrase is the single bit 1) is then labeled with a 1 to indicate that
it is the first phrase found. The process then repeats, starting with the next bit of input and at the root of the tree. The next bit is read (a one), and again the right branch of the root node is examined. However, in this case the node is now labeled with a 1, indicating that the string it represents occurred earlier in the parse. Thus, the next bit of the input is read (a one), and the right branch of this node is now examined. This node is labeled 0, and thus the input is parsed with the phrase “11.” This new node is then marked with a 2, indicating it is the second new phrase in the parsing. The process continues until the entire input is consumed as shown in Figure 13.

In the above example, a key situation occurs on the fifth, seventh and eighth strings parsed. These phrases are parsed because there were no left or right nodes to examine in the master dictionary. For example, in the fifth phrase, a zero bit is read and the node labeled 3 is examined. The next bit is read (a zero)

```
1110... 1110... 1110... 1110...
```

Figure 12: Process for parsing a string given a master dictionary
and the node labeled 3 does not have a left branch. At this point the phrase is parsed as the string 00, but is not added to the parse dictionary since there is no node to label. This is exactly the mechanism by which we "cripple" the original Lempel-Ziv algorithm to yield a parameterized version. Note that this procedure no longer parses the input into distinct phrases; however, the same Lempel-Ziv decompression algorithm may be used to retrieve the original string from the compressed string.

Figure 13: The parsing of a complete string given a master dictionary
If the master dictionary is the same as the dictionary produced when the string is parsed with the original Lempel-Ziv algorithm, it is easy to see that the parameterized Lempel-Ziv algorithm gives the same parsing and compression as the original algorithm. In addition, any extension of a master tree of this form will also give an identical parse to the original Lempel-Ziv algorithm. Thus, as the resource is level is increased, the compression of the string tends towards the original Lempel-Ziv compression. This is shown in Figure 14.

In order to compute a compression depth measurement, several compression

1 11 0 10 00 111 001 1110

Figure 14: Example showing a master dictionary
values must be computed with various amounts of resource. Since the compression depth measurement requires that the resource be measured by a number, we define the amount of resource to be the size (number of nodes) of the master dictionary. However, an efficient method for determining the structure of the tree at each size remains to be addressed. Ideally, the algorithm to compute the Lempel-Ziv compression depth at resource level $n$ would evaluate the compression of the string for every master dictionary of size $n$. However, this is computationally infeasible. Here, we use a recursive algorithm based on the master tree of size $n - 1$ to compute the master tree of size $n$.

As shown in Figure 15, the master tree of size $n - 1$ is extended at each node having fewer than two successors by adding each possible successor, one at a time for the entire tree. The parsing algorithm defined above is executed, and the number of times the new node is referenced in the parse is counted. This is computed for each possible new node, corresponding to each possible single legal phrase that could be added to the master dictionary. The master dictionary is then extended by the node that is referenced the maximum number of times among the candidate new nodes. Roughly, this procedure chooses to extend the master dictionary by a string which extends one of the strings currently in the dictionary by one bit and occurs the most frequently in the string to be parsed. This gives a very fast computation of the entire compression depth graph since
each time the master dictionary is increased by one, only a linear number of new strings (nodes) require their frequencies to be computed.

4.2 Compression Depth and LZ Depth

We now define strongly LZ compression deep sequences in a manner similar to the way we defined computationally deep sequences. Intuitively, a sequence is strongly compression deep if the prefixes of the sequence require more than a "minimal" amount of resources to find ever increasing "buried" redundancy in the sequence. This means that a simple sequence is either nearly random with respect to the LZ compression algorithm, in which case no number of entries in the dictionary will compress the sequence; or very redundant with respect to the LZ compression algorithm, in which case optimal compression can be achieved.

Figure 15: Increasing the size of the master dictionary
with a constant number of dictionary entries per bit of input.

Motivated by the definition of computational depth, we define LZ compression depth classes as follows.

**Definition 4.12.** For \( t, g : \mathbb{N} \to \mathbb{N} \) and \( n \in \mathbb{N} \), define the sets

\[
DLZ_t(n) = \{ S \in C \mid Cdepth_{g(n)}^{LZ}(S[0..n-1]) > t(n) \}
\]

and

\[
DLZ_t^s = \{ S \in C \mid S \in DLZ_t(n) \text{ a.e.} \}.
\]

We also define strong LZ compression depth analogously.

**Definition 4.13.** A sequence \( S \) is **strongly LZ compression deep** if for all \( c, s \in \mathbb{N} \)

\[
Cdepth_{s}^{LZ}(S[0..n-1]) \geq cn \text{ a.e.}
\]

With these definitions, it is easy to see that for \( c, s, \bar{c}, \bar{s} \in \mathbb{N} \), with \( \bar{c} \geq c \) and \( \bar{s} \geq s \), \( DLZ_{\bar{s}}^{\bar{c}} \subseteq DLZ_{s}^{c} \). This gives us Figure 16, which is analogous to Figure 3.

We now show that this notion behaves analogously to computational depth and recursive computational depth in the sense that, for suitable notions of randomness and simplicity, no strongly LZ deep sequence is random or simple. We begin by
defining two natural notions of simplicity based on repeated patterns, and proving that they are equivalent. We then define a notion of LZ simplicity in terms of the size of the LZ dictionary generated by a sequence, and show that the class of LZ simple sequences properly contains the class of sequences defined by the two equivalent natural definitions.

**Definition 4.14.** A sequence $S \in C$ is periodic, and we write $S \in \text{PERIODIC},$
if there exist strings \( u, v \in \{0, 1\}^* \) such that

\[ S = uv^\infty. \]

Recall that FST is the set of all finite-state transducers.

**Definition 4.15.** A sequence \( S \in \mathbb{C} \) is *FST-simple*, and we write \( S \in \text{FSTSIMPLE} \), if there exists \( F \in \text{FST} \) such that for all \( n \in \mathbb{N} \) there exists \( m \in \mathbb{N} \) such that

\[ S[0..n - 1] \subseteq F(0^m). \]

Definitions 4.14 and 4.15 are natural in the sense that Definition 4.14 defines the set of sequences that are ultimately a repetition of some string. Definition 4.15 is the sequence output from a finite-state transducer when driven by the input \( 0^\infty \). It is well-known and easy to show that these two notions of simple are the same, i.e., \( \text{PERIODIC} = \text{FSTSIMPLE} \).

We now define a notion of simplicity by measuring the amount of compression, (i.e., the rate at which the dictionary grows) as the sequence is parsed by the LZ compression algorithm. We then show that it properly contains the class
PERIODIC. We need the following definition which gives us the "width" of each level of the parse tree after a string has been parsed by the LZ compression algorithm.

**Definition 4.16.** Given a string $x \in \{0,1\}^*$, define the function $\text{width}_{LZ} : \{0,1\}^* \times \mathbb{N} \to \mathbb{N}$, the width of the LZ parse tree of $x$, by

$$
\text{width}_{LZ}(x, n) = \text{the number of strings of length } n \text{ in the dictionary } T 
$$

after the string $x$ is parsed.

**Definition 4.17.** Given a sequence $S \in \mathbb{C}$, define the function $\text{seqw}_{LZ} : \mathbb{C} \times \mathbb{N} \to \mathbb{N}$, the sequence width of the LZ parse tree of $S$, by

$$
\text{seqw}_{LZ}(S, n) = \lim_{m \to \infty} \text{width}_{LZ}(S[0..m - 1], n). 
$$

We can now define the set of simple sequences based on how fast the dictionary grows. Intuitively, a sequence is LZ simple if, as the infinite sequence is parsed by the LZ algorithm, the width of the dictionary is bounded by a constant. This means that the prefixes of the sequence compress to within a constant factor of optimal.

