CHOO XIANJUN, DAVIN

AUTOMATIC COMPLEXITY

A study of the complexity
of binary strings in
automata theory

Supervised by

Frank Stephan

in partial fulfilment of the requirements for the degree of

B.Sc (Hons) in Mathematics

Automatic Complexity

Acknowledgements

First and foremost, I would like to thank my advisor Dr. Frank Stephan for his guidance and wisdom. I am grateful for him dedicating an hour or so per week to my project, despite his hectic schedule. His valuable advice has also gotten me out of several road blocks throughout the course of this project. I always learn a lot from the weekly discussions that we have.

I started this project with a vague idea that I wanted to do something theoretical, perhaps something related to algorithmic randomness, recursion theory or learning theory. I wanted to pick up something new and exciting. However, after some discussion with Frank, we agree that 1 year isn't a long time and I would learn the most if I worked on a project where I already had some background or could pick up quickly. This led to the study of Kolmogorov complexity and automata, both of which I have learnt in my courses in Computer Science. Initially, I was disappointed that I wouldn't be picking up a new topic of choice but as the year progresses, I have gained tremendous appreciation for the study of bounds. It is usually easy to find an upper bound for the complexity, hard to find the lower bound, and harder to get both the upper and lower bounds to agree, even up to some constant factor. I am thankful that Frank has been very patient with me on this experiential journey over the past year.

I cannot thank my family enough for their unconditional support and for being understanding for the weekends that I spend away from home to work in school. My gratitude also extends to my close friends who have stuck with me, listened to my ranting, offered advice, and simply being a joy to be around with - Desmond: You have been a spiritual guidance and helpful mentor/senior in my journey in Mathematics. I hope you can become world champ and/or represent Singapore in Olympics arm wrestling some day; (99bao) Chris and Shawn: While the future is uncertain for all of us now, I wish you guys all the best in your academic endeavours; (Turing in the deep/Project find KK a GF) Keng Kiat, Kah Hou, Yik Jiun, Chun Mun and Yanxian: Our group conversation has never failed to brighten up my day and thank you for being one of the first lines of response in any kind of situation.

Abstract

This thesis studies binary string complexity on the DFA, NFA and CFG computation models. Matching asymptotic lower and upper bounds have been proven for each of these models. Under our complexity definition, we study some interesting binary strings such as the Morse-Thue sequence, characteristic sequence of the set of all palindromes, and Martin-Löf random strings. We conclude with a program that computes the complexity for short prefixes of these binary strings. The results empirically support the bounds that we have proven earlier.

Contents

Chapter 1

Introduction

1.1 Motivation

Kolmogorov complexity is a field of study which studies the descriptive complexity of objects of interest, usually sets or strings. Defined loosely, the complexity of an object is the *minimum description length* required to describe the object in a particular model or setting. The complexity of a string x is often denoted by $K(x) = \min\{|w| : w \text{ describes } x \text{ in the model}\}$. Common computational models include Turing machines, unlimited register machines, recursive functions, etc. The complexity of an object is not dependent on its length, but on the intrinsic representation of the object. If, within a certain model, an object has some sort of patterns in its representation, it will tend to have low complexity. We defer to [VL97] for an introduction to Kolmogorov complexity and its applications.

For starters, let us consider binary strings in the descriptive model of the English language. Despite being infinite in length, the string $x = 101010...$ has a finite description length. One possible description would be "10 repeated infinitely" (22 characters), hence $K(x) \leq 22$. A keen reader would notice 2 key issues with this example. Firstly, the English language is ambiguous. Secondly, it is unclear how one finds the minimum description length for any given string.

The ambiguity of English gives rise to paradoxes such as the *Berry Paradox*. One of its versions states: If w is "The first positive integer that cannot be specified in less than a billion words", then what is the integer that w describes? [Cha94] provides an interesting exposition on the topic. As a result of such ambiguity, we resort to well-defined models such as Universal Turing Machines.

The answer to the second issue is a negative one. The Kolmogorov complexity of any given string is not computable. Refer to Theorem A.0.1 for a simple proof.

It is usually easy to find an upper bound for the complexity[1], hard to find the lower bound, and harder to get both the upper and lower bounds to agree[2]. In practice, a simple way to obtain the upper bound of any object is just to run any existing compression algorithm on it and output its compressed length.

In this project, we consider models that are weaker than Turing machines and study the complexity of binary strings in those models. For convenience, unless stated otherwise, all occurrences of "strings" refer to "binary strings".

1.2 Prior work

In [SW01], Shallit and Wang considered the complexity of strings in the model of deterministic finite automata (DFA). In [Hyd13; HKH15], Hyde and Kjos-Hanssen extended Shallit and Wang's work by using the same complexity definition but with the model of non-deterministic finite automata (NFA). To avoid confusion, the complexity measures are denoted by $A_D(x)$ and $A_N(x)$ to represent complexity with respect to DFA and NFA respectively.

Definition 1.2.1.
(Shallit and Wang [SW01], Hyde and Kjos-Hanssen [Hyd13; HKH15])
Let $\Sigma = \{0, 1\}$ and $x \in \Sigma^*$ with $|x| = n$.

$$A_D(x) = \min\{|M| : M \text{ is a DFA such that } L(M) \cap \Sigma^n = \{x\}\}$$

$$A_N(x) = \min\{|M| : M \text{ is a NFA such that } L(M) \cap \Sigma^n = \{x\}\}$$

That is to say, any DFA (resp. NFA) M with x as the only length n string in its language has at least $A_D(x)$ (resp. $A_N(x)$) number of states.

Alternatively, the definition can be made as follows:

$$A(x) = \min\{|M| : \forall y \in \{0,1\}^*, (|y| = |x|) \Rightarrow (M(y) = 1 \iff y = x)\}$$

where M is DFA for $A_D(x)$ and NFA for $A_N(x)$.

[1] We just construct a working description and check its descriptive length.
[2] Even up to some constant factor.

Theorem 1.2.2.
(Shallit and Wang [SW01], Hyde and Kjos-Hanssen [Hyd13; HKH15])

- *For almost all strings* x, $A_D(x) \leq \frac{3}{4}|x| + \log(|x|)\sqrt{\frac{|x|}{8}}$
- $\exists$ *constant* C, *such that for almost all strings* x, $A_D(x) > C\frac{|x|}{\log(|x|)}$
- *For all strings* x, $A_N(x) \leq \frac{|x|}{2} + 1$
- *For almost all strings* x, $\forall b > 0, b \neq 1$, $A_N(x) > \sqrt{\frac{|x|}{\log_b(|x|)}}$

Under this definition, Hyde and Kjos-Hanssen also studied the complexity of square-free and cube-free words in [Hyd13; HKH15]. One of cube-free words they studied was the Morse-Thue sequence. Let us denote the infinite Morse-Thue sequence as MT, and its length n prefix as MT_n.

Theorem 1.2.3 (Hyde and Kjos-Hanssen, [HKH15])**.**

$$1 = \liminf_n \frac{A_N(n)}{|n|/2}$$

In [CP89], Champarnaud and Pin gave bounds to the number of states needed to represent a set of binary strings of length n. To be precise, they looked at $\Sigma = \{0, 1\}$ and defined $f(n) = \max\{\min\{|M| : M \text{ recognises } L\} : L \subseteq \Sigma^n\}$. One can interpret $f(n)$ as the upper bound on the number of states required to recognise sets of length n binary strings using minimal-sized DFA.

Theorem 1.2.4 (Champarnaud and Pin, [CP89])**.**

$$1 = \liminf_{n\to\infty} \frac{nf(n)}{2^n} \leq \limsup \frac{nf(n)}{2^n} = 2$$

Rearranging their result above and considering asymptotic bounds, we have that $f(n) \in \mathcal{O}(\frac{2^n}{n})$. This result was also noted in Section 3 of [LW14] by Leeuwen and Wiedermann. It is a similar bound as our upper bound analysis for DFA, as we will show later.

1.3 Our approach

There are many ways to represent set membership as shown below:

- Explicitly state the members (good if the set is finite)
  e.g. $A = \{1, 3, 7, 33, 65, 8693\}$

- Represent the members with some kind of generator/pattern
  e.g. $A = \{2x : x \in \mathbb{N}\}$

- Perform set operations on known/defined sets
  e.g. $A = \{2x : x \in \mathbb{N}\} \cap \{3x : x \in \mathbb{N}\}$

Here, we consider another way of doing so. We define $x[i] = \begin{cases} 0 & i \text{ is not in the set} \\ 1 & i \text{ is in the set} \end{cases}$

For example, $\{\text{Even numbers}\} = 10101010... = x$

For a given binary string x, Shallit, Wang, Hyde and Kjos-Hanssen looked at the bit-by-bit behaviour of x. As opposed to that, we will look at the index behaviour of x. We define the complexity of a given binary string x as the least number of states in an automata M such that when given $i \in \mathbb{N}$, M accepts i if the i^{th} bit of x is '1' and rejects i if the i^{th} bit of x is '0'[3]. In this perspective, an automata that represents x recognises the language of all the indices that are '1' in x. We provide a more formal definition in Section 1.4.

In this report, we consider both the deterministic finite automata (DFA) and non-deterministic finite automata (NFA). DFA are easier to analyze and compute bounds while NFA may allow us to save up to an exponential number of states. To differentiate the 2 types of automata choices and to avoid confusion with prior work's notations, we adopt the notations $D(x)$ and $N(x)$ for complexity with respect to DFA and NFA, as per our definitions, respectively.

After exploring DFA and NFA, it is only natural to consider the model of Context Free Grammar (CFG). We define string complexity with respect to CFG, denoted $C_k(x)$, later and compare it with $D(x)$ and $N(x)$.

[3] We ignore the behaviour of M on indices beyond the length of x.

1.4 Preliminaries

Automata

We can define an automaton by a tuple $(Q, \Sigma, \delta, q_0, F)$

where	Q	$\subseteq$	$\mathbb{N}$	is the set of states
	Σ	$=$	$\{0,1\}$	is the set of symbols
	δ	$:$	$Q \times \Sigma \to Q$	is the transition function
	q_0	$\in$	Q	is the initial/starting state
	F	$\subseteq$	Q	is the set of accepting/final states

The size of the automaton M, denoted by $|M|$, refers to the number of states it has (i.e. $|M| = |Q|$). We do not allow ϵ transitions. Suppose $q, q' \in Q$, $x \in \Sigma$, and $\delta(q, x) = q'$. Reading x at state q will cause the automaton to transit to state q'. If $(q, x) \notin \delta$, then reading x at state q will result in a *forever rejecting state*. For convenience, we sometimes leave this little detail out in our proofs and constructions. It will only result in an additional state in our bound analysis.

Enumeration sequences

In this report, we consider 3 different enumeration sequences:

(i) Natural numbers $\mathbb{N}$: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, ...\}$

(ii) Binary numbers $\mathbb{B}$: $\{0, 1, 10, 11, 100, 101, 110, 111, 1000, ...\}$

(iii) Length-lexicographical ordering $\mathbb{L}$: $\{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, ...\}$

There exists bijections $b : \mathbb{N} \to \mathbb{B}$ and $ll : \mathbb{N} \to \mathbb{L}$. Note that $\mathbb{B} \subset \{0,1\}^*$ and $\mathbb{L} = \{0,1\}^*$. We can convert between $\mathbb{B}$ and $\mathbb{L}$ as follows:

- To convert $b(n)$ to $ll(n)$:
  Binary-add 1 to $b(n)$, then throw away the most significant bit of '1'.
- To convert $ll(n)$ to $b(n)$:
  Add a most significant bit of '1', then binary-minus 1 from the result.

String operations

Let $x, y \in \{0,1\}^*$ be arbitrary binary strings, $i \in \mathbb{L}$ be an arbitrary index string, and $p, j, a, b \in \mathbb{N}$ be arbitrary natural numbers. Here are some string operations.

- Length
  $|x|$ represents the length of string x.
  For example, if $x = 101$, then $|x| = 3$.
  While this is the same $|\cdot|$ notation as the size of an automaton, there should not be any confusion based on context.

- Prefix
  x_n represents the length n prefix of x.
  For example, if $x = 101010...$, $x_3 = 101$.

- Concatenation
  xy represents the concatenation of x followed by y.
  For example, if $x = 1010, y = 0011$, then $xy = 10100011$.

- Repeat
  x^p represents the concatenation of p copies of x.
  For example, if $x = 110, p = 3$, then $x^p = 110110110$.

- Reverse
  x^R represents the reverse string of x.
  For example, if $x = 110$, then $x^R = 011$.

- Bit flip
  $\overline{x}$ denote the bit flip of the string x.
  For example, if $x = 11010$, then $\overline{x} = 00101$.

- Repeated bit flips
  Let $flip(x, n)$ be the value of x bit flipped n times.
  Then $flip(x, n) = \begin{cases} x & \text{if n is even} \\ \overline{x} & \text{if n is odd} \end{cases}$

- Character/Bit indexing
  $x[j]$ denotes the j^{th} character/bit in the string x. The first character is $x[0]$.

- Substring indexing
  $x[a : b] = x[a]x[a + 1]...x[b]$ denotes the substring of x from index a to index b, inclusive, with $a \leq b$.

- Automata reading
  $A(i)$ denotes an automata A reading the input i.
  Note that the domain which automata read from is $\mathbb{L}$.