**Definition 4.18.** A sequence $S \in \mathbb{C}$ is LZ simple and we write $S \in \text{LZSIMPLE}$
if there exists \( c \in \mathbb{N} \) such that for all \( n \in \mathbb{N} \)

\[
\operatorname{seq}_{LZ}(S, n) \leq c.
\]

We now show that \( \text{PERIODIC} \nsubseteq \text{LZSIMPLE} \). We first show that every periodic sequence is LZ simple. We then exhibit a string that is LZ simple but not periodic to prove that the containment is proper. We need the following definition.

**Definition 4.19** ([56]). For a string \( x \in \{0, 1\}^* \), define the function \( \text{necklace} : \{0, 1\}^* \rightarrow \mathcal{P}((0, 1)^*) \) by

\[
\text{necklace}(x) = \left\{ (x \cdot x)[i..i + |x| - 1] \mid 0 \leq i < |x| \right\}.
\]

For example, \( \text{necklace}(0100) = \{0100, 1000, 0001, 0010\} \).

We now show that if a sequence is periodic, then the LZ dictionary generated by the sequence is bounded by a constant, and hence \( \text{PERIODIC} \nsubseteq \text{LZSIMPLE} \).

**Lemma 4.20.** If \( S \in \text{PERIODIC} \), then there exists a constant \( c \in \mathbb{N} \) such that for all \( n \in \mathbb{N} \),

\[
\operatorname{seq}_{LZ}(S, n) \leq c.
\]
Proof. Let $S = uv^\infty$ be periodic. Let $z$ be the shortest prefix of $S$ such that $u \subseteq z$ and $S[z..\infty]$ begins a new phrase in the parsing of $S$. Let $T$ be the dictionary after the LZ algorithm has parsed $z$. If suffices to show that, for each $n \in \mathbb{N}$, the subsequent parsing of $S$ (beginning after $z$) produces at most $|v|$ many phrases of length $n$, since we can then take $c = |T| + |v|$ in the statement of the lemma.

To show this, let $x$ be a phrase that is produced in the subsequent parsing of $S$. Then there is some $w \in \text{necklace}(v)$ such that $x \subseteq w^\infty$. Since $|\text{necklace}(v)| \leq |v|$, at most $|v|$ such strings $x$ can have any given length $n$. \hfill \square

Note that if $S = v^\infty$ ($u = \lambda$), then for all $n \in \mathbb{N}$,

$$\text{seqw}(S, n) = |\text{necklace}(v)|.$$

**Theorem 4.21.** $\text{FSTSIMPLE} \subseteq \text{LZSIMPLE}$

**Proof.** First we show that every sequence $S \in \text{FSTSIMPLE}$ is also in $\text{LZSIMPLE}$.

Let $S \in \text{FSTSIMPLE}$. As we have noted, $S$ is periodic so by Lemma 4.20, the width of the dictionary is bounded as $S$ is parsed. Thus $S \in \text{LZSIMPLE}$.

To see that there is a sequence that is LZ-simple but not FST-simple, consider the sequence $f(1) \cdot f(2) \cdot f(3) \ldots$, where the function $f : \mathbb{N} \rightarrow \{0,1\}^*$ is defined
recursively as follows.

\[
f(n) = \begin{cases} 
1 & \text{if } n = 1 \\
(f(n - 1) \cdot n^{th} \text{ bit of the halting language}) & \text{if } n > 1 
\end{cases}
\]

It is clear that this sequence has \(\text{width}_{\text{LZ}}(S, n) = 1\) for all \(n\), yet is not periodic.

\(\square\)

We have now defined a class of simple strings under LZ compression. We now show that LZ compression can be used to define a notion of randomness that is more powerful (i.e. more restrictive) than standard definitions for random using the normality criteria described in [40].

### 4.3 Randomness and LZ Compression

We now define the notion of LZ randomness in terms of Lempel-Ziv compression. We then prove two theorems that show LZ randomness yields a stronger definition of randomness when compared to normal sequences. The first theorem says that every sequence that is LZ random must also be normal. The second, more surprising result, says that there are normal sequences that compress almost optimally under the LZ compression algorithm. In fact, Lathrop and Strauss have proven a stronger fact that states that there is a normal sequence \(S\) that parses optimally under the LZ compression algorithm, i.e., the width of the dictionary is
1 for all prefixes of $S$. We begin by defining the notion of LZ randomness.

**Definition 4.22.** A sequence $S$ is *LZ random* and we write $S \in \text{LZRAND}$ if there exists a constant $c \in \mathbb{N}$ such that for all $n \in \mathbb{N}$

$$C_{LZ}(S[0..n-1]) \geq n - c.$$

**Definition 4.23.** A sequence $S$ is *weakly LZ random* and we write $S \in \text{wkLZRAND}$ if

$$\lim_{n \to \infty} \frac{C_{LZ}(S[0..n-1])}{n} \geq 1.$$

It is clear from the above definitions that $$\text{LZRAND} \subset \text{wkLZRAND}.$$ We now turn our attention to proving that every weakly LZ random sequence is also normal, and that the converse is not true. To do this, we prove that if $S$ is not a normal sequence, then the LZ compression algorithm will compress $S$. We first prove three lemmas necessary for our proof.

**Definition 4.24.** A dictionary $T$ is *unskewed* if there do not exist $x \in T$ and $y \notin T$ such that $|x| > |y|$. 
Definition 4.25. The depth of a dictionary $T$, denoted $\text{depth}(T)$ is the length of the longest string in the dictionary, i.e.,

$$\text{depth}(T) = \max \{|x| \mid x \in T\}$$

Definition 4.26. A dictionary is complete if it is unskewed and contains $2^{\text{depth}(T)} - 2$ entries, not including the empty string.

Definition 4.27. The bitsize of a dictionary $T$, denoted $\text{bitsize}(T)$, is

$$\sum_{x \in T} |x|.$$ 

It is straightforward to show that if $T$ is a complete dictionary with $\text{depth}(T) = n$, then $\text{bitsize}(T) = (n - 1)2^{n+1} + 2.$

Lemma 4.28. Let $T$ and $T'$ be dictionaries, and let $s \in \mathbb{N}$. If $T$ is unskewed, $\text{bitsize}(T') \leq s$, and for all $x \in T^c$, $\text{bitsize}(T) + |x| > s$, then $|T'| \leq |T|$.

Proof (sketch). Suppose that $T$ is unskewed, $\text{bitsize}(T') \leq s$, and $|T'| > |T|$. Then a sequence of string substitutions may be made on the dictionary $T'$, obtaining a dictionary $T''$ such that $T'' = |T'|$, $\text{bitsize}(T'') \leq \text{bitsize}(T')$, and $T'' \not\supset T$. If we fix $x \in T'' - T$, then we have $\text{bitsize}(T) + |x| \leq \text{bitsize}(T'') \leq \text{bitsize}(T') \leq s$. \qed
Lemma 4.29. If \( j \geq 4 \) and \( j = (n - 1)2^{n+1} + 2 \) for some \( n \in \mathbb{N} \), then

\[
\log j - \log \log j - 1 - \frac{4}{j} \leq n \leq \log j - \log(\log j - \log \log j - 2) - 1.
\]

Proof. It is clear that

\[
\log(j - 2) = \log(n - 1) + n + 1.
\]

Thus,

\[
n = \log(j - 2) - \log(n - 1) - 1 \quad \text{(4.1)}
\]

\[
= \log(j - 2) - \log[\log(j - 2) - \log(n - 1) - 2] - 1,
\]

\[
\leq \log j - \log[\log j - \log(n - 1) - 2] - 1. \quad \text{(4.2)}
\]

It follows from (4.1) that

\[
n - 1 \leq \log j. \quad \text{(4.3)}
\]
By inequality (4.3) and equation (4.1), we have

\[ n \geq \log(j - 2) - \log \log j - 1. \]

Since \( \log(j - 2) = \log j + \log \left(1 - \frac{2}{j}\right) \), and for \( j \geq 4 \), \( \log \left(1 - \frac{2}{j}\right) \geq \frac{2}{j} \), we have

\[ n \geq \log j - \log \log j - 1 - \frac{4}{j}. \]

Finally, by inequalities (4.3) and (4.2), we have

\[ n \leq \log j - \log [\log j - \log \log j - 2] - 1. \]

Lemma 4.30. For all \( c_1, c_2 > 0 \) there exists \( \rho < 1 \) such that for all \( x \in \{0, 1\}^* \), if \( n \in \mathbb{N} \) is the largest element of \( \mathbb{N} \) such that \( (n - 1)2^{n+1} + 2 \leq |x| \), and at least \( c_1|x| \) bits of \( x \) are in LZ phrases of \( x \) longer than \((1 + c_2)n\), then

\[ \frac{C_{\text{LZ}}(x)}{|x|} \leq \rho + o(1). \]
Proof. We will find an upper bound on the size of the dictionary $T$ given by the Lempel-Ziv parsing of $x$. This will then yield an upper bound for the LZ compression ratio of $x$, since

$$\frac{C_{LZ}(x)}{|x|} \leq \frac{|T| \log |T|}{|x|}.$$ 

We define a long phrase to be an element of $T$ whose length is greater than $(1 + c_2)n$. We call all other elements of $T$ short phrases.

Let $s \leq (1 - c_1)|x|$ denote the number of bits in short phrases. We derive an upper bound for the size of the subtree $T_s$ consisting of the short phrases in $T$. By Lemma 4.28, $|T_s| \leq |T_s'|$, where $T_s'$ is an unskewed dictionary with

$$\text{bitsize}(T_s') \leq \text{bitsize}(T_s) = s.$$ 

Let $T_s''$ denote the largest complete binary tree contained in $T_s'$, and let $b = \text{bitsize}(T_s') - \text{bitsize}(T_s'')$. Notice that

$$s - b = \text{bitsize}(T_s'') = \sum_{i=0}^{d} i2^i = (d - 1)2^{d+1} + 2.$$
where \( d = \text{depth}(T_s'') \). It follows by Lemma 4.29 that \( d \) satisfies the inequalities

\[
d \geq \log(s - b) - \log \log(s - b) - 1 - \frac{4}{s - b}
\]

and

\[
d \leq \log(s - b) - \log \log(s - b) - 2 - 1.
\]

Thus, it follows that

\[
|T_s'''| < 2^{d+1} \\
\leq 2^{\log(s-b) - \log(\log(s-b) - \log \log(s-b) - 2)} \\
= \frac{s - b}{\log(s - b) - \log \log(s - b) - 2}.
\]

The phrases in \( T_s' \setminus T_s'' \) all have length exactly \( d + 1 \), and account for at most \( b \) bits, i.e.,

\[
\sum_{x \in T_s' \setminus T_s''} |x| \leq b.
\]

Therefore, there are at most

\[
\frac{b}{d + 1} \leq \frac{b}{\log(s - b) - \log \log(s - b) - \frac{1}{s - b}} \\
\leq \frac{b}{\log(s - b) - \log \log(s - b) - 2}.
\]
phrases in $T_s' \setminus T_s''$. Thus, the total number of phrases in $T_s'$ is bounded above by

$$
\frac{s - b}{\log(s - b) - \log \log(s - b) - 2} + \frac{b}{\log(s - b) - \log \log(s - b) - 2}.
$$

Thus we have

$$
|T_s'| \leq \frac{s}{\log(s - b) - \log \log(s - b) - 2}.
$$

It is clear that $b \leq (d + 1)2^{d+1}$ and $s - b = (d - 1)2^{d+1} + 2 \geq (d - 1)2^{d+1}$. It follows that

$$
b \leq \frac{d + 1}{d - 1} (s - b),
$$

Since $\frac{d+1}{d-1} \leq 3$ for all $d \geq 2$, it is clear that

$$
b \leq 3(s - b),
$$

whence

$$
b \leq \frac{3s}{4}.
$$

It follows that

$$
\frac{s}{4} \leq s - b,
$$

and thus

$$
\frac{s}{\log(s - b) - \log \log(s - b) - 2} \leq \frac{s}{\log s - \log \log s - 4}.
$$
Thus, the number of short phrases is at most

$$\frac{s}{\log s - \log \log s - 4}.$$  

On the other hand, the number of long phrases is at most

$$\frac{|x| - s}{(1 + c_2)n}.$$  

Thus, an upper bound on the total number of phrases is given by

$$|T| \leq \frac{s}{\log s - \log \log s - 4} + \frac{|x| - s}{(1 + c_2)n},$$  

and the compression ratio is upper bounded by

$$\frac{|T| \log |T|}{|x|} \leq \left( \frac{\frac{s}{|x|}}{\log s - \log \log s - 4} + \frac{1 - \frac{s}{|x|}}{(1 + c_2)n} \right) \log |T|.$$  

Using Lemma 4.28, the total number of phrases is bounded above by $2^{n+1}$, so $\log |T| \leq n + 1$, and it follows that the compression ratio is at most

$$\frac{|T| \log |T|}{|x|} \leq \left( \frac{\frac{s}{|x|}}{\log s - \log \log s - 4} + \frac{1 - \frac{s}{|x|}}{1 + c_2} \right) \left( 1 + \frac{1}{n} \right). \quad (4.4)$$
We want an upper bound for the right hand side of inequality (4.4) for \( s \) in the range \( \frac{n^2}{2} \leq s \leq (1 - c_1)|x| \). (Since there must be at least one phrase of each length less than \((1 + c_2)n, s \geq \frac{n^2}{2}\).) We consider two cases, \( s \leq 2^n \) and \( s > 2^n \).

First, if \( s \leq 2^n \), then \( \frac{n^2}{2} \leq s \leq \frac{|x|}{n} \), so

\[
\frac{s}{\log s - \log \log s - 1} \leq \frac{1}{\log s - \log \log s - 4} \leq \frac{1}{2 \log n - \log n - 5} \leq o(1).
\]

Now

\[
\frac{1 - \frac{s}{|x|}}{1 + c_2} \leq \frac{1}{1 + c_2},
\]

so the right-hand side of inequality (4.4) is at most

\[
\frac{1}{1 + c_2} + o(1)
\]

as \( n \) gets large.

On the other hand, if \( s > 2^n \approx |x|/n \), then \( 2^n < s < n2^{n+3} \), so

\[
\log s - \log \log s - 1 < n < \log s < n + \log n + 3
\]
Note that the factor \((1 + \frac{1}{n})\) is constant with respect to \(s\), so the first derivative of (4.4) with respect to \(s\) is always greater than zero:

\[
\left[ \frac{\frac{s}{|x|}}{\log s - \log \log s - 4} + \frac{1 - \frac{s}{|x|}}{1 + c_2} \right]'
\]

\[
= \frac{1}{|x|} \left[ \frac{(\log s - \log \log s - 4)/n - |s| \frac{1}{n} \left( \frac{1}{s} - \frac{1}{s \log s} \right)}{(\log s - \log \log s - 4)^2} - \frac{1}{1 + c_2} \right]
\]

\[
= \frac{1}{|x|} \left[ \frac{1}{\log s - \log \log s - 4} - \frac{\frac{1}{n} \left( 1 - \frac{1}{\log s} \right)}{(\log s - \log \log s - 4)^2} - \frac{1}{1 + c_2} \right]
\]

\[
> \frac{1}{|x|} \left[ 1 - \frac{1}{1 + c_2} - o(1) \right]
\]

> 0,

for sufficiently large \(n\).

It follows that equation (4.4) is maximized at the largest allowable value of \(s\), i.e., \(s = (1 - c_1)|x|\). For this value of \(s\), the compression ratio is at most

\[
\frac{1 - c_1}{1 - o(1)} + \frac{c_1}{1 + c_2} = \rho + o(1),
\]

where

\[
\rho = 1 - c_1 \left( 1 + \frac{1}{1 + c_2} \right) < 1.
\]

Comparing the maxima achieved in the two cases, we see that the overall maximum occurs in the second case. \(\Box\)
We need the following theorem, essentially due to Chernoff [13], which gives bounds on large deviations when applying the probabilistic method. We use the following particular form of the Chernoff bound from [30].