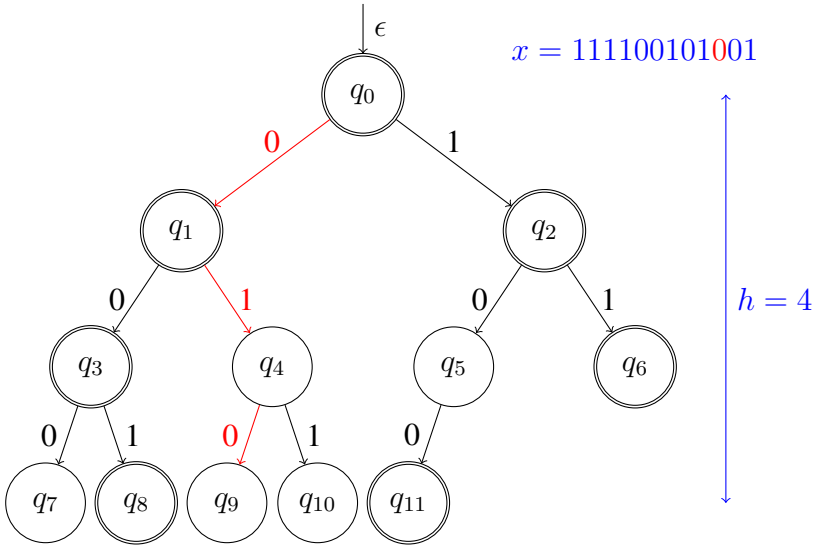


Figure 1.1: Trivial tree automaton of x

Trivial Tree Automata

For any given string x, we can build a *trivial tree automaton* T_x that represents x such that $T_x(ll(i)) = x[i], \forall i \in \mathbb{N}$. For each $i \in \mathbb{N}$, let q_i represent the i^{th} bit of x. We can visualize T_x as a binary tree such that $T_x(ll(i))$ reaches the state labelled q_i. In particular, the root node is labelled q_0. For example, consider $x = 111100101001$. Figure 1.1 shows the trivial tree automaton T_x (i.e. $T_{111100101001}$). The red path from the root q_0 gives us:

$$T_{111100101001}(ll(9)) = T_{111100101001}(010) = 0$$

This corresponds to $x[9] = 0$. Observe that the height h of a trivial tree automaton T_x is computed as follows: $h = 1 + \lfloor \log_2(|x|) \rfloor$, where $|x|$ is the length of x.

At any state q_n (for $n \in \mathbb{N}$):

- Reading a '0' makes a transition to q_{2n+1}
- Reading a '1' makes a transition to q_{2n+2}.

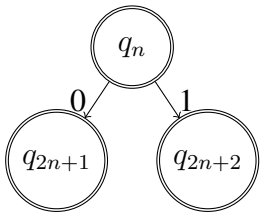


Figure 1.2: State transition in trivial tree automata

Useful properties about binary trees

- Height counting in trees start from 1.
  A tree with just the root node is of height 1.
- There are 2^{h-1} nodes at height h of a full binary tree of height $\geq h$.
- A full binary tree of height h has $2^0 + 2^1 + ... + 2^{h-1} = 2^h - 1$ nodes.

Formal definition of $D(x)$ and $N(x)$

Now, let us restate our definition of automatic complexity more formally.

Definition 1.4.1. (Produce)

An automaton M *produces* a binary string x if $M(ll(i)) = x[i], \forall i \in \mathbb{N}$.

The definition is clear when M is a DFA. If M is a NFA, $M(ll(i)) = 1$ if and only if there exists an accepting path from q_0 in M upon reading $ll(i)$.

Definition 1.4.2. (DFA complexity of binary string x)

$$D(x) = \min\{|M| : M \text{ is a DFA that produces } x\}$$

Definition 1.4.3. (NFA complexity of binary string x)

$$N(x) = \min\{|M| : M \text{ is a NFA that produces } x\}$$

Usage of trivial tree automata

By definition, the trivial tree automaton T_x of any x produces x.
We also know that the size of T_x is simply the size of x (i.e. $|T_x| = |x|$).

When x is finite, $|T_x| = |x|$ will be an upper bound on the complexity of x.
This is obvious since we can describe x by simply enumerating each digit.

When x is infinite, $|T_x|$ will be infinitely huge.
In this case, one may argue that it is foolish to use T_x to reason about the complexity of x. However, we have found that it is often helpful to draw out T_x, to some finite height, to visualise patterns. Such visualisations gave us valuable insights that lead to some of the results that we present later.

1.5 Outline of report

First, we will go through some theorems related to DFA and NFA on general binary strings. Next, we will see how they relate to CFG. Then, we will explore the automatic complexity of some interesting strings. Lastly, we end off with a discussion about how we wrote a computer program to empirically verify our bounds. At the same time, the program gives an empirical but numerically tighter bound for the strings we studied. The appendix covers side proofs and the language class of repeated subwords, which we explored briefly but did not gain sufficient traction to be worthy of a spot in the thesis itself.

Chapter 2

Basic results

In this section we show upper and lower bound complexities for general binary strings in both deterministic and non-deterministic automata. In our results, we parameterize the complexity of a binary string x by its length $|x|$. This makes sense if x has a finite length. If x is an infinite string, then apply our results to x_n. Our results agree with intuition that the bounds increases as $|x|$ increases.

2.1 Deterministic Finite Automata (DFA)

Upper bounds

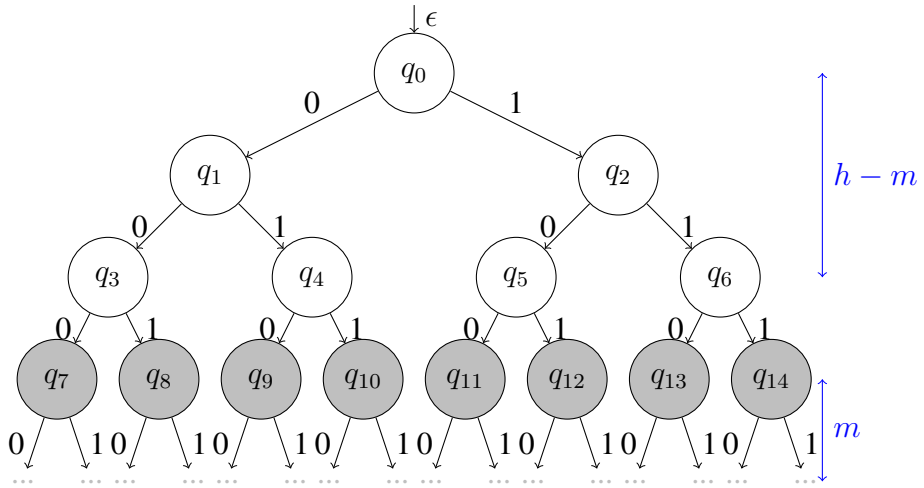
Proposition 2.1.1. ***(Trivial bound)***
For a fixed length n, for any string x of length n, $D(x) \leq |x|$.

Proof. The trivial automata T_x is a witnessing automata that fulfils the definition of $D(x)$, hence $D(x) \leq |T_x| = |x|$. □

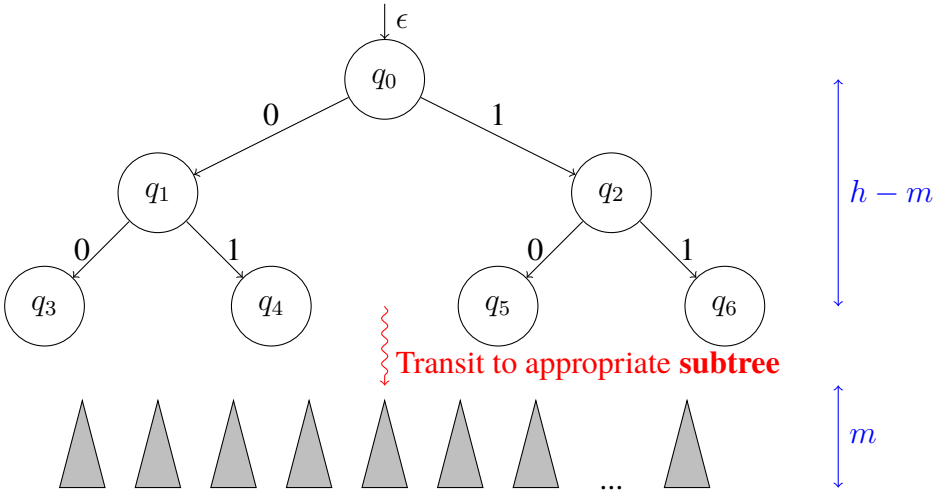
Theorem 2.1.2.
For a fixed length n, for any string x of length n, $D(x) \leq \frac{32 \cdot |x|}{\log_2(|x|)}$.

Proof. Recall that the height of T_x is $h = 1 + \lfloor \log_2(|x|) \rfloor$.
The key idea of the proof is that the lower half of the trivial automata is likely to repeat itself, so we only need to use one copy of each of the subtrees.

In the extreme case, consider just the lowest layer of the trivial automata. There are only 2 possible subtrees: a single accepting node (A) or a single rejecting node (R). However, there up to 2^{h-1} subtrees of height 1 in the lowest layer of T_x. So, if we remove all those subtrees and just have the upper layers point to



(a) The original T_x. We will *brute force* the gray states.



(b) Transformed automaton. We have 2^{2^m-1} possible subtrees (gray triangles).

Figure 2.1: Illustration for proof of DFA Upper bound

either A or R, we only need $2^{h-1} - 1 + 2$ states (potentially saving up to $2^{h-1} - 2$ states), reducing the upper bound for $D(x)$.

Generalising this approach, to reduce the upper bound for $D(x)$ via modification of T_x, we can find an optimal height m such that we can remove from T_x the lowest m layers and include all possible trees of height m.

After removing the bottom m layers of T_x, we have $2^{h-m} - 1$ nodes left. On the other hand, trees of height m have $2^m - 1$ nodes and there are 2^{2^m-1} possible such trees. This gives us a new upper bound $D(x) \leq 2^{h-m} - 1 + 2^{2^m-1}$.

Before we proceed to evaluate this inequality, let us first check the conditions for this approach to be meaningful. Clearly, we hope to remove more nodes than the new nodes required. i.e. $|x| - (2^{h-m} - 1) \geq 2^{2^m - 1}$. In the worst case, x has only 1 state at the lowest level and we remove only (1 + Number of nodes in a full trees of height $m - 1$) nodes.

If $h \geq 2^m + 1$ and $m \geq 2$, then in the worst case,

$$\begin{aligned} |x| - (2^{h-m} - 1) &\geq 1 + 2^{h-m} \cdot (\text{\# of nodes in a full tree of height } m-1) \\ &= 1 + 2^{h-m} \cdot (2^{m-1} - 1) \\ &= 1 + 2^{h-1} - 2^{h-m} \\ &= 1 + 2^{h-2} + 2^{h-2} - 2^{h-m} \\ &\geq 2^{h-2} \\ &\geq 2^{2^m - 1} \end{aligned}$$

There are 2 possible cases: (i) $2^{2^m-1} \geq 2^{h-m}$ and (ii) $2^{h-m} \geq 2^{2^m-1}$.
Consider the largest m such that these inequalities hold[1].

Case 1 ($2^{2^m-1} \geq 2^{h-m}$):

$$\begin{aligned} D(x) &\leq 2^{h-m} - 1 + 2^{2^m-1} \\ &\leq 2 \cdot 2^{2^m-1} && \text{, since } 2^{2^m-1} \geq 2^{h-m} \geq 2^{h-m} - 1 \\ &= 2^{2^m} \\ &\leq 2^h && \text{, since } h \geq 2^m + 1 \geq 2^m \\ &= 2^{1+\lfloor \log_2(|x|) \rfloor} && \text{, since } h = 1 + \lfloor \log_2(|x|) \rfloor \end{aligned}$$

But this does not yield new information since Proposition 2.1.1 already gives us $D(x) \leq |x| = 2^{\log_2(|x|)}$.

Case 2 ($2^{h-m} \geq 2^{2^m-1}$):

By the choice of m, $2^{h-(m+1)} \leq 2^{2^{m+1}-1}$.

$$\begin{aligned} 2^{h-m-1} &\leq 2^{2^{m+1}-1} \\ h - m - 1 &\leq 2^{m+1} - 1 \\ h &\leq 2^{m+1} + m \\ &\leq 2^{m+2} \\ m &\geq \log_2(h) - 2 \end{aligned}$$

Now, consider the upper bounds and lower bounds of h with respect to $\log_2(|x|)$.
We have $\log_2(|x|) \leq h = 1 + \lfloor \log_2(|x|) \rfloor \leq 1 + \log_2(|x|) \leq 2 \cdot \log_2(|x|)$.

[1] i.e. $m + 1$ will flip the inequality.