**Theorem 4.31.** Let \( p \in [0,1] \), and let \( X_1, \ldots, X_N \) be independent 0/1-valued random variables such that \( P[X_i = 1] = p \) for all \( 1 \leq i \leq N \). Then, for all \( 0 \leq \varepsilon \leq 1 \),
\[
P \left[ \sum_{i=1}^{N} X_i \geq (1 + \varepsilon)pN \right] \leq e^{-\frac{\varepsilon^2 pN}{3}}.
\]

**Lemma 4.32.** For all strings \( s \in \{0,1\}^* \) and all \( \delta > 0 \) there exists \( \rho < 1 \) such that for all sufficiently long finite strings \( x \in \{0,1\}^* \), if \( s \) appears with frequency more than \( 2^{-|s|} + \delta < 1 \) in \( x \) then \( C_{LZ}(x) \leq \rho|x| \).

**Proof.** Fix \( s \) and \( \delta \) as in the hypothesis. Define \( n \) to be the largest element of \( \mathbb{N} \) such that \((n - 1)2^{n+1} + 2 \leq |x|\). The Lempel-Ziv algorithm partitions \( x \) into consecutive phrases, and there are \( |T| \leq 2^{n+1} \) phrases that require \( \log|T| \leq n + 1 \) bits of the compressed strings to specify.

For each \( i \leq |x| - |s| + 1 \), we call the bits \( x[i..i+|s| - 1] \) a run of \( x \) of length \( |s| \).

We partition the runs of length \( |s| \) within \( x \) as follows. A small number \((o(|x|))\) of runs span two Lempel-Ziv phrases; we ignore these (and the potential appearances of \( s \) there). The other runs each have a relative position within a single Lempel-Ziv phrase of \( x \), and we partition these runs into \( |s| \) equivalence classes according to
the relative position within a phrase, modulo $|s|$. Since $s$ appears with frequency $2^{-|s|} + \delta$ over all of $x$, it follows that there exists an equivalence class $E$ in which $s$ appears with frequency $2^{-|s|} + \frac{\delta}{2}$.

Let

$$\eta = \frac{\delta^2 2^{|s|}}{12|s| \ln 2}.$$ 

Our objective is to use Lemma 4.30 with $c_2 = \eta$ and $c_1$ to be specified below. Applying the Chernoff bound with $\epsilon = 2^{|s|} \frac{\delta}{2}$, $N = \frac{(1+\eta')n}{|s|}$, and $p = 2^{-|s|}$, for each $\eta' \leq \eta$, there are at most

$$e^{-\frac{\delta^2 2^{|s|} (1+\eta')n}{12|s|}} 2^{(1+\eta')n}$$

strings of length $(1 + \eta')n$ in which $s$ appears more than

$$\left(2^{-|s|} + \frac{\delta}{2}\right) \frac{(1+\eta')n}{|s|}$$
times in the equivalence class $E$. This gives at most
\[
e^{-\frac{\varepsilon^2 \eta |s| (1 + \eta') n}{2|s|}} 2^{(1 + \eta') n} = e^\left(\ln 2 - \frac{\varepsilon^2 |s|}{2|s|}\right)(1 + \eta') n
\]
\[
= 2^n \left(1 - \frac{\varepsilon^2 |s|}{2|s|} n^2\right)^{(1 + \eta')}
\]
\[
= 2^{n(1 - \eta)(1 + \eta')}
\]
\[
\leq 2^{(1 - \eta^2)n}
\]
\[
= \alpha^n,
\]
such strings of length exactly $(1 + \eta') n$, where $\alpha = 2^{(1 - \eta^2)} < 2$. Fixing $\alpha < \beta < 2$, it follows that (for sufficiently large $n$) there are at most
\[
(1 + \eta)n \alpha^n < \beta^n
\]
strings of length less than or equal to $(1 + \eta)n$ in which $s$ appears more than
\[
(2^{-|s|} + \frac{\delta}{2}) \frac{(1 + \eta)n}{|s|} n
\]
times in the equivalence class $E$.

Consider the short phrases in the parsing of $x$ of length at most $(1 + \eta)n$, and let $b$ denote the number of bits in these phrases. Our goal now is to compute a lower bound on the quantity $|x| - b$, i.e., the number of bits in long phrases of length greater than $(1 + \eta)n$. We can then apply Lemma 4.30 and conclude that $x$ compresses for sufficiently long $x$. We compute this lower bound by first
computing an upper bound on the occurrences of \( s \) in \( E \) in the parsing of \( x \).

Let \( K = \frac{1}{|s|} (2^{-|s|} + \delta)|x| \) be the number of occurrences of \( s \) in \( x \). Let

a) \( K_1 \) be the number of occurrences of \( s \) in short phrases, i.e., phrases of length at most \( (1 + \eta)n \), in which \( s \) appears with frequency more than \( 2^{-|s|} + \frac{\delta}{2} \) in \( E \),

b) \( K_2 \) be the number of occurrences of \( s \) in \( E \) in short phrases in which \( s \) appears with frequency at most \( 2^{-|s|} + \frac{\delta}{2} \), and

c) \( K_3 \) be the number of occurrences of \( s \) in long phrases, i.e., phrases of length greater than \( (1 + \eta)n \).

It is clear that

\[
K \leq K_1 + K_2 + K_3 + \frac{o(|x|)}{|s|},
\]

where \( \frac{o(|x|)}{|s|} \) accounts for the number of occurrences of \( s \) that overlap phrases. We now compute upper bounds for \( K_1, K_2 \) and \( K_3 \).

Consider the short phrases in the parsing of \( x \). There are at most \( (1 + \eta)n^\beta n \) bits in short phrases in which \( s \) appears with frequency greater than \( 2^{-|s|} + \frac{\delta}{2} \) in \( E \), constituting at most \( (1 + \eta)n^\beta n \) bits in short phrases with this property. The frequency of \( s \) in these phrases is at most 1, and thus

\[
K_1 \leq \frac{1}{|s|} ((1 + \eta)n^\beta n).
\]
In short phrases in which $s$ appears with frequency at most $s^{-|s|} + \frac{\delta}{2}$ in $E$, there are $b - (1 + \eta)n \beta^n \leq b$ bits. In these phrases, the frequency of $s$ is at most $2^{-|s|} + \frac{\delta}{2}$, and thus

$$K_2 \leq \frac{1}{|s|} \left( (2^{-|s|} + \frac{\delta}{2}) b \right).$$

Finally, consider the bits in phrases longer than $(1 + \eta)n$ of which there are $|x| - b$. The string $s$ appears in these phrases with frequency at most 1, and thus

$$K_3 \leq \frac{1}{|s|} (|x| - b).$$

We now have

$$K \leq K_1 + K_2 + K_3 + \frac{o(|x|)}{|s|},$$

or

$$\frac{1}{|s|} \left[ (1 + \eta)n \beta^n + (2^{-|s|} + \frac{\delta}{2}) b + (|x| - b) + o(|x|) \right] \geq \frac{1}{|s|} (2^{-|s|} + \delta)|x|.$$

Solving for $b$, we have

$$b \leq |x| \left( 1 - \frac{\delta - o(1)}{2(1 - 2^{-|s|} - \frac{\delta}{2})} \right).$$
Thus,
\[ |x| - b \geq |x| \cdot \frac{\delta - o(1)}{2(1 - 2^{-|x|} - \frac{\delta}{2})} \geq c_1|x| \]
for sufficiently large \( x \), where
\[ c_1 = \frac{\delta}{2(1 - 2^{-|x|} - \frac{\delta}{2})} \]

Finally, since \( |x| - b \) is the number of bits in phrases longer than \((1 + \eta)n\), we apply Lemma 4.30 with \( c_2 = \eta \) and conclude that there exists a \( \rho < 1 \) such that
\[ \frac{C_{LZ}(x)}{n} < \rho + o(1). \]

**Theorem 4.33.** Every weakly LZ random sequence is normal.

**Proof.** We show that any sequence \( S \not\in \text{NORMAL} \) is compressed by the Lempel-Ziv compression algorithm. Since \( S \) is not normal, then there are infinitely many prefixes of \( S \) in which a string \( s \in \{0, 1\}^* \) appears with frequency more than \( 2^{-|s|} + \epsilon \). By choosing long prefixes, and applying Lemma 4.32 we have that infinitely may prefixes of \( S \) compress, whence \( S \) is not LZ random.