Then,

$$\begin{aligned}
D(x) &\leq 2^{h-m} - 1 + 2^{2^m-1} \\
&\leq 2 \cdot 2^{h-m} && \text{, since } 2^{h-m} \geq 2^{2^m-1} \geq 2^{2^m-1} - 1 \\
&= 2^{h-m+1} \\
&\leq 2^{h-(\log_2(h)-2)+1} && \text{, since } m \geq \log_2(h) - 2 \\
&= 2^{h-\log_2(h)+3} \\
&= 8 \cdot \frac{2^h}{h} \\
&\leq 8 \cdot \frac{2^{2 \cdot log_2(|x|)}}{\log_2(|x|)} && \text{, since } \log_2(|x|) \leq h \leq 2 \cdot \log_2(|x|) \\
&= \frac{32 \cdot |x|}{\log_2(|x|)}
\end{aligned}$$

Hence, for any given string x, we can modify T_x as above to yield $D(x) \leq \frac{32 \cdot |x|}{\log_2(|x|)}$ by picking m such that $2^{h-m} \geq 2^{2^m-1}$ and $h \geq 2^m + 1$. □

Lower bounds

It is not possible to give any meaningful lower bound for all general binary strings because for any length n. This is because there is a string of n 1's, which has a complexity of $D(1^n) = 1$. One such witnessing automata for $x = 1^n$ is simply an accepting state with self-loops.

Theorem 2.1.3.
For a fixed length n, there is a string x of length n such that $D(x) \geq \frac{|x|}{3 \cdot \log_2(|x|)}$.

Proof. We provide a simple counting argument as follows:
Consider all possible k-state automata $M = (\Sigma, Q, \delta, F)$. i.e. $|Q| = k$. From each state, each of the 0 or 1 transitions can take us to any of the k states (including itself). This yields $k^k \cdot k^k = k^{2k}$ possible transition functions. Then since every state could be accepting or rejecting, we multiply the result by 2^k. This gives us a total of $k^{2k} \cdot 2^k = (2^{\log_2(k)})^{2k} \cdot 2^k = 2^{2k \log_2(k)+k}$ possible k-state automata[2].

Now we fix k to be the smallest possible natural number such that $2^{2k \log_2(k)+k} \geq 2^{|x|}$. By the pigeonhole principle that we cannot uniquely represent all possible strings x of length n if we restrict ourselves to only automata with less than k states. i.e. There exists a string x of length n such that $D(x) \geq k$.

[2]Note that this already includes smaller sized automata since some permutations of the transitions may result in unreachable states, which effectively reduces the size of the automata implicitly.

Now, for that string x such that $D(x) \geq k$, we have:

$$\begin{aligned}
2^{2k\log_2(k)+k} &\geq 2^{|x|} && \text{, from above} \\
2k\log_2(k)+k &\geq |x| && \text{, taking logarithm base 2} \\
k(2\log_2(k)+1) &\geq |x| && \text{, factoring out k} \\
D(x)(1+2\log_2(D(x))) &\geq |x| && \text{, since } D(x) \geq k \text{ and log is monotonic} \\
D(x)(1+2\log_2(|x|)) &\geq |x| && \text{, since } D(x) \leq |x| \text{, by Proposition } 2.1.1 \\
D(x) &\geq \tfrac{|x|}{1+2\log_2(|x|)} \\
D(x) &\geq \tfrac{|x|}{3\cdot\log_2(|x|)} && \text{, since } \log_2(|x|) \geq 1
\end{aligned}$$

Therefore, for strings of length n, there is a string x such that $D(x) \geq \frac{|x|}{3\cdot\log_2(|x|)}$. □

2.2 Non-deterministic Finite Automata (NFA)

Upper bounds

Proposition 2.2.1. ***(Trivial bound)***
For a fixed length n, for any string x of length n, $N(x) \leq |x|$.

Proof. A DFA is also a NFA, so $N(x) \leq D(x) \leq |x|$. □

Theorem 2.2.2.
For a fixed length n, for any string x of length n, $N(x) \leq 8 \cdot \sqrt{|x|}$.

Proof. Similar to the DFA upper bound proof, we prove the bound by modifying the trivial tree automaton T_x. Here, we modify T_x as such:

For some $m \in \mathbb{N}$,

- Maintain the top $h - m$ levels of T_x to behave as a DFA
- Modify the bottom m levels of T_x to behave as a NFA (inverted tree)

$\forall i \in \mathbb{L}$ of length $\geq h - m$, we can view it as a concatenation of 2 shorter binary strings: $i = \sigma\tau$, where $|\sigma| = h - m$, $|\tau| = |i| - (h - m)$, and $\sigma, \tau \in \mathbb{L}$.

The idea here is to merge all similar τ together to reduce the number of states required. That is to say, we replace all nodes in the lower m levels of the T_x with states labelled τ, for $0 \leq |\tau| \leq m$. Since each bit of τ can be 0 or 1, there are $2^0 + 2^1 + ... + 2^m = 2^{m+1} - 1$ such states in total.

For any 2 such nodes τ_1 and τ_2, $\delta(\tau_1, a) = \tau_2 \iff \tau_1 = a\tau_2$, for $a \in \{0, 1\}$. The only accepting state is the state labelled ϵ (i.e. The case when $|\tau| = 0$).

For every $i \in \mathbb{L}$ of length $\geq h - m$ with $x[ll^{-1}(i)] = 1$ and $i = \sigma\tau$ (as mentioned above), we make the appropriate '0' or '1' transition from the node at height $h - m$ to the node labelled τ in the lower part.

Check:

Every $i \in \mathbb{L}$ of length $< h - m$ behaves correctly by definition of T_x.
For every $i \in \mathbb{L}$ of length $\geq h - m$, by construction:
If $x[ll^{-1}(i)] = 1$, the automaton will reach the ϵ accepting state after reading i.
If $x[ll^{-1}(i)] = 0$, there is no transition path from the σ nodes to the τ nodes.

By removing the bottom m levels of T_x, we save on $|x| - (2^{h-m} - 1)$ nodes and are left with $2^{h-m} - 1$ nodes. We then added $(2^{m+1} - 1)$ τ nodes, giving us $N(x) \leq 2^{h-m} - 1 + 2^{m+1} - 1 = 2^{h-m} + 2^{m+1} - 2$. For the construction to make sense, we need $|x| - (2^{h-m} - 1) \geq 2^{m+1} - 1$. In the worst case, x has only 1 state at the lowest level and we remove only (1 + Number of nodes in a full trees of height $m - 1$) nodes.

If $h \geq m + 3$ and $m \geq 2$, then in the worst case,

$$\begin{aligned}
|x| - (2^{h-m} - 1) &\geq 1 + 2^{h-m} \cdot (\text{\# of nodes in a full tree of height } m - 1) \\
&= 1 + 2^{h-m} \cdot (2^{m-1} - 1) \\
&= 1 + 2^{h-1} - 2^{h-m} \\
&= 1 + 2^{h-2} + 2^{h-2} - 2^{h-m} \\
&\geq 2^{h-2} \\
&\geq 2^{m+1} \\
&\geq 2^{m+1} - 1
\end{aligned}$$

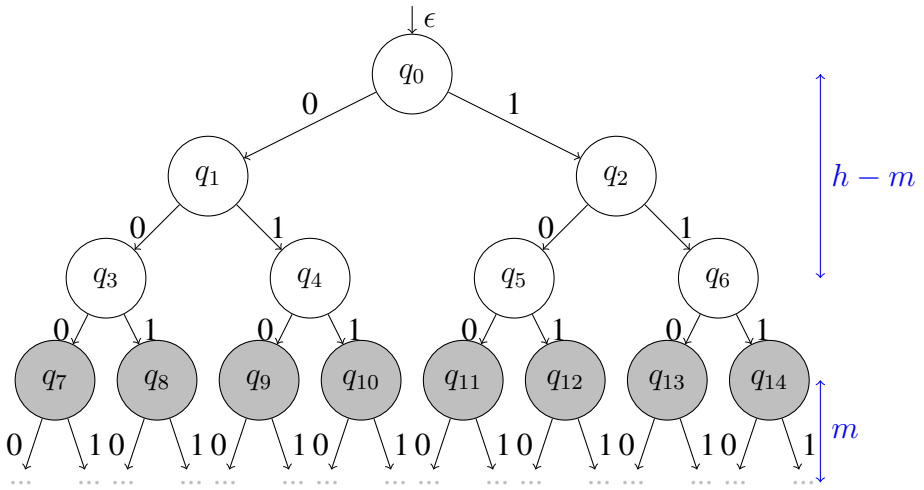
To simplify analysis, consider m that roughly divides the 2 portions into half, where $m + 1$ will switch the relative sizes.

There are 2 possible cases: (i) $2^{m+1} \geq 2^{h-m}$ and (ii) $2^{h-m} \geq 2^{m+1}$.
Consider the largest m such that these inequalities hold[3].

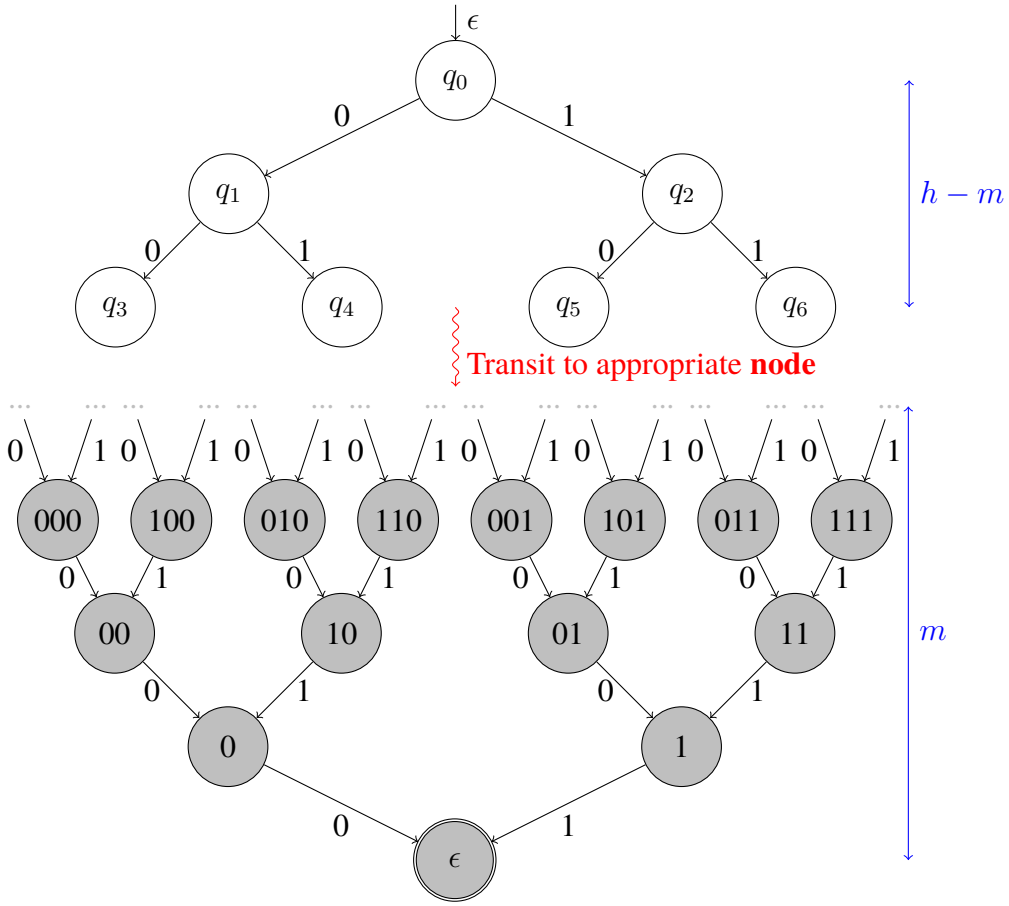
Case 1 ($2^{m+1} \geq 2^{h-m}$):

$$\begin{aligned}
N(x) &\leq 2^{h-m} + 2^{m+1} - 2 \\
&\leq 2 \cdot 2^{m+1} && \text{, since } 2^{m+1} \geq 2^{h-m} \geq 2^{h-m} - 2 \\
&\leq 2^{h} && \text{, since } h \geq m + 3 \geq m + 2 \\
&= 2^{1+\lfloor \log_2(|x|) \rfloor} && \text{, since } h = 1 + \lfloor \log_2(|x|) \rfloor
\end{aligned}$$

[3] i.e. $m + 1$ will flip the inequality.



(a) The original T_x. We will turn the gray states into a NFA.



(b) Transformed automaton. We have $2^{m+1} - 1$ nodes in the inverted gray subtree.

Figure 2.2: Illustration for proof of NFA Upper bound

But this does not yield new information since Proposition 2.2.1 already gives us $N(x) \leq |x| = 2^{\log_2(|x|)}$.

Case 2 ($2^{h-m} \geq 2^{m+1}$):
By the choice of m, $2^{h-(m+1)} \leq 2^{m+2}$.

$$\begin{aligned} 2^{h-(m+1)} &\leq 2^{m+2} \\ h-m-1 &\leq m+2 \\ m &\geq \tfrac{h-3}{2} \end{aligned}$$

Then,

$$\begin{aligned} N(x) &\leq 2^{h-m} + 2^{m+1} - 2 \\ &\leq 2 \cdot 2^{h-m} && \text{, since } 2^{h-m} \geq 2^{m+1} \geq 2^{m+1} - 2 \\ &= 2^{h-m+1} \\ &\leq 2^{h-\frac{h-3}{2}+1} && \text{, since } m \geq \tfrac{h-3}{2} \text{, from above} \\ &= 2^{\frac{h+5}{2}} \\ &= 2^{\frac{6+\lfloor \log_2(|x|) \rfloor}{2}} && \text{, since } h = 1 + \lfloor \log_2(|x|) \rfloor \\ &= 2^{3+\frac{\lfloor \log_2(|x|) \rfloor}{2}} \\ &\leq 2^{3+\frac{\log_2(|x|)}{2}} \\ &= 8 \cdot \sqrt{|x|} \end{aligned}$$

Hence, for any given string x, we can modify T_x as above to yield $N(x) \leq 8 \cdot \sqrt{|x|}$ by picking m such that $2^{h-m} \geq 2^{m+1}$ and $h \geq m+3$. □

Lower bounds

Similar to the analysis of DFA, it is not possible to give any meaningful lower bound for all general binary strings. This is because for any length n, there is a string of n 1's, which has a complexity of $N(1^n) = 1$.

Proposition 2.2.3. *For a fixed length $n \geq 1$, there is a string x of length n such that $N(x) \geq \sqrt{\frac{|x|}{3}}$.*

Proof. For NFA, there are 4 possible transitions $(\emptyset, \{0\}, \{1\}, \{0,1\})$. For every 2 states, exactly 1 of the 4 transitions must occur. So, with k states, we have $2 \cdot 4^{\binom{k}{2}}$ possible transition functions (both directions). Then since every state could be accepting or rejecting, we get a total of $2^{k+1} \cdot 4^{\binom{k}{2}} \leq 2^k \cdot 4^{k^2} = 2^{k+2k^2}$ different possible NFA.[4] Fix k to be the smallest possible natural number such

[4]Note that this already includes smaller sized automata since some permutations of the transitions may result in unreachable states, which effectively reduces the size of the automata implicitly.

that $2^{k+2k^2} \geq 2^{|x|}$. Apply the pigeonhole principle. For the string x such that $N(x) \geq k$, we have:

$$
\begin{array}{rcll}
2^{k+2k^2} & \geq & 2^{|x|} & \text{, from above} \\
k + 2k^2 & \geq & |x| & \text{, taking logarithm base 2} \\
N(x) + 2(N(x))^2 & \geq & |x| & \text{, since } N(x) \geq k \\
3(N(x))^2 & \geq & |x| & \text{, since } (N(x))^2 \geq N(x) \\
(N(x))^2 & \geq & \frac{|x|}{3} & \\
N(x) \geq \sqrt{\frac{|x|}{3}} & \text{, since } N(x) \geq 0 & &
\end{array}
$$

□

Chapter 3

Context Free Grammar

3.1 Definitions

After looking at DFA and NFA, the natural extension is to consider the complexity with respect to context free grammars.

We can define a context free grammar (CFG) by a tuple (NT, T, R, S),

where	NT	$=$	$\{a_i : i \in \mathbb{N}\}$	is the set of non-terminals
	T	$=$	$\{\epsilon, 0, 1\}$	is the set of terminals
	R	$:$	$NT \to (NT \cup T)^*$	is the set of production rules
	S	$\in$	NT	is the starting symbol

Since we are considering binary strings x, the set of terminals T is fixed to $\{0, 1\}$. So, any useful notion of complexity has to deal with either NT and/or R. In this report, we explore a definition with respect to the set of non-terminals NT.

Before we proceed with the formal definition, notice that the production rules can produce arbitrary amount of symbols on the right hand side. That is bad since given any x, we can have a CFG with only one non-terminal producing the entire string (i.e. $a_0 \to x$). This prevents us from having any useful analysis regarding CFGs. To rectify this, we limit the number of symbols that can appear on the right hand side of any production rules.

Definition 3.1.1. (k-bounded CFG)
For $k \geq 2$, a context free grammar is k-bounded if for every rule in R, the rules produce at most k symbols. i.e. If we have a rule $A \to w$, then $|w| \leq k$.

Now, we define the k-bounded CFG complexity $C_k(x)$ of a binary string x in a similar fashion as our definitions $D(x)$ and $N(x)$.

Definition 3.1.2. (Produce)
A k-bounded CFG $G = (NT, T, R, S)$ *produces* a binary string x if

$$S \Rightarrow^* ll(i) \iff x[i] = 1, \forall i \in \mathbb{N}$$

Definition 3.1.3. (k-bounded CFG complexity of binary string x)

$$C_k(x) = \min\{|NT| : G \text{ is a } k\text{-bounded CFG that generates } x\}$$

3.2 Upper bounds

Upper bounds for relatively short x is trivial. The bulk of the proof will work on sufficiently long x with respect to k. The idea is to generate all possible strings up to length $\max_i\{\lceil \frac{|ll(i)|}{k+1} \rceil\}$, then form up the rules using those generators. We first illustrate with an example before we proceed with the proof.

An example

Let $k = 2$, $|x| = 65$ and
$x = 10000000001110101111010110010000110011110111100000101000010111110$.

Figure 3.1 shows us the indices that the CFG has to capture for string x. In this example, $\max_i\{\lceil |ll(i)| \rceil\} = 6$, so $\max_i\{\lceil \frac{|ll(i)|}{k+1} \rceil\}$ = 3. Therefore,we will have the generator rules A_j and B_j for $j = \{\epsilon, 0, 1, 00, 01, 10, 11\}$ such that

- $A_j \Rightarrow^* j$
- $A_jB_j \Rightarrow^* ll(i) \iff ll(i)$ is some index to be captured

For each index $ll(i)$ to be captured, we split it into $k + 1$ substrings. As a convention, we increase the length of the substrings from the right to the left, as its length increases. Here, we need at most 1 S, $(1+2+4) = 7$ A_j and $(1+2+4) = 7$ B_j symbols. In fact, we only need $S, A_\epsilon, A_0, A_1, A_{00}, A_{01}, A_{10}, A_{11}, B_\epsilon, B_0, B_1, B_{10}$.

Hence, we have $C_2(x) \leq 12$ via:

$$
\begin{array}{rcll}
R & = & \{ & A_\epsilon \to \epsilon, \quad A_0 \to 0, \quad A_1 \to 1, \\
& & & A_{00} \to 00, \quad A_{01} \to 01, \quad A_{10} \to 10, \quad A_{11} \to 11, \\
& & & \\
& & & S \to A_\epsilon B_\epsilon, \quad B_\epsilon \to A_\epsilon, \\
& & & S \to A_0 B_0, \quad S \to A_1 B_1, \quad S \to A_{10} B_{10}, \\
& & & \\
& & & B_0 \to A_1 A_1, \quad B_0 \to A_0 A_{01}, \quad B_0 \to A_0 A_{10}, \quad B_0 \to A_0 A_{11}, \\
& & & B_0 \to A_1 A_{00}, \quad B_0 \to A_1 A_{10}, \quad B_0 \to A_{00} A_{01}, \quad B_0 \to A_{00} A_{10}, \\
& & & B_0 \to A_{01} A_{01}, \quad B_0 \to A_0 A_{110}, \quad B_0 \to A_{01} A_{11}, \quad B_0 \to A_{10} A_{00}, \\
& & & B_0 \to A_{10} A_{10}, \quad B_0 \to A_1 A_{011}, \quad B_0 \to A_{11} A_{00}, \quad B_0 \to A_{11} A_{01}, \\
& & & \\
& & & B_1 \to A_0 A_0, \quad B_1 \to A_0 A_1, \quad B_1 \to A_1 A_1, \quad B_1 \to A_0 A_{00}, \\
& & & B_1 \to A_0 A_{01}, \quad B_1 \to A_1 A_{00}, \quad B_1 \to A_{00} A_{11}, \quad B_1 \to A_{01} A_{01}, \\
& & & B_1 \to A_{10} A_{10}, \quad B_1 \to A_{11} A_{00}, \quad B_1 \to A_{11} A_{01}, \quad B_1 \to A_{11} A_{10}, \\
& & & B_1 \to A_{11} A_{11} \quad B_{10} \to A_{00} A_{00} \\
& & \} &
\end{array}
$$

Theorem 3.2.1. *For any given string x, $C_k(x) \leq 8 \cdot \sqrt[k+1]{|x|}$.*

Proof.
Let m be the maximum length of any index of x: $m = \lceil \log_2(|x|) \rceil$. After splitting it into $k+1$ segments, let $s = \max_j |A_j| = \lceil \frac{m}{k+1} \rceil$. We create a A_j rule for each possible length $\leq s$. Depending on the string x, we may need at most one B_j for each A_j. Hence, total number of symbols needed:

$$
\begin{array}{rcl}
S + \text{Number of } A_j + \text{Number of } B_j & = & 1 + 2 \cdot (1 + 2 + 2^2 + ... + 2^s) \\
& = & 1 + 2 \cdot (2^{s+1} - 1) \\
& = & 2 \cdot 2^{\lceil \frac{m}{k+1} \rceil} - 1 \\
& \leq & 2 \cdot 2^{\frac{m}{k+1}+1} \\
& = & 2 \cdot 2^{\frac{\lceil \log_2(|x|) \rceil}{k+1}+1} \\
& \leq & 2 \cdot 2^{\frac{\log_2(|x|)+1}{k+1}+1} \\
& = & 2 \cdot 2^{\frac{\log_2(|x|)}{k+1}+\frac{k+2}{k+1}} \\
& \leq & 2 \cdot 2^{\frac{\log_2(|x|)}{k+1}+2} \\
& = & 8 \cdot \sqrt[k+1]{|x|}
\end{array}
$$

□

In the example above, we have $k = 2$, $|x| = 65$, and
$C_2(x) \leq 12 \leq 8 \cdot \sqrt[3]{|x|} = 32.16580606871246$.

i	$\rightarrow$	$ll(i)$		S	$\Rightarrow^*$	$ll(i)$
0	$\rightarrow$	ϵ		S	$\Rightarrow^*$	ϵ
10	$\rightarrow$	011		S	$\Rightarrow^*$	0 1 1
11	$\rightarrow$	100		S	$\Rightarrow^*$	0 0 01
12	$\rightarrow$	101		S	$\Rightarrow^*$	0 0 10
14	$\rightarrow$	111		S	$\Rightarrow^*$	0 0 11
16	$\rightarrow$	0001		S	$\Rightarrow^*$	0 1 00
17	$\rightarrow$	0010		S	$\Rightarrow^*$	0 1 10
18	$\rightarrow$	0011		S	$\Rightarrow^*$	0 00 01
19	$\rightarrow$	0100		S	$\Rightarrow^*$	0 00 10
21	$\rightarrow$	0110		S	$\Rightarrow^*$	0 01 01
23	$\rightarrow$	1000		S	$\Rightarrow^*$	0 01 10
24	$\rightarrow$	1001		S	$\Rightarrow^*$	0 01 11
27	$\rightarrow$	1100		S	$\Rightarrow^*$	0 10 00
32	$\rightarrow$	00001		S	$\Rightarrow^*$	0 10 10
33	$\rightarrow$	00010		S	$\Rightarrow^*$	0 10 11
36	$\rightarrow$	00101	$\Rightarrow$	S	$\Rightarrow^*$	0 11 00
37	$\rightarrow$	00110		S	$\Rightarrow^*$	0 11 01
38	$\rightarrow$	00111		S	$\Rightarrow^*$	1 0 0
39	$\rightarrow$	01000		S	$\Rightarrow^*$	1 0 1
41	$\rightarrow$	01010		S	$\Rightarrow^*$	1 1 1
42	$\rightarrow$	01011		S	$\Rightarrow^*$	1 0 00
43	$\rightarrow$	01100		S	$\Rightarrow^*$	1 0 01
44	$\rightarrow$	01101		S	$\Rightarrow^*$	1 1 00
50	$\rightarrow$	10011		S	$\Rightarrow^*$	1 00 11
52	$\rightarrow$	10101		S	$\Rightarrow^*$	1 01 01
57	$\rightarrow$	11010		S	$\Rightarrow^*$	1 10 10
59	$\rightarrow$	11100		S	$\Rightarrow^*$	1 11 00
60	$\rightarrow$	11101		S	$\Rightarrow^*$	1 11 01
61	$\rightarrow$	11110		S	$\Rightarrow^*$	1 11 10
62	$\rightarrow$	11111		S	$\Rightarrow^*$	1 11 11
63	$\rightarrow$	100000		S	$\Rightarrow^*$	10 00 00

Figure 3.1: Indices to capture for x in CFG example

3.3 Lower bounds

It is not possible to give any meaningful lower bound for all general binary strings. This is because for any length n, there is a string of n 1's, which has a complexity of $C_k(1^n) = 1$. For $k \geq 2$, $x = 1^n$ can be produced by a k-bounded CFG with:

$$R = \{S \to \epsilon, S \to 0, S \to 1, S \to S0, S \to S1\}$$

Theorem 3.3.1. *For a fixed length n, there is a string x of length n such that $|x| = n$ and $C_k(x) \geq \sqrt[k+1]{|x|}$.*

Proof. Let $|NT| = c$. For every non-terminal $a \in NT$, we have $a \to (NT \cup N)^i$, for $i \in \mathbb{N}, 0 \leq i \leq k$. Let us count the number of possible production rules with c non-terminal symbols.

$$\begin{aligned} c \cdot (1 + (c+2) + (c+2)^2 + ... + (c+2)^k) &= c\tfrac{(c+2)^{k+1}}{c+1} \\ &\leq (c+2)^{k+1} \\ &\leq 2c^{k+1} \qquad \text{, for } c \geq 2 \end{aligned}$$

With $|NT| = c$, there are at most $2^{2c^{k+1}}$ possible CFG. We fix c to be the smallest possible natural number such that $2^{2c^{k+1}} \geq 2^{|x|}$. Apply the pigeonhole principle. For the string x such that $C_k(x) \geq c$, we have:

$$\begin{aligned} 2^{2c^{k+1}} &\geq 2^{|x|} && \text{, from above} \\ 2c^{k+1} &\geq |x| && \text{, taking logarithm base 2} \\ c &\geq \sqrt[k+1]{\tfrac{|x|}{2}} && \text{, taking root base } k+1 \\ C_k(x) &\geq \sqrt[k+1]{\tfrac{|x|}{2}} && \text{, since } C_k(x) \leq c \end{aligned}$$

□

3.4 Comparison with automatic complexity

	Lower bound (existential)	Upper bound (for all strings)
DFA	$D(x) \geq \frac{\lvert x\rvert}{3 \cdot \log_2(\lvert x\rvert)}$	$D(x) \leq \frac{32 \cdot \lvert x\rvert}{\log_2(\lvert x\rvert)}$
NFA	$N(x) \geq \sqrt{\frac{\lvert x\rvert}{3}}$	$N(x) \leq 8 \cdot \sqrt{\lvert x\rvert}$
CFG	$C_k(x) \geq \sqrt[k+1]{\frac{\lvert x\rvert}{2}}$	$C_k(x) \leq 8 \cdot \sqrt[k+1]{\lvert x\rvert}$

Chapter 4

Specific problems

4.1 Morse-Thue sequence

The Morse-Thue sequence is a sequence with various interesting properties.[1] Let us denote the infinite Morse-Thue sequence by the string MT. In general, one can pick $MT[0]$ to be 0 or 1. Without loss of generality, we choose $MT[0] = 0$. We highlight 2 common ways to generate the Morse-Thue sequence. We will use the second method to prove our theorem later.