Theorem 4.33 gives an easy proof that the Champernowne sequence is normal [12, 40]. Since for any prefix of the Champernowne sequences, the dictionary is unskewed, it is easily seen to be LZ random. Thus, it is a normal sequence.

We now show that not every normal sequence is even weakly LZ random. We prove this by explicitly constructing a sequence \( S \) that is normal, but compresses
nearly optimally, i.e., $\rho(S) = 0$.

**Definition 4.34.** For $y \in \{0,1\}^*$ and $n \leq |y|$ the $n$-**necklace** of $y$ is the set

$$\text{necklace}_n(y) = \left\{ x \in \{0,1\}^n \mid (\exists z \in \text{necklace}(y)) x \subseteq z \right\},$$

i.e., the set of all $n$-bit prefixes of elements of the necklace of $y$.

**Definition 4.35.** A de Bruijn string of order $n$ is a string $y$ of length $2^n$ such that

$$\text{necklace}_n(y) = \{0,1\}^n$$

For example, 0110 is a de Bruijn string of order 2, and 00010111 is a de Bruijn string of order 3. It is well-known that de Bruijn strings of an order are easily constructed [73]. We fix a de Bruijn string of each order $n$ and call it $dB_n$. This sequence of strings is useful because $dB_3$ contains $2^{3-1}$ 0s and 1s; $2^{3-2}$ 00s, 01s, 10s, and 11s; and exactly one each of the strings of length 3. In general, a string $x \in \{0,1\}^{\leq n}$ appears $2^{n-|x|}$ times in the first $2^n$ bits of $dB_n \cdot dB_n$.

**Lemma 4.36 ([70]).** If $a_1, a_2, a_3, \ldots \in \mathbb{N}$ is sequence of integers with limit infinity, then the sequence

$$S = dB_1^{a_1} \cdot dB_2^{a_2} \cdots dB_k^{a_k} \cdots$$
is a normal sequence.

**Theorem 4.37.** For all $\rho > 0$, there exists a normal sequence $S$ such that

$$\frac{LZ(S[n,n-1])}{n} < \rho \text{ a.e.}$$

**Proof.** Let $S = dB_1 a_1 \cdot dB_2 a_2 \cdot dB_3 a_3 \cdots$, where $a_1, a_2, \cdots$ is a sequence of integers.

We now choose this sequence so that the compression ratio of $S$ goes to 0. Define $x_k$ to be the concatenation of the strings $dB_1 a_1 \cdot dB_2 a_2 \cdots \cdot dB_k a_k$. We now define the sequence $a_1, a_2, \ldots$ by

$$a_i' = \min \left\{ n \in \mathbb{N} \mid \forall k \geq |x_{i-1}| + n|dB_i| \frac{C_{i,Z} \{x_{i-1} \cdot dB_i \cdot dB_{i+1}^{\infty}[0..k-1]\}}{k} \leq \rho \right\}$$

(4.5)

$$a_i = 1 + \max_{j \leq a_i} a_j'.$$  

(4.6)

At the interface of each $x_m$ and $dB_m a_m$, the continued parsing of this string either follows an existing strand of the dictionary, or adds a new entry to the dictionary. In the latter case, the dictionary may be increased in size and therefore in width by at most one, and thus after $x_m$ has been parsed, the dictionary is of width at most

$$m + \sum_{j=1}^{m} j = \frac{m(m+3)}{2}.$$

At this point, the parsing of $dB_m a_m$ begins at the root of the dictionary and can be treated as periodic. Thus, by Lemma 4.20 the set in (4.5) is non-empty.
Lemma 4.36 tells us that the sequence \( S \) is normal. 

We now have the following relationships among LZRAND, wkLZRAND and NORMAL:

\[
\text{LZRAND} \subsetneq \text{wkLZRAND} \subsetneq \text{NORMAL}.
\]

We now turn our attention to finite-state compressors and their relationship to LZ random sequences.

**Definition 4.38.** A **compressor** is a finite-state transducer

\[
C = (Q, \{0, 1\}, \{0, 1\}, \delta, \lambda, q_0).
\]

The semantics of compressors is the same as for finite-state transducers. An input from \( \{0, 1\}^* \) is read by the compressor one bit at a time. At each step, the compressor reads the next bit from the input, and according to the transition function, changes state and outputs a string from \( \{0, 1\}^* \). The output of a compressor \( F \) given input \( x \in \{0, 1\}^* \), denoted \( F(x) \), is the pair consisting of the output string produced by the transition function together with the state the compressor is in after all the input is read. In this thesis, we consider inputs from \( \{0, 1\}^* \) only. Thus, for the compressor to compress an input string, the transition function of the compressor must necessarily contain transitions of the form \( (q, x) = (q', \lambda) \), i.e., transitions where no output is produced.
Compressors are best visualized by state diagrams like the one shown in Figure 4.3. In this example, $F(000000000000) = (1111, q_a)$; however, $F(111) = (000100010001, q_a)$. This example explicitly shows an example of a compressor that compresses long sequences of zeros, but perform poorly on sequences with many ones.

**Definition 4.39** (Huffman [35, 36]). A compressor $F$ is *information lossless* (IL) if $F_f$ is one-to-one.

Note that the compressor in Figure 17 is information-lossless.

![Figure 17: A finite-state compressor](image-url)
It is well-known and easy to prove that a sequence $S \in C$ is normal if and only if it is not compressible by any information lossless finite-state compressor. Briefly, if a sequence $S \in C$ is compressible by a finite-state compressor, then the sequence $S$ must drive the compressor through a loop that outputs a string shorter than the input in that loop. Since the sequence does compress, the substring that drives the compressor through the loop must occur too frequently, and thus $S$ is not normal. On the other hand, if $S$ is not normal, then it is easy to construct a Huffman code [34] that compresses $S$. This code is then easily used to construct a compressor that compresses $S$. In light of this fact, we now have the following new theorem.

**Theorem 4.40.** If $S$ is compressible by an information-lossless, finite-state compressor, then $S$ is compressible by the LZ compression algorithm.

**Proof.** Since $S$ is information-lossless finite-state compressible, it is not normal. By Theorem 4.33, $S$ is compressible by the LZ compression algorithm. \(\square\)

### 4.4 LZ Strong Depth

We now turn our attention toward defining a notion of strong LZ compression depth. To do this, we modify the definition of strong computational depth as follows.
Definition 4.41. A sequence $S$ is *strongly LZ compression deep* and we write $S \in \text{strLZDEEP}$ if for all $c \in \mathbb{N}$ and all $s \in \mathbb{N}$,

$$\text{Cdepth}_s^LZ(S[0..n - 1] > cn) \text{ a.e.}$$

Intuitively, a sequence is strongly compression deep if its prefixes contain redundancy that cannot be compressed to within $s$ bits of its minimum compression with $cn$ resources, for all $c$ and $s$.

We now show that simple sequences cannot be strongly LZ compression deep.

Theorem 4.42.

$$\text{strLZDEEP} \cap \text{LZSIMPLE} = \emptyset.$$

Proof. Assume $S \in \text{LZSIMPLE}$. It suffices to show that $S$ is not strongly LZ deep. Since $S \in \text{LZSIMPLE}$, $\text{width}_LZ(S, n) \leq cn$ for some $c \in \mathbb{N}$. Thus for all $n \in \mathbb{N}$, $S[0..n - 1]$ can be compressed to its optimum with at most $cn$ dictionary entries. That is, $C_{LZ}(S[0..n - 1]) = C_{LZ}^c(S[0..n - 1])$. Then, for all $s \in \mathbb{N}$, $\text{Cdepth}_s^LZ \leq cn$. Thus, $S \notin \text{strLZDEEP}$. \qed

Corollary 4.43. $\text{PERIODIC} \cap \text{strLZDEEP} = \text{FSTSIMPLE} \cap \text{strLZDEEP} = \emptyset$.