Method 1 (Bit flipping)

For $n \in \mathbb{N}$,

$$\begin{aligned} MT[0] &= 0 \\ MT[2^n : 2^{n+1} - 1] &= \overline{MT[0 : 2^n - 1]} \end{aligned}$$

For example,

$$\begin{array}{lllllllll} n = 0 & : & MT[2^0 : 2^1 - 1] & = & MT[1 : 1] & = & \overline{MT[0 : 0]} & = & 1 \\ n = 1 & : & MT[2^1 : 2^2 - 1] & = & MT[2 : 3] & = & \overline{MT[0 : 1]} & = & 10 \\ n = 2 & : & MT[2^2 : 2^3 - 1] & = & MT[4 : 7] & = & \overline{MT[0 : 3]} & = & 1001 \\ n = 3 & : & MT[2^3 : 2^4 - 1] & = & MT[8 : 15] & = & \overline{MT[0 : 7]} & = & 10010110 \end{array}$$

So, $MT = 0|1|10|1001|10010110... = 0110100110010110....$

[1] We point readers to `http://en.wikipedia.org/wiki/Thue-Morse_sequence` and `http://mathworld.wolfram.com/Thue-MorseSequence.html` to learn more about the properties of the Morse-Thue sequence.

Method 2 (Grammar generation)

For $n \in \mathbb{N}$,

$$\begin{aligned} MT[0] &= 0 \\ MT[2n] &= MT[n] \\ MT[2n+1] &= 1 - MT[n] = \overline{MT[n]} \end{aligned}$$

Consider the following DFA A_{MT} which outputs '1' at accepting states B and C, and '0' at rejecting states A and D.

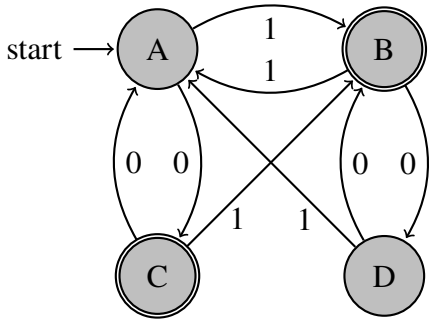


Figure 4.1: Automaton A_{MT} that produces the Morse-Thue sequence MT

Theorem 4.1.1. *The automaton A_{MT} produces the Morse-Thue sequence MT. That is to say $A_{MT}(ll(n)) = MT[n], \forall n \in \mathbb{N}$.*

Proof. We prove by induction on the number of bits of the index read. In this proof, we refer to the second method of construction of MT and convert indices $n \in \mathbb{N}$ between the other 2 enumerations systems: $\mathbb{B}$ and $\mathbb{L}$

Base cases:

Reading 0 bit: $A_{MT}(\epsilon) = MT[0] = 0$.
Reading 1 bit[2]: $A_{MT}(0) = MT[1] = 1$ and $A_{MT}(1) = MT[2] = 1$.
For any larger $n \in \mathbb{N}$, we can always write $b(n) = 1xy$, for some $x, y \in \{0,1\}^*$.

Induction hypothesis:

For some $N_0 \in \mathbb{N}$, for every $n \in \mathbb{N}, n \leq N_0$, we have $A_{MT}(ll(n)) = MT[n]$.

Inductive cases:

It suffices to check[3]:

[2]We included "reading 1 bit" in the base case to make the proof neater later on.
[3]As mentioned in Figure 1.1, $A_x(ll(n)0) = x[2n+1]$ and $A_x(ll(n)1) = x[2n+2], \forall n \in \mathbb{N}$.

(i) $A_{MT}(ll(n)0) = MT[2n+1]$

(ii) $A_{MT}(ll(n)1) = MT[2n+2]$

The proof for (i) is straightforward.

Referring to MT construction method 2, we see that $MT[2n+1] = \overline{MT[n]}$. This is reflected in A_{MT}: At any state, reading a '0' will cause it to transit from an accepting state to a rejecting state, or vice versa. So, (i) holds for all $n \in \mathbb{N}$.

For (ii), we split into 2 cases when k is an even or odd number.

We first calculate the expected behaviour of A_{MT} based on definition of the Morse-Thue sequence and verify that A_{MT} exhibits the predicted behaviour.

Case 1: n is even (i.e. $b(n)$ ends with a '0')

$$\begin{aligned}
A_{MT}(ll(n)1) &= A_{MT}(ll(2k)1) && \text{, for some } k \in \mathbb{N} \\
&= A_{MT}(ll(4k+2)) && \text{, since } A_{MT}(ll(n)1) = A_{MT}(ll(2n+2)) \\
&= A_{MT}(ll(2(2k+1))) \\
&= A_{MT}(ll(2k+1)) && \text{, since } MT[2n] = MT[n] \\
&= 1 - A_{MT}(ll(k)) && \text{, since } MT[2n+1] = 1 - MT[n] \\
&= 1 - A_{MT}(ll(2k)) && \text{, since } MT[2n] = MT[n] \\
&= 1 - A_{MT}(ll(n)) && \text{, since } n = 2k
\end{aligned}$$

This is reflected in A_{MT}. Since n is an even number, $ll(n)$ ends with a '1'. When A_{MT} reads n, it lands in either State A or B. In either case, reading another single '1' will switch it from a rejecting state to an accepting state, and vice versa.

Case 2: n is odd (i.e. $b(n)$ ends with a '1')
Define $f(n) = \frac{n-1}{2}$ and $g(n) = \frac{n}{2}$, $\forall n \in \mathbb{N}$. f should only be executed on odd numbers and g only on even numbers. In binary, both f and g essentially throws away the least significant bit of $b(n)$. Intuitively, f and g reverses the notion of $n \to 2n+1$ and $n \to 2n$ respectively. This means $MT[f^k(n)] = flip(MT[n], k)$ and $MT[g^k(n)] = MT[n]$, for appropriate values of n, $\forall k \in \mathbb{N}$.

Case 2a: $b(n) = 1^j$, for some $j \in \mathbb{N} \setminus \{0\}$.
Then, $b(2n+2) = 10^{j+1}$. We will relate $MT[n]$ to $MT[2n+2]$ via $MT[0]$[4].

Since $b(n) = 1^j$, $b(f^j(n)) = 0$.
So, $MT[n] = flip(MT[0], j)$.

Since $b(2n+2) = 10^{j+1}$, $b(f(g^{j+1}(2n+2))) = 0$.
So, $MT[2n+2] = flip(MT[0], 1)$.

[4]Recall that $b(0) = 0 = ll(0)$.

Therefore, $MT[n] = MT[2n+2] \iff j$ is odd.

We verify that this is reflected in A_{MT}:

- $b(n) = 1^j \Rightarrow ll(n) = 0^j$
- $b(2n+2) = 10^{j+1} \Rightarrow ll(n) = 0^j 1$

If j is odd, after reading j '0's, A_{MT} will be in State C.
Reading another '1' will cause it to transit to State B.
The output of A_{MT} are the same at states B and C.

If j is even, after reading j '0's, A_{MT} will be in State A.
Reading another '1' will cause it to transit to State B.
The output of A_{MT} are different at states A and B.

Case 2b: $b(n) = 1x01^j$, for some $x \in \{0,1\}^*, j \in \mathbb{N} \setminus \{0\}$.
Then, $b(2n+2) = 1x10^{j+1}$. We will relate $MT[n]$ to $MT[2n+2]$ via $MT[b^{-1}(1x)]$.

Since $b(n) = 1x01^j$, $b(g(f^j(n))) = 1x$.
So, $MT[n] = flip(MT[b^{-1}(1x)], j)$.

Since $b(2n+2) = 1x10^{j+1}$, $b(f(g^{j+1}(n))) = 1x$.
So, $MT[n] = flip(MT[b^{-1}(1x)], 1)$.

Therefore, $MT[n] = MT[2n+2] \iff j$ is odd.

We verify that this is reflected in A_{MT}:

- $b(n) = 1x01^j \Rightarrow ll(n) = x10^j$
- $b(2n+2) = 1x10^{j+1} \Rightarrow ll(n) = x10^j 1$

After reading some arbitrary string $x1$, A_{MT} is either in states A or B.

- If $A_{MT}(x1)$ reaches state A and j is odd,
  then $A_{MT}(x10^j)$ reaches state C and $A_{MT}(x10^j)$ reaches state B.
  The output of A_{MT} are the same at states C and B.
- If $A_{MT}(x1)$ reaches state B and j is odd,
  then $A_{MT}(x10^j)$ reaches state D and $A_{MT}(x10^j)$ reaches state A.
  The output of A_{MT} are the same at states D and A.
- If $A_{MT}(x1)$ reaches state A and j is even,
  then $A_{MT}(x10^j)$ reaches state A and $A_{MT}(x10^j)$ reaches state B.
  The output of A_{MT} are different at states A and B.

- If $A_{MT}(x1)$ reaches state B and j is even,
  then $A_{MT}(x10^j)$ reaches state B and $A_{MT}(x10^j)$ reaches state A.
  The output of A_{MT} are different at states B and A.

□

Corollary 4.1.2. $\forall n \in \mathbb{N}, n \geq 4, D(MT_n) \leq 4$

Proof. A_{MT} in Theorem 4.1.1 is a witnessing automaton for the Morse-Thue sequence MT. □

It is interesting to note that under the complexity definition of A_N, Theorem 1.2.3 tells us that the complexity of the infinite Morse-Thue sequence increases linearly with respect to the length of the prefix of interest. Meanwhile, in our notion of automatic complexity, the infinite Morse-Thue sequence only has a finite complexity, regardless of the length of prefix of interest.

4.2 Palindromes

The language of palindromes $A_P = \{wyw^R : w \in \{0,1\}^*, y \in \{\epsilon, 0, 1\}\}$ is a well-known context free grammar that can only be represented by a machine at least as complex as a non-deterministic push-down automata.

In this section, we consider the string pal:

$$pal[i] = 1 \iff ll(i) \text{ is a palindrome (i.e. } ll(i) \in A_P)$$

Definition 4.2.1. (Pivot element)
For a palindrome $wyw^R \in A_P$, we denote y as the pivot element of wyw^R. Then, $y \in \{\epsilon, 0, 1\}$.

Definition 4.2.2. (Trailing set of σ)
Given any non-empty prefix $\sigma \in \{0,1\}^+$, $TS_\sigma = \{\tau : \sigma\tau \in A_P\}$ is the trailing set of σ. By definition of palindromes, the length of every element in TS_σ is unique[5]. There is a total ordering on the elements of TS_σ via the subset relation. Denote the maximum element by τ^*, then $\forall \tau \in TS_\sigma, \tau \subseteq \tau^*$.

[5]For any non-empty string w, $w0$ and $w1$ cannot both be palindromes.

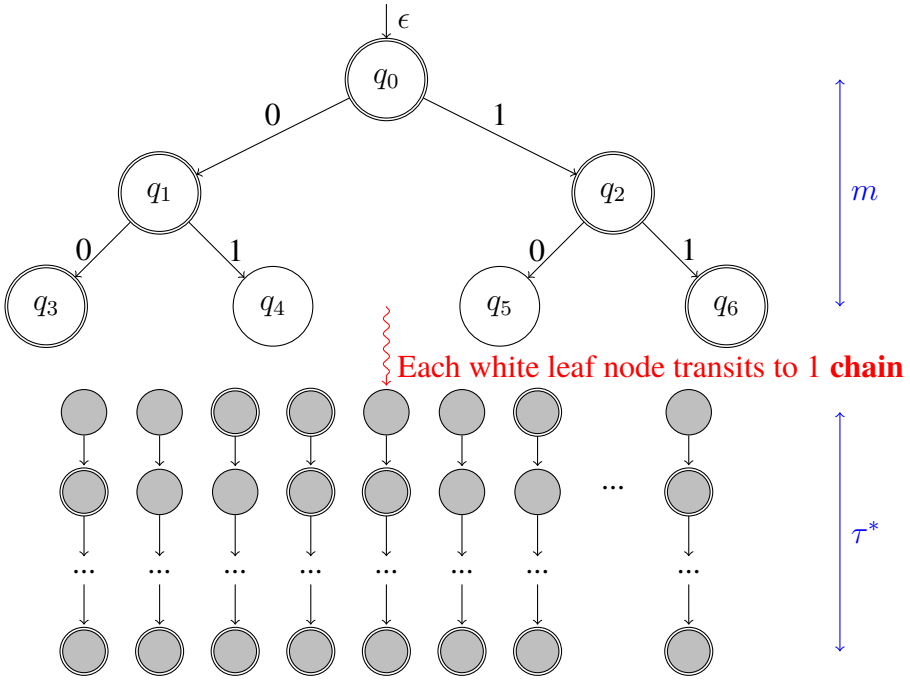


Figure 4.2: Illustration for proof of easy palindrome upper bound. Each white leaf transits to a gray chain of length at most τ^*. Note that the gray chains do not necessarily have the same length. In this example, $U = \{q_3, q_4, q_5, q_6\}$.

Upper bound complexities

We first show a simple upper bound and then proceed to show a tighter upper bound that is in the same order of growth as the lower bound.

Proposition 4.2.3. *For a fixed length* $n \geq 1$, $D(pal_n) \leq 2\log_2(n)\sqrt{n}$

Proof. Let $m = \lceil \frac{\log_2(n)}{2} \rceil$. We consider the trivial tree automaton T_{pal_n}. We preserve the top m levels of the tree and modify the bottom $h - m$ levels. Note that the pivot element lies in the preserved top m levels.

Let U be the set of nodes at the lowest unmodified level of T_{pal_n} (i.e. the nodes at height m from the root). For each $u \in U$, let σ_u be the string used to reach u from the root and TS_{σ_u} be the corresponding trailing set. We know that $\tau_u^* \leq h - m, \forall u \in U$.

Now, for each $u \in U$, we extend a sequence of transitions according to τ_u^*. Along this extension, we make the nodes an accepting state if and only if they extend

u to form a palindrome. For any other transitions, we will simply transit to a forever rejecting state.

There are 2^m nodes at height m and each τ^* is at most length $h-m$. Including the forever rejecting state, we need the following number of states in our construction:

$$\begin{aligned}(h-m)\cdot 2^m+1 &= (1+\lfloor\log_2(n)\rfloor-\lceil\tfrac{\log_2(n)}{2}\rceil)\cdot 2^{\lceil\frac{\log_2(n)}{2}\rceil}+1\\ &\leq \log_2(n)\cdot 2^{\frac{\log_2(n)}{2}+1}\\ &\leq 2\log_2(n)\sqrt{n}\end{aligned}$$

□

Proposition 4.2.4. *For a sufficiently large, fixed length* n, $D(pal_n)\leq 5\sqrt{n}$

Proof. Let $m=\lceil\frac{\log_2(n)}{2}\rceil$. We consider the trivial tree automaton T_{pal_n}. We preserve the top m levels of the tree and modify the bottom $h-m$ levels. Note that the pivot element lies in the preserved top m levels.

Let U be the set of nodes at the lowest unmodified level of T_{pal_n} (i.e. the nodes at height m from the root). For each $u\in U$, let σ_u be the string used to reach u from the root and TS_{σ_u} be the corresponding trailing set. We know that $\tau_u^*\leq h-m, \forall u\in U$.

There are 2 possible scenarios:

1. $|TS_{\sigma_u}|=1$
2. $|TS_{\sigma_u}|\geq 2$

In the first scenario, instead of creating a chain per u (like in Proposition 4.2), we have an inverted bottom half (like in Figure 2.2). All the nodes, except for the last one, will be rejecting states. We transit u to an appropriate node accordingly.

For the second scenario, we capture the information in 2 different encodings. Before we proceed, we lay down some definitions.

- Let x, y be the 2^{nd} longest and longest possible palindromes (i.e. $x\subset y$)
- $|x|=q$, $|y|=r$, $q<r\leq\log_2(n)$
- x_w be the substring such that $x=x_wzx_w^R$ for some pivot $z\in\{\epsilon,0,1\}$
- y_w be the substring such that $y=y_wz'y_w^R$ for some pivot $z'\in\{\epsilon,0,1\}$.

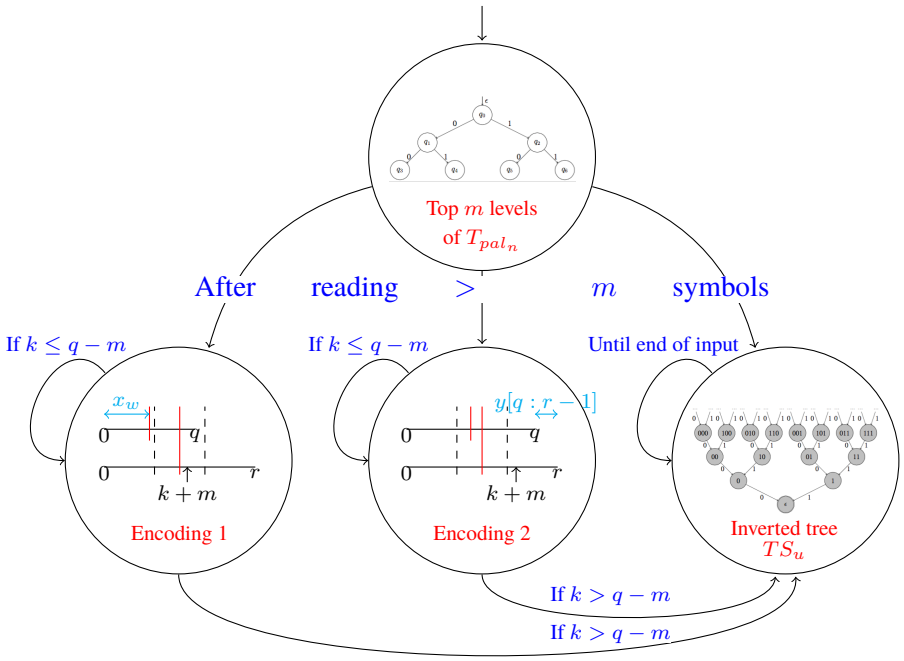


Figure 4.3: The 4 node classes and their transitions

Let $m + k$ be the index we are currently at ($0 < k \leq q - m$).

If $m < q \leq \frac{2r}{3}$ (i.e. $m < |x| \leq \frac{2|y|}{3}$)
we store $\langle x_w, b(q), b(r), k\rangle = \langle x[0 : \lceil \frac{q}{2} \rceil - 1], b(q), b(r), b(k)\rangle$ (Encoding 1).

Otherwise, $\frac{2r}{3} < q < r$ (i.e. $\frac{2|y|}{3} < |x| < |y|$),
we store $\langle y[q : r-1], b(q), b(r), b(k)\rangle$ (Encoding 2).

Encoding 1 uses $\lceil \frac{q}{2} \rceil + \log_2(q) + \log_2(r) + \log_2(k)$ bits.
Encoding 2 uses $r - q + \log_2(q) + \log_2(r) + \log_2(k)$ bits.

To decode, refer to Section B.2 and Subsection B.2.

We represent each generated tuple via the encodings with a node. If $k < q - m$, then $\langle \cdot, k\rangle$ transits to $\langle \cdot, k+1\rangle$ on the next expected bit[6]. If $k = q - m$, then $\langle \cdot, k\rangle$ transits to the inverted tree from scenario 1 on the next expected bit. Figure 4.3 illustrates this graphically.

In our construction, we have the following 4 classes of nodes:

[6]Suppose we have seen u bits, then the next *expected* bit is $\tau^*_u[u]$. Reading the other bit will transit to a forever rejecting state.

1. Top m levels
   Labelled by: $\langle 00, i\rangle$, for $0 \leq i \leq 2^m - 1$

$$\begin{aligned} \text{Total number of possible states} &: 2^m \\ &\leq 2 \cdot 2^{\frac{\log_2(n)}{2}} \\ &= 2\sqrt{n} \end{aligned}$$

2. Inverted tree (if the trailing set only has 1 element)
   Labelled by: $\langle 01, j\rangle$, for $0 \leq j \leq 2^{h-m-1} - 1$

$$\begin{aligned} \text{Total number of possible states} &: 2^{h-m} \\ &\leq 2^{\frac{\log_2(n)}{2}} \\ &= \sqrt{n} \end{aligned}$$

3. Encoding 1 (if the trailing set only ≥ 2 elements and $m < q \leq \frac{2r}{3}$)
   Labelled by: $\langle 10, x[0 : \lceil \frac{q}{2} \rceil - 1], b(q), b(r), b(k)\rangle$

$$\begin{aligned} \text{Total number of possible states} &: 2^{\lceil \frac{q}{2} \rceil + \log_2(q) + \log_2(r) + \log_2(k)} \\ &\leq 2^{\lceil \frac{2\log_2(n)}{6} \rceil + \log_2(\log_2(n)) + \log_2(\log_2(n)) + \log_2(\log_2(n))} \\ &\leq 2^{\frac{\log_2(n)}{3} + 3\log_2\log_2(n) + 1} \\ &\leq 2^{\frac{\log_2(n)}{2}} \\ &= \sqrt{n} \end{aligned}$$

4. Encoding 2 (if the trailing set only ≥ 2 elements and $\frac{2r}{3} < q < r$)
   Labelled by: $\langle 11, y[q : r - 1], b(q), b(r), b(k)\rangle$

$$\begin{aligned} \text{Total number of possible states} &: 2^{\lceil \frac{r-q}{2} \rceil + \log_2(q) + \log_2(r) + \log_2(k)} \\ &\leq 2^{\lceil \frac{\log_2(n)}{3} \rceil + \log_2(\log_2(n)) + \log_2(\log_2(n)) + \log_2(\log_2(n))} \\ &\leq 2^{\frac{\log_2(n)}{3} + 3\log_2\log_2(n) + 1} \\ &\leq 2^{\frac{\log_2(n)}{2}} \\ &= \sqrt{n} \end{aligned}$$

Hence, in total, we need at most $5\sqrt{n}$ states. For the inequalities[7] in Encodings 1 and 2, $\frac{\log_2(n)}{2} \geq \frac{\log_2(n)}{3} + 3\log_2(\log_2(n)) + 1$ when $n \geq 1.0843 * 10^{40}$. □

Proposition 4.2.5. *For a sufficiently large, fixed length n, $N(pal_n) \leq 5\sqrt{n}$*

Proof. A DFA is a NFA. So, $N(pal_n) \leq D(pal_n) \leq 5\sqrt{n}$. □

[7] Numerically solved using Wolfram Alpha: `http://www.wolframalpha.com/input/?i=log_2%28n%29%2F2+%3E%3D+log_2%28n%29%2F3+%2B+3+log_2%28log_2%28n%29%29+%2B+1`

Lower bound complexities

Proposition 4.2.6. *For a fixed length* $n \geq 1$, $D(pal_n) \geq \frac{\sqrt{n}}{2}$

Proof. For an arbitrary string of length m, say $a_1a_2...a_m$, there is only one unique string σ of length m such that $a_1a_2...a_m\sigma = a_1a_2...a_ma_m...a_2a_1$ is a palindrome.

That is to say, the equivalence class for every derivation $a_1a_2...a_m$ is unique.

For pal_n, there are $\sum_{m=0}^{\lfloor \frac{\log_2(n)}{2} \rfloor} 2^m \geq 2^{\frac{\log_2(n)}{2}-1} = \frac{\sqrt{n}}{2}$ derivations, so we need at least $\frac{\sqrt{n}}{2}$ states. □

Proposition 4.2.7. *For a fixed length* $n \geq 1$, $N(pal_n) \geq \frac{\sqrt{n}}{2}$

Proof. After seeing the symbols $a_1a_2...a_m$, for $1 \leq m \leq \lfloor \frac{\log_2(n)}{2} \rfloor$, the NFA must be in some state $S_{a_1...a_m}$ such that:

(a) The automaton can reach an accepting state by reading $a_ma_{m-1}...a_1$

(b) The automaton cannot reach an accepting state by reading any other length m sequence of symbols

By (a) and (b), $S_{a_1...a_m} = S_{b_1...b_m} \iff a_1...a_m = b_1...b_m$.

For pal_n, there are $\sum_{m=0}^{\lfloor \frac{\log_2(n)}{2} \rfloor} 2^m \geq 2^{\frac{\log_2(n)}{2}-1} = \frac{\sqrt{n}}{2}$ such states. □

Conclusion

Theorem 4.2.8. *For a sufficiently large, fixed length* n,

- $\mathcal{O}(\sqrt{n}) \leq D(pal_n) \leq \mathcal{O}(\sqrt{n})$
- $\mathcal{O}(\sqrt{n}) \leq N(pal_n) \leq \mathcal{O}(\sqrt{n})$

Proof. Refer to Theorems in the subsections above. □

It is known that DFA and NFA are equivalent but NFA may often offer a more compact representation (up to an exponential number of states saved) due to non-deterministic transitions. This is evident in the general bound results that we obtained in the previous sections.

Here, the language of palindromes provide us a nice example which shows that its complexity are equivalent in both DFA and NFA.