Proof. By Theorem 4.21 $\text{PERIODIC} \subseteq \text{LZSIMPLE}$. \qed
We now show that LZ-random sequences are also shallow. In fact, for all but finitely many $n$, the LZ compression depth of the first $n$ bits of an LZ random sequence is 0.

**Theorem 4.44.** $\text{strLZDEEP} \cap \text{LZRAND} = \emptyset$.

**Proof.**

\[
S \in \text{LZRAND} \implies (\exists s)(\forall n)C_{LZ}(S[0..n - 1]) \geq n - s
\]
\[
\implies C_{LZ}^0 \geq n - s
\]
\[
\implies (\exists s)(\forall n)S \notin DLZ_3^0
\]
\[
\implies (\exists s)S \notin DLZ_3^0
\]
\[
\implies S \notin \text{strLZDEEP}.
\]

Finally, we show that the definition of strongly LZ compression depth is not vacuous.

**Theorem 4.45.** $\text{strLZDEEP} \neq \emptyset$

**Proof.** We define $S$ to be the unique sequence such that $x_m \subseteq S$ for all $m \in \mathbb{N}$. where the sequence $x_0, x_1, x_2, \ldots$ of strings is defined recursively as follows. For
the basis step, let $x_0 = \lambda$. For the induction step, assume that $x_m$ has been defined. Then

$$x_{m+1} = x_m \cdot f_{m+1} \cdot l_{m+1},$$

where $f_{m+1}$ and $l_{m+1}$ are defined as follows. $f_{m+1}$ is a string that, given $x_m$ and the dictionary associated with $x_m$, fills in the dictionary so that every string of length $m + 1$ is in the dictionary. $l_{m+1}$ is a string that, given $x_m \cdot f_{m+1}$ and its associated dictionary, adds entries to the dictionary in lexicographic order with the restrictions that

1. every string in the dictionary produced by $x_m \cdot f_{m+1}$ is the prefix of the same number of entries in the dictionary produced by $x_m \cdot f_{m+1} \cdot l_{m+1}$, and

2. $C_{LZ}(x_m \cdot f_{m+1} \cdot l_{m+1}) \leq \frac{1}{2}|x_m \cdot f_{m+1} \cdot l_{m+1}| - 2^{m+1}.$

Intuitively, we "bury a lot of redundancy" in between a few randomly placed strings.

To see that $S$ is strongly LZ deep, let $c \in \mathbb{N}$ and $s \in \mathbb{N}$ be arbitrary. We will show that $S \in DLZ^c_s$. First, it is clear that for all but finitely many $n \in \mathbb{N}$ that $C_{LZ}(S[0..n-1]) \leq \frac{1}{2}n$. Since the dictionary for $S$ will eventually contain every string of length $l$ for all $l \in \mathbb{N}$, it is also clear that for all $t \in \mathbb{N}$, and for all but finitely many $n \in \mathbb{N}$, $C_{LZ}^t(S[0..n-1]) \geq n$. Since entries in the dictionary are entered in lexicographic order when $S$ is parsed, we can choose a significantly long
prefix of $S$ so that

(1) $C_{LZ}(S[0..n-1]) \leq C^i_{LZ}(S[0..n-1]) - s$, and

(2) the minimum number of resources required to compress $S[0..n-1]$ is at least $cn$.

Thus, $S \in DLZ_c$ and since $s$ and $c$ were arbitrary, $S \in strLZDEEP$. \qed
CHAPTER 5. GENETIC ALGORITHMS AND COMPRESSION DEPTH

Darwin [17] studied the relationship between natural selection and complexity of organisms, eventually linking less complex older organisms to more complex recent organisms through the process of evolution. An analog to natural selection is also used in computer science to search for better (more fit) answers embedded in large search spaces. Using genetic algorithms and compression depth, we experimentally show that “natural selection” in computer science also produces complexity as the population evolves. Since genetic algorithms may be considered an algorithmic process by which random objects are turned into more complex objects (in the sense of compression depth) over a long period of time via evolution, it exemplifies the notion of the slow-growth law. Complexity is produced slowly and thus cannot be created without a commensurate history of computation.

Genetic algorithms are commonly used to solve optimization problems where other search algorithms fail. First introduced by Holland [31, 32], genetic algorithms are an important tool for finding solutions to a variety of computational problems [60, 27, 23, 18, 26, 25]. Basically, a genetic algorithm operates by initially producing a random sampling (population) of possible “solutions” (genes) and then, using genetic operators, forms new possible “solutions” in the popula-
tion. Once this is completed, a fitness function is used to remove weaker (less fit) genes from the population. This procedure is iterated, producing a new generation of genes in the population each iteration.

This chapter applies compression depth to measure the relative complexity of the behavior of finite-state transducers that play iterated prisoner's dilemma. Using a genetic algorithm to evolve a population of finite-state transducers, we show experimentally that the compression depth of their behavior grows, depending on the interactions of the population as they are evolved.

5.1 Prisoner's Dilemma

Prisoner's dilemma is a simple game played between two players. First introduced by Flood and Dresher [20], this game is now widely used in many fields, including image processing, function optimization, biology, economics, engineering, and computer science [24]. A typical example of the type of problem that can be modeled by the Prisoner's dilemma is illustrated as follows [3]. Suppose a drug dealer and drug buyer wish to make a deal where money is exchanged for drugs. The dealer and the buyer arrive at opposite sides of a designated busy street. The dealer then places a sealed envelope containing the drugs under a nearby mailbox. The buyer places a sealed envelope containing the money under a bush. Then, they each cross the street at the same time and retrieve the envelope that
the other person left. The situation can have four outcomes. The dealer and the buyer may both be honest and "cooperate" with each other in which case the drug dealer places the drug in the envelope and the buyer places money in the envelope. On the other hand, they both could be dishonest and "defect" by placing parsley and newspaper in the envelopes respectively. In the other two cases one person cooperates and the other person defects, in which case one person ends up with both the drugs and the money.

The situation described above is modeled by the game called Prisoner's dilemma as follows. In this game, each player either cooperates or defects simultaneously (in ignorance of the other player's play) and a payoff (usually an amount of money) is returned to each player based on their plays according to a payoff table. Formally, we define the payoff table as $H, L, C, D$, usually depicted as shown in Figure 18. If both players cooperate, then they each receive a payoff of $C$. If both players defect, then they each receive the payoff $D$. If one player defects and the other player cooperates, then the player that defects receives a payoff of $H$, and the player that cooperates receives a payoff of $L$. Thus, the payoff table defines the potential risk and gain for defecting, and must obey the relations $L < D < C < H$ and $\frac{(L+H)}{2} < C$.

As is standard [5], in this thesis, we adopt the values $H = 5$, $L = 0$, $D = 1$, and $C = 3$ for all experiments.
5.2 Iterated Prisoner’s Dilemma

Iterated prisoner’s dilemma (IPD), introduced by Axelrod [4, 5], extends the prisoner’s dilemma game to an arbitrary number of play repetitions. Under this new game, complex interactions form between the players as they “learn” about the other player’s strategies. There are numerous strategies that can be used, and their success is usually dependent on the players. For example, a simple strategy may be to always cooperate, no matter what the other player does in the hopes that he will do the same. This strategy yields an average per play winnings of $C = 3$ in the presence of another always-cooperative player. A more ruthless strategy is to always defect in the hope that you are playing with a generous player. More complex strategies can be used such as tit-for-tat (cooperate on the first round and then play what the other player played the last round), random, or various secret handshakes. Secret handshakes allow two players to identify each other using a code of defects and cooperates [61, 21]. Once the the two players
identify each other, they switch their plays to always cooperate.