4.3 Power length sequence

A regular language is a context-free grammar but the converse may not be true. In the previous section, we saw how the palindrome CFG matches the lower bounds of automatic complexity (up to a constant factor). Because the language of palindrome is a CFG, it only has a constant complexity with respect to C_k. In this section, we study a language that is not constant in the complexity of CFG that exhibits an exponential drop in complexity from automata to CFG. For simplicity, we study C_2 but we believe similar results apply for general C_k.

Consider the language $A_{2^m} = \{w \in \{0,1\}^* : |w| = 2^m, m \in \mathbb{N}\}$, represented by the string pow:

$$pow[i] = 1 \iff |ll(i)| \text{ is a power of 2}$$

Definition 4.3.1. **(m^*)**
For a fixed prefix n, we define $m^* = \max\{m : 2^{2^m} \leq n\}$.
That is, $2^{m^*} = \lfloor \log_2(n) \rfloor$ is the length of longest accepted $ll(i)$ indices in pow_n.

Upper bound complexities

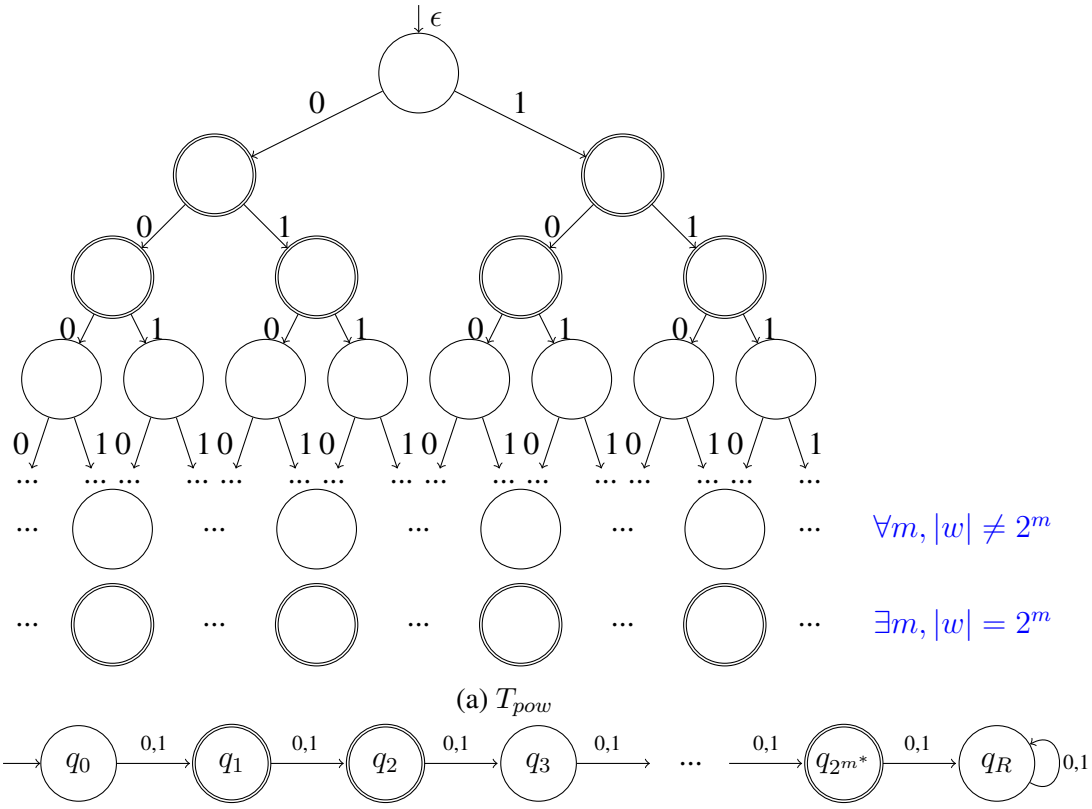
Proposition 4.3.2. *For a fixed length $n \geq 1$, $D(pow_n) \leq 2^{m^*} + 2$*

Proof. Taking a look at T_{pow}, we see that nodes at every level of the trivial tree are equivalent: They are either rejecting or all accepting, and since this pattern holds for all levels, we can cluster all nodes on the same level to just one node. We can prove this fact by induction on the height of T_{pow} but we do not need to. This is because to show upper bound, we just need to give a working DFA.

Refer to Figure 5.5. Consider the following DFA defined by $(Q, \Sigma, \delta, q_0, F)$:

$$\begin{aligned}
\text{where} \quad Q &= \{q_0, q_1, ..., q_{2^{m^*}}, q_R\} \\
\Sigma &= \{0,1\} \\
\delta &= \{(q_i, 0, q_{i+1}) : i \in \{0,1,...,2^{m^*}-1\}\} \cup \\
&\quad \{(q_i, 1, q_{i+1}) : i \in \{0,1,...,2^{m^*}-1\}\} \cup \\
&\quad \{(q_{2^{m^*}}, 0, q_R)\} \cup \{(q_{2^{m^*}}, 1, q_R)\} \cup \\
&\quad \{(q_R, 0, q_R)\} \cup \{(q_R, 1, q_R)\} \\
F &= \{q_{2^m} : m = \{0,1,2,...,2^{m^*}\}\}
\end{aligned}$$

The idea is that q_i will represent level i in T_{pow}, where i is the number of bits read by the DFA so far. The additional q_R is a forever rejecting state when we read beyond 2^{m^*} symbols. Hence, $D(pow_n) \leq 2^{m^*} + 2$. □



(a) T_{pow}

(b) Compressed DFA

Figure 4.4: Trivial tree automaton and DFA for pow

Proposition 4.3.3. *For a fixed length* $n \geq 1$, $N(pow_n) \leq 2^{m^*} + 2$

Proof. A DFA is a NFA, hence the result from the previous proposition applies. That is, $N(pow_n) \leq D(pow_n) \leq 2^{m^*} + 2$ □

Proposition 4.3.4. *For a fixed length* $n \geq 1$, $C_2(pow_n) \leq m^* + 1$

Proof. The idea behind the proof is divide-and-conquer, a common technique in the field of Computer Science and problem solving. Consider the following CFG defined by (NT, T, R, S):

$$\begin{array}{rrcl} \text{where} & NT & = & \{S\} \cup \{A_m : m \in \{0, 1, 2, ..., m^* - 1\}\} \\ & T & = & \{0, 1\} \\ & R & = & \{S \to A_0\} \cup \{S \to A_m A_m : m \in \{0, 1, 2, ..., m^* - 1\}\} \cup \\ & & & \{A_m \to A_{m-1} A_{m-1} : m \in \{1, 2, ..., m^* - 1\}\} \cup \\ & & & \{A_0 \to 0, A_0 \to 1\} \end{array}$$

S (only) transits to A_0 or 2 copies of A_m, and $A_m \Rightarrow^* \{0,1\}^{2^m}$ (only), for $m \in \{0, 1, 2, ..., m^* - 1\}$. Hence, the CFG only produces $\{0,1\}^{2^m}$, for $m \in \{0, 1, 2, ..., m^*\}$, which are the $ll(i)$ indices we want to capture in pow_n. □

Lower bound complexities

Proposition 4.3.5. *For a fixed length $n \geq 1$, $N(pow_n) \geq \max\{1, 2^{m^*-1}\}$*

Proof. When $m^* = 0$, we have $N(pow_n) \geq 1$.

When $m^* \geq 1$, consider an arbitrary automata that recognises pow_n. Between the set of nodes that accept indices of length 2^{m^*-1} and set of nodes that accept indices of length 2^{m^*}, there must be at least 2^{m^*-1} distinct non-accepting states. So, $N(pow_n) \geq 2^{m^*-1}$.

Hence, $N(pow_n) \geq \max\{1, 2^{m^*-1}\}$. □

Proposition 4.3.6. *For a fixed length $n \geq 1$, $D(pow_n) \geq \max\{1, 2^{m^*-1}\}$*

Proof. A DFA is a NFA, so the result from the previous proposition applies. That is, $D(pow_n) \geq N(pow_n) \geq \max\{1, 2^{m^*-1}\}$ □

Proposition 4.3.7. *For a fixed length $n \geq 1$, $C_2(pow_n) \geq m^*$*

Proof. Consider an arbitrary index $w \in pow_n$ such that $|w| = 2^{m^*}$.

Case 1: In the derivation of $S \Rightarrow^* w$, there are no repeated non-terminals. Then, we have at least $|w| = 2^{m^*}$ non-terminals.

Case 2: In the derivation of $S \Rightarrow^* w$, there are repeated non-terminals. Refer to Figure 4.5. Let A be the lowest (farthest from S) non-terminal that is repeated in the derivation tree of w. By choice of A, both b and d are derived from unique non-terminals. Pick the smallest $k \in \mathbb{N}$ such that $0 \leq \max\{|b|, |d|\} \leq 2^k$.

We have:

$$\begin{array}{lll} S & \Rightarrow^* & abcde = w \\ S & \Rightarrow^* & aAe \\ A & \Rightarrow^* & bAd \\ A & \Rightarrow^* & bcd \\ A & \Rightarrow^* & c \end{array}$$

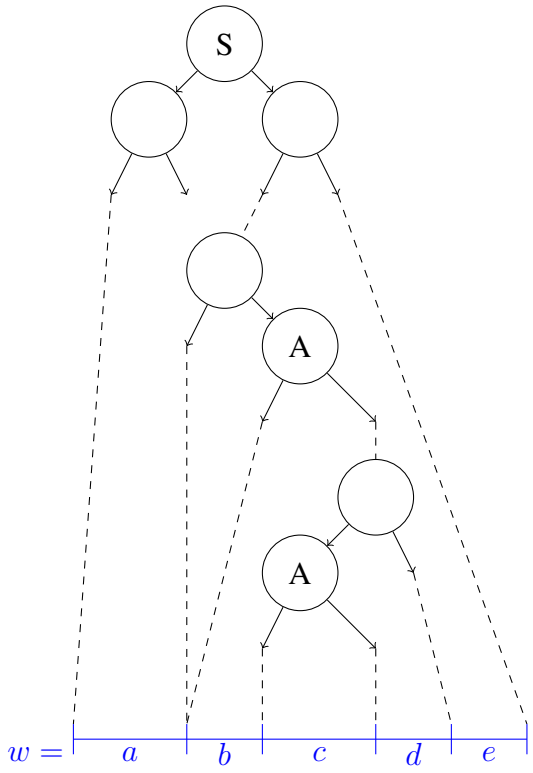


Figure 4.5: Derivation of index w, where $w \in pow_n$ and $|w| = 2^{m^*}$

Then:

$$\begin{aligned} S &\Rightarrow^* aAe \\ &\Rightarrow^* abAde \\ &\Rightarrow^* abbcdde \end{aligned}$$

Since $|abbcdde| > |abcde| = |w|$, $abbcdde \notin pow_n$ and $|abbcdde| > 2^{m^*}$, then $2^{m^*+1} \leq |abbcdde| = |abcde| + |b| + |d| \leq 2^{m^*} + 2^k + 2^k$. Manipulating this equation gives us $m^* - 1 \leq k$. Adding S, we need at least m^* non-terminals.

Conclusion

The 2 cases cover all possibilities.

So, we need at least $\min\{2^{m^*}, m^*\} = m^*$ non-terminals. □

Conclusion

Theorem 4.3.8. *For a fixed length $n \geq 1$,*

- $\mathcal{O}(\log_2(n)) \leq D(pow_n) \leq \mathcal{O}(\log_2(n))$
- $\mathcal{O}(\log_2(n)) \leq N(pow_n) \leq \mathcal{O}(\log_2(n))$
- $\mathcal{O}(\log_2(\log_2(n))) \leq C_2(pow_n) \leq \mathcal{O}(\log_2(\log_2(n)))$

Proof. From the above propositions, for a fixed length $n \geq 1$, we have:

- $\max\{1, 2^{m^*-1}\} \leq D(pow_n) \leq 2^{m^*} + 2$
- $\max\{1, 2^{m^*-1}\} \leq N(pow_n) \leq 2^{m^*} + 2$
- $m^* \leq C_2(pow_n) \leq m^*$

By definition, $m^* = \max\{m : 2^{2^m} \leq n\}$.
So, $2^{m^*} \leq \log_2(n)$ and $m^* \leq \log_2(\log_2(n))$. Therefore, the results follow. □

4.4 Martin-Löf Random strings

What makes a string random? In standard statistics and probability theory, the phrase *a random choice* is equivalent to selecting an item out of n items, each with equal chance of $\frac{1}{n}$. So, given a random choice in the set $\{0, 1\}$, both bits are equally likely. Continuing a random selection for n bits, we get a *random*[8] sequence of n bits. From this point of view, the strings 1111111111..., 1010101010..., 1101010010... are all equally likely to occur.

However this is hardly satisfying as it seems that the next bit in a sequence of all 1's is more predictable than a sequence where the next bit seems to be independent of whatever bits that came before it. *How then do we formalise what is a random string and what tests should they pass in order to be considered random?*

[8] in the probabilistic sense

To formally define the notion of a test for randomness, Martin-Löf looked to the field of theory of computation. Since then, it has been shown that there are 3 equivalent characterisations of algorithmic random strings:

- Constructive null covers (in terms of randomness tests)
  This is Martin-Löf's original definition. Interested readers can read the seminal paper [**Martin-Lof1966**].
- Kolmogorov complexity (in terms of compression)
  In this form, a random string cannot be compressed well since there is no intrinsic structure to it.
- Constructive martingales (in terms of gambling and prediction)
  This characterisation tells us a sequence of bits is random if we cannot profit (in the long run) by gambling on betting whether the next bit is 0 or 1, given whatever prefix of that random sequence.