5.3 Genetic Algorithms and Iterated Prisoner's Dilemma

As mentioned earlier, genetic algorithms are a method for optimizing, using natural selection and genetic operators to navigate through the search space. In this section we describe how to play iterated prisoner's dilemma using finite-state transducers, and how to use a genetic algorithm to evolve the finite-state transducers to win a larger payoff per round.

We begin by describing how a finite-state transducer (FST) can be used to control a player in IPD. Informally, the FST uses the other player's last play, either cooperate or defect, as its input. Based on its current state, the FST will transition to a new state and output either a C or D for cooperate or defect. This is deemed the FST's next play. This procedure is repeated for each new play. Initially, the FST assumes that the other player cooperated the last play. Note that since the states of the FST form a memory, it is easy to implement strategies such as tit-for-tat and secret handshaking.

Formally, we define an IPD player as follows.

**Definition 5.1.** An IPD player is a finite-state transducer $M = (Q, \{C, D\}, \delta, \lambda, q_0)$.

where $\delta : Q \times \{C, D\} \rightarrow Q$ and $\lambda : Q \times \{C, D\} \rightarrow \{C, D\}$.

We use a standard genetic algorithm [24] to evolve a population of 100 IPD
players. The fitness criterion is the average payoff per play achieved; the higher
the average payoff, the higher the fitness value. We now give a detailed description
of the parameters in the genetic algorithm used here.

Each IPD player in the population of 100 contains 2000 states. The transition
function is either mapped to random states or state number 0. The output func­
tion is similarly initialized to either random outputs, or all defects. We study the
compression depth of the IPD player under both of these initial conditions. In
either case, the start state is selected randomly from among all possible states.

Internal to the genetic algorithm, the IPD player is represented by a single
array, each array location representing a single state. Thus, the index to the array
location serves to identify the state. For example, information about state 0 is
found in array location 0. Each array location contains four elements. The first
two elements are integers and define the next state transition function. That is,
the first of these two integers gives the array index of the state to change to if the
last play of the other player is defect. Similarly, the second integer gives the array
index of the state to change to if the last play of the other player is cooperate.
The other two elements of the array location define the output function, either
cooperate or defect, i.e., the play made by the player this round.

We now define the crossover operator. Let $M_1$ and $M_2$ be two IPD players,
and let $A_1$ and $A_2$ be the arrays containing the information about the transition
functions of $M_1$ and $M_2$, respectively. Choose two random numbers between 0 and the minimum of number of states in either $M_1$ or $M_2$, less 1, inclusive. These two numbers are then ordered with the smaller denoted as $j$ and the larger denoted as $k$. Let $M_3$ with array $A_3$ and $M_4$ with array $A_4$ be the result of the crossover.

The crossover operator applied to $M_1$ and $M_2$ is defined as follows. For each $l \in \mathbb{N}$ such that $0 < l < j$, $A_3[l] = A_1[l]$ and $A_4[l] = A_2[l]$. For each $l \in \mathbb{N}$ such that $j < l < k$, $A_3[l] = A_2[l]$ and $A_4[l] = A_1[l]$. For each $l \in \mathbb{N}$ such that $k < l < \text{(number of states in $M_1$)}$, $A_3[l] = A_1[l]$. Finally, for each $l \in \mathbb{N}$ such that $k < l < \text{(number of states in $M_2$)}$, $A_4[l] = A_2[l]$. Note that the pointers in the transition function do not change even though the contents of the state they point to may change. Thus, this crossover operator can, with a significant probability, produce offspring with drastically different behavior than the parents. However, since the IPD player parent's cells are copied to the same relative location (array index) of the children's cells, it is possible to also preserve local behavior, usually due to small loops contained within a single crossover region.

We have four mutation operators that operate on an IPD player. The first mutation operator complements a randomly chosen bit in the output transition function. (i.e, it changes a defect to a cooperate, or a cooperate to a defect.) The second mutation operator changes a transition in the transition function from one state to another state selected uniformly at random. The third mutation
operator randomly deletes a state, and changes any other state that "points" to that state to a new state at random. Our last mutation operator adds a state to the IPD player, randomly setting all its information. In the experiments described here, the mutation rate is 5 percent with an equally likely chance of each type of mutation being selected.

As is standard for genetic algorithms, a percentage of the most fit members of the population are selected to reproduce using the crossover and mutation operators. The offspring then replace the least fit members of the population. The fitness function provides the measure of fitness by which this selection process takes place. In the experiments below, each IPD player (finite-state transducer) plays IPD against every other IPD player in the population for 4000 plays. The fitness is then the average payoff over all plays and all other players. The top 10 players with the highest fitness value are then selected to reproduce, and their offspring replace the 10 players with the lowest fitness. The process is then repeated to produce the next generation.

5.4 Compression Depth, Genetic Algorithms, and Iterated Prisoner's Dilemma

In this section we experimentally investigate the compression depth of the structure of finite-state machines (IPD players) that play iterated prisoner’s dilemma
as they are evolved by a genetic algorithm. We measure the compression depth of a representation of each IPD player in the population. As described below, this representation must be simple so that compression depth is not introduced as an artifact of the representation.

For the purpose of a control experiment, we select 10 players at random each generation to reproduce. The offspring replace 10 other players selected at random. This effectively removes all selective pressure from the genetic algorithm. We then measure the compression depth of each player every 100 generations and average the values over all players. This gives the expected depth generated by the genetic algorithm when no selection pressure is present. The depth of each player (FST) in the population is then measured by converting the player to a string of binary bits. This conversion must be "simple" in order for the conversion process itself to not introduce depth. In the experiments described here, we convert an IPD player to a string of bits as follows. For all states $q \in Q$, we convert $\delta(q,a)$ and $\lambda(q,a)$ to the string $\cdot A(q,a)$, where $a \in \{C,D\}$. Since in this case, $\delta(q,a)$ is a number that yields the next state, the binary string above is roughly $\log |Q|$ in size. We then concatenate these strings together beginning with state 0 to yield a simple binary description, whose total length is roughly $2|Q| \log |Q|$.

As expected, when the population of players is initialized randomly, and no selection pressure is present, the results after each generation are still nearly ran-
dom, and hence LZ random. We say nearly random because the "add a state" mutation operator is the source for a small amount of non randomness. This particular mutation operator adds a state to the IPD player, but is unreachable from any other state. Thus, the IPD player is no longer random. However, since other mutation operators may change a transition from one state to this new state, and combined with the fact that mutations are rare, 10000 generations produce an average of 125 of these types of mutations. Experimental simulation shows that the binary representation after 10000 generations with no selection pressure remains LZ random. This result was verified by performing the simulation a total of 3 times. Recall that the resource-bounded LZ compression algorithm with an empty dictionary outputs the input when the input is not LZ compressible. Since the LZ compression is the minimum taken over all dictionary sizes, the LZ compression for any dictionary size is the length of the input. Since the binary description of the IPD player remains LZ random through all generations, the depth at each generation for all IPD players is 0 at all significance levels $s$. These control simulations were carried out 3 times, and verified that the compression depth of 10 randomly selected IPD players was 0 at 100 generation intervals.

We also verify that if members of the population are initialized so that their binary representation is all zeros, then the lack of any selection pressure also does not produce structure in the form of higher compression depth in the binary
representations of IPD players. Simulating this situation for 4000 generations, we verify that no significant increase in compression depth occurs by examining the average compression depth of 10 IPD players every 100 generations. This simulation was performed a total of 3 times with no significant difference in the results. For comparison purposes, the graph of one of these control experiments labeled “control” is shown in Figures 20 and 21.