The above three characterisations meet Martin-Löf randomness, provided that one chooses the right notion of constructivity or randomness. We will now formalise the notion of Kolmogorov complexity using prefix-free machines and then prove that Martin-Löf random strings attain the lower bound complexities we have shown for DFA, NFA and CFG up to a constant factor.

Definition 4.4.1. (Prefix-free machine)
A partial mapping V from $\{0,1\}^*$ to $\{0,1\}^*$ is a prefix-free machine if and only if the domain is prefix-free and V is partial-recursive, that is, computed by some algorithm or, more formally, Turing machine.

Here, the domain is prefix-free if and only if for all $x \in dom(V)$ and all $y \in \{0,1\}^+$, $xy \notin dom(V)$.

Definition 4.4.2. (Martin-Löf Random (MLR))
x is Martin-Löf Random (MLR) if and only if
$\neg\exists V\ \exists^\infty n\ \exists p \in \{0,1\}^*\ [V \text{ is prefix-free} \wedge |p| < n \wedge V(p) \downarrow = x_n]$

In other words, if x is MLR and V a prefix-free machine, then for almost all n, the prefix x_n of x of length n has only codes p in V which are longer than n.

Proposition 4.4.3. *For any string x_n, $n \geq 2^3 = 8$, we can encode it in a prefix-free way in $3\log_2(n) + \log_2(f(k))$ binary bits, where k =complexity(x_n) in the chosen representation (DFA/NFA/CFG) and $f(k)$ is the number of possible machines with parameter k.*

Proof. Given a representation (DFA/NFA/CFG) and k, we can obtain $f(k)$ by counting argument. We can then fix an ordering over all possible machines. Using the chosen machine, we can produce x_n if we know the prefix length n. Since $k \leq n$, we can represent k in $\log_2(n)$ bits. To make the encoding prefix-free, we prepend the encoding with $1^{log_2(n)-4}0$. For all sufficiently large n, this prefix exists; for small n, the coding is not considered.

x_n	$\|n\| - 2^3$	n	k	Index out of $f(k)$ machines
# bits	$\log_2(n) - 3$	$\log_2(n) + 1$	$\log_2(n) + 1$	$\log_2(f(k))$ + 1

Hence, in total, we need $3\log_2(n) + \log_2(f(k))$ bits. □

Remark: The encoding length can be brought down to $2\log_2(n) + \log_2(f(k))$ bits since we can write a program for each representation to compute k given the x_n. For example, our program in Chapter 5 computes k for x_n in the DFA model.

Deterministic Finite Automata (DFA)

With k states, we have $2^{2k\log_2(k)+k}$ possible automata. By Proposition 4.4.3, we can encode a string x_n with:

$$\begin{aligned} & 3\log_2(n) + \log_2(2^{2k\log_2(k)+k}) \\ = \; & 3\log_2(n) + 2k\log_2(k) + k \qquad \text{bits.} \end{aligned}$$

If x is MLR, then $\forall^\infty n[3\log_2(n) + 2k\log_2(k) + k \geq n]$. Then,

$$\begin{array}{lll} n \leq 3\log_2(n) + 2klog_2(k) + k & & \\ n \leq 3\log_2(n) + k(2\log_2(n) + 1) & \text{, because } \log_2(k) \leq \log_2(n) & \\ k \geq \frac{n-3\log_2(n)}{1+2\log_2(n)} & \text{(Rearranging)} & \\ k \geq \frac{n}{3\log_2(n)} & \text{, for } n \geq 64.9917^9 & \end{array}$$

Non-deterministic Finite Automata (NFA)

With k states, we have at most 2^{k+2k^2} possible automata. By Proposition 4.4.3, we can encode a string x_n with:

$$\begin{array}{rl} & 3\log_2(n) + \log_2(2^{k+2k^2}) \\ = & 3\log_2(n) + k + 2k^2 \qquad \text{bits.} \end{array}$$

If x is MLR, then $\forall^\infty n[3\log_2(n) + k + 2k^2 \geq n]$. Then,

$$\begin{array}{rcll} n & \leq & 3\log_2(n) + k + 2k^2 & \\ 0 & \leq & 2k^2 + k + (3\log_2(n) - n) & \text{(Rearranging)} \\ k & \geq & \frac{-1+\sqrt{1^2-4(2)(3\log_2(n)-n)}}{2(2)} & \text{(Solve for roots. Negative root N/A)} \\ k & \geq & -\frac{1}{4} + \sqrt{\frac{1}{16} - \frac{24}{16}\log_2(n) + \frac{n}{2}} & \\ k & \geq & -\frac{1}{4} + \sqrt{\frac{n}{2.5}} & \text{, for } n \geq 98.7636^{10} \\ k & \geq & \sqrt{\frac{n}{3}} & \text{, for } n \geq 20.5823^{11} \end{array}$$

Context Free Grammar (CFG)

With c non-terminals, we have at most $2^{2c^{k+1}}$ possible CFG. By Proposition 4.4.3, we can encode a string x_n with:

$$\begin{array}{rl} & 3\log_2(n) + \log_2(2^{2c^{k+1}}) \\ = & 3\log_2(n) + 2c^{k+1} \qquad \text{bits.} \end{array}$$

If x is MLR, then $\forall^\infty n[3\log_2(n) + 2c^{k+1} \geq n]$. Then,

$$\begin{array}{rcll} n & \leq & 3\log_2(n) + 2c^{k+1} & \\ c & \geq & \sqrt[k+1]{\frac{n-3\log(n)}{2}} & \text{(Rearranging)} \\ c & \geq & \sqrt[k+1]{\frac{n}{3}} & \text{, for } n \geq 51.0695^{12} \end{array}$$

Conclusion

Theorem 4.4.4. *Martin-Löf Random strings x meet the lower bound complexities of DFA/NFA/CFG, up to a constant factor, for almost all prefixes x_n of x.*

Proof. Refer to the arguments in the previous subsections. □

Chapter 5

Computation of explicit bounds in the DFA model

During the course of our project, we studied various binary strings. At one point in time, we were trying to find a natural language that matches the existential lower bounds we have proved in Chapter 2. The language of palindromes attain the lower bound of NFA up to a constant factor. The natural follow-up question was: *Could we find a language that attain the lower bound for DFA?*

5.1 Methodology

We wrote a simple Python2 script and executed it on the school's compute cluster[1] for about a day. Note that due to the inherent complexity of the strings, the program is able to analyse longer lengths of certain strings over others, within the same amount of time.

For each binary string of interest, we build a generator function and passed it through our program. After extracting the data, we compared how they change with respect to some parameters (refer to Subsection 5.1). In all cases, we consider n prefixes of the string for $n \geq 100$ because we are more interested in the asymptotic behaviour as n increases.

The program

From the trivial tree automaton of a sequence, we check if any of the nodes are equivalent and can therefore by combined. We first build a subroutine

[1] 16 *Intel(R) Xeon(R) CPU E5520 @ 2.27GHz* CPUs with 24GB RAM, running Python 2.6.6

MATCH(i, j, lst) that helps us decide if nodes i and j are equivalent. Notice that MATCH makes use of the transition pattern described in Figure 1.2.

Algorithm 1 MATCH(i, j, lst)

if $j \geq |lst|$ **then**
  return True
else
  $current\_match \leftarrow (lst[i] == lst[j])$
  $left\_match \leftarrow$ MATCH$(2 * i + 1, 2 * j + 1, lst)$
  $right\_match \leftarrow$ MATCH$(2 * i + 2, 2 * j + 2, lst)$
  return $(current\_match$ and $left\_match$ and $right\_match)$
end if

Figure 5.1: The recursive MATCH subroutine

Algorithm 2 FINDPATTERN(seq)

$eq \leftarrow \emptyset$
$seen \leftarrow \emptyset$
for $i \in \{0, 1, 2, ..., |seq| - 1\}$ **do**
  if $i \notin seen$ **then**
    $matches \leftarrow \emptyset$
    for $j \in \{i + 1, i + 2, ..., |seq| - 1\}$ **do**
      if MATCH(i, j, seq) **then**
        $matches \leftarrow matches \cup \{j\}$
        $seen \leftarrow seen \cup \{j\}$
      end if
    end for
    $eq \leftarrow output \cup \{\{i, matches\}\}$
  end if
end for
return eq

Figure 5.2: The FINDPATTERN function

Building upon MATCH, we create a function FINDPATTERN(seq) which returns us $eq = \{\{i, matches_i\}\} = \{\{\text{Node } i, \{\text{Nodes equivalent to node } i\}\}\}$, the set of equivalence classes. Then, $|eq|$ will be the DFA complexity of the fixed prefix length sequence. To ensure correct counting of unique nodes needed to produce the sequence, an index may appear in more than one $matches_i$ but such indices will never appear as the representative node i.

Parameters

The table below shows the parameters that are of interest to us.

Parameter	Description	Name in graph
$\|x\|$	Prefix length of interest	n
$\|eq\|$	Number of unique nodes needed[2]	y
$\frac{y}{\log_2(\|x\|)}$	Ratio between y and the $\log_2(\|x\|)$ bound	$logRatio$
$\frac{y}{\frac{\|x\|}{\log_2(\|x\|)}}$	Ratio between y and the $\frac{\|x\|}{\log_2(\|x\|)}$ bound	$DFARatio$
$\frac{y}{\sqrt{\|x\|}}$	Ratio between y and the $\sqrt{\|x\|}$ bound	$NFARatio$

Table 5.1: Table of parameters

5.2 Specific examples

In this section we look at 4 binary strings:

- Morse-Thue sequence (MT)
- Palindrome sequence (pal)
- Power length sequence (pow)
- Champernowne sequence ($cham$)

For each string, we use a subscripted n to indicate the length n prefix of the string. For example, $MT_3 = MT[0:2] = 011$.

[2]Computed using FINDPATTERN

Morse-Thue sequence (MT)

There are 2 ways to generate MT as mentioned before.

- For $n \in \mathbb{N}$,

$$\begin{aligned} MT[0] &= 0 \\ MT[2^n : 2^{n+1} - 1] &= \overline{MT[0 : 2^n - 1]} \end{aligned}$$

- For $n \in \mathbb{N}$,

$$\begin{aligned} MT[0] &= 0 \\ MT[2n] &= MT[n] \\ MT[2n+1] &= 1 - MT[n] = \overline{MT[n]} \end{aligned}$$

For MT, we have proved the following theoretical result: (Corollary 4.1.2) $\forall n \in \mathbb{N}, n \geq 4, D(MT_n) \leq 4$. Our program gives us the chart in Figure 5.3. It is not surprising to see that we get constant line of $y = 4$.

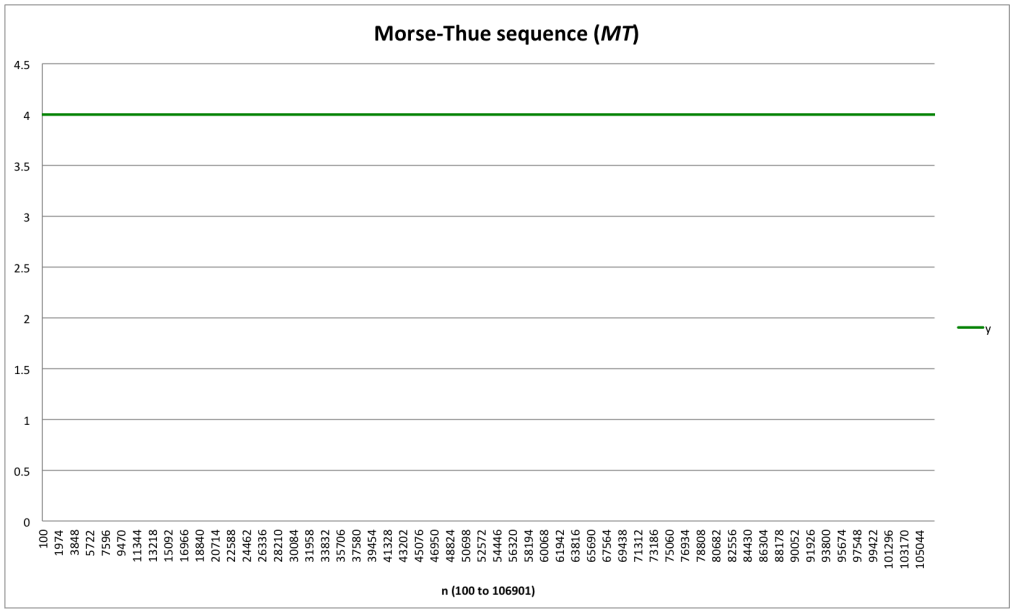


Figure 5.3: Output of program on the Morse-Thue sequence (MT)

Palindrome sequence (pal)

Recall the definition of pal: $pal[i] = 1 \iff ll(i)$ is a palindrome (i.e. $ll(i) \in A_P$). For pal, we have proved the following theoretical result: (Theorem 4.2.8) $\mathcal{O}(\sqrt{n}) \leq D(pal_n) \leq \mathcal{O}(\sqrt{n})$. To be more precise, $\frac{\sqrt{n}}{2} \leq