We now contrast these control experiments with experiments where selection pressure is included using the fitness function described above. In our first experiment, we initialize the players with random values. As shown in Figure 19, we see that the average compression depth of the ten most fit players generally increases as more generations (computation time) is provided. Due to the lack of sufficient computing power, it is unknown how much depth can be accumulated in this manner, and at what rate it is accumulated. However, this experiment was repeated 3 times with nearly the same result and indicates that for non-trivial selection pressures, complexity (in the form of compression depth) increases as the population evolves. The control simulation is not shown in this figure since, as mentioned above, the compression depth for no selection with randomly initialized players remains 0 for all players for all generations.

Also shown in Figure 19 is the relationship to fitness of the top 10 IPD players after each generation. (For these simulations, data was taken after every gen-
eration.) It is interesting that the rapid shift of the population to a nearly all cooperate strategy precedes the rise in compression depth. We speculate that once the population made this shift, the top IPD players began to find ways to exploit other IPD players in the population. In turn, these players learn how to exploit other players as well as to defend against exploitation. Thus, we believe that the IPD players develop complex strategies to recognize the play of other players to either exploit them or defend against them. These strategies are defined by the finite-state machines in the IPD players, and by necessity evolve to more complex strategies. Thus, we conjecture that the complex nature of these strategies translates to intricate structure in the transition function as measured by increasing compression depth.

Due to the nature of the LZ compression algorithm used in this thesis, the time required to evaluate the depths of strings that are random or nearly random is much longer than for simple strings. This effect is mainly due to the width and size of the dictionary generated by nearly random strings. For this reason, we now present results of experiments where the players are initialized to values that have binary representations of all zeros.

We performed 10 experiments of this kind. It is most significant that in each simulation, the compression depth rose substantially above the control simulation, without exception. The results of these 10 simulations divide into two categories:
those where the top 10 members of the population rise to a fitness above 2.5, and those that do not. We recorded 6 simulations in which the fitness of the top 10 IPD players rose above a fitness value of 2.5. The other four simulations, the top 10 IPD players never achieved a fitness value above 2.0. Figures 20 and 21 show typical examples of these two situations. It is not surprising that in both cases, the compression depth rises almost immediately. We conjecture that this initial rise is due mostly to mutation which adds information in the form of random bits into the "information-free" initial population. However, since a selection pressure is present, the IPD players in which the random information produces (marginally) higher fitness values are saved and propagated in the population. Although they do not perform much better than the all-defect strategy, we conjecture that the selection pressure adds enough bias to begin organizing strategies and structure in the IPD player. Thus, the compression depth initially rises.

In the 6 simulations in which the fitness rose over 2.5, the fitness value of the top 10 IPD players rose sharply over relatively a small number of generations. This rise in fitness signaled a steep rise in the compression depth in each of the 6 simulations. We conjecture that a phenomenon similar to that in the population initialized randomly occurs at this point. We conjecture that the IPD players are again finding strategies for exploiting and defending against exploitation. Although this behavior is also possible with low fitness values, we did not observe
this in any of our simulations. It is also known [3] that this behavior is rare and not surprising that we did not witness it. This matter clearly deserves more investigation.

We also can interpret these simulations another way. We conjecture that the compression depth is measuring organization in potential solutions. When enough organization has accumulated in a sufficient number of players, the probability of producing a more fit player becomes higher. This raises the following provocative questions. Suppose that a population is first evolved to maximize compression depth, and then evolved to maximize the fitness function. Would the number of generations required to find a fit solution decrease? Is there a “universal” initial population made up of compression deep players, that will almost always speed up the time to solution? Can an initially deep population help guarantee convergence? Can a compression deep string be used in place of the random number generator to yield a compression deep population?

Figure 21 shows the compression depth and fitness of a typical simulation in the group of 4 populations whose top 10 IPD players never rose above a fitness value of 2.0. The compression depth of all four of these simulations initially rose to a value of over 150, but failed to reach a value over 200. We conjecture that the IPD players in these populations did not learn in the amount of time given to exploit the other players. Thus, the compression depth did not attain the values
shown in Figure 20.

The simulations and experiments described in this chapter yield new insight into the nature of genetic algorithms. We have clearly demonstrated an important link between evolution in genetic algorithms and compression depth. The results of the simulations here show that compression depth may be a useful tool for measuring the complexities of strategies (programs or data structures) in any system that stores and organizes information. We conjecture that compression depth can further be used to not only measure, but to guide search algorithms towards complex solutions and away from local minimums and maximums. It is clear that compression depth, and future complexity measures that are computable approximations computational depth, could play a significant role in experimental and theoretical computer science.
Figure 19: Typical simulation result when population is initialized randomly.
Figure 20: Simulation result of population initialized to all zero.
Figure 21: Simulation result of population initialized to all zero.
CHAPTER 6. CONCLUSION

In this thesis, we have defined the notion of recursive depth. Using this notion, the results in this thesis, together with earlier results of Bennett [8], Juedes, Lathrop, and Lutz [38], and Fenner, Lutz, and Mayordomo [19], establish the following relationships.

\[
\text{rec-wkDEEP} \subseteq \text{wkUSEFUL} \subseteq \text{rec-strDEEP} \subseteq \text{wkDEEP} \subseteq \text{strDEEP}
\]

We conjecture that the inclusion \( \text{wkUSEFUL} \subseteq \text{rec-strDEEP} \) is also proper, i.e., that rec-strong depth is not a sufficient condition for weak usefulness. It is also clear that the latency parameter \( l \) in the definition of recursive depth can be further restricted to yield a notion of computational depth inside exponential or polynomial complexity classes. Investigations into these questions are clearly important to complexity theory, and may be important to the field of computer science at large. Notions of polynomial depth may shed new light on the relationship between information and computation for the class \( \text{NP} \) and other subrecursive complexity classes. Further investigations in computational depth will clearly lead
to a better understanding of the role that information plays in the complexity of computation.

Also in this thesis, we have given an intuitive and theoretically grounded notion of compression depth. We have defined strong LZ compression depth and proven that strongly LZ deep sequences are not LZ random or LZ simple. We have also shown that the class LZRAND is properly contained in the class of all normal sequences, yielding new insight into LZ compression.

It is easily shown that if a transducer receives a sequence \( S \in \text{LZSIMPLE} \), then the output from the transducer is also LZ-simple. This and other considerations lead us to believe that FST reductions can be used to define a slow-growth law for LZ compression depth. We thus leave the following conjecture as an open problem.

**Conjecture 6.1.** Let \( S, T \in \mathcal{C} \) if \( S \in \text{strLZDEEP} \) and \( S \leq_{\text{FST}} T \) then \( T \in \text{strLZDEEP} \).

Finally we have shown experimentally that LZ compression depth may be used to measure the organizational complexity that arises in the population of a genetic algorithm. In particular, finite-state transducers that play iterated prisoner's dilemma were evolved using a genetic algorithm. We have found that the compression depth gradually rises even if the fitness does not, and suggest that the compression depth is measuring organization in potential solutions.
We further conjecture that when enough organization has accumulated in a sufficient number of players, the probability of producing a more fit player becomes higher. This raises the following provocative questions. Suppose that a population is evolved to first maximize compression depth, and then evolved to maximize the fitness function. Would the number of generations required to find a fit solution decrease? Is there a “universal” initial population made up of compression deep players, that will almost always speed up the time to solution? Can an initially deep population help guarantee convergence? Can a compression deep string be used in place of the random number generator to yield a compression deep population?

Further study of compression depth and its relationship to genetic algorithms may potentially lead to new theories in artificial life, genetic algorithms, and optimization.
BIBLIOGRAPHY


[70] M. J. Strauss. personal communication.


ACKNOWLEDGEMENTS

I thank my advisor Jack Lutz for his guidance and support without which this dissertation would not have been possible. I also thank the members of my committee Dan Ashlock, Jonathon Smith, Giora Slutzki, Vasant Honavar, and Martin Strauss for many useful comments. I especially thank the support of my parents. I also thank Jennifer Donnelly, John Walker, Jennifer Freeman and Jerry Burch for useful discussions and proofreading of this dissertation. Finally, I thank Stomping Grounds for their coffee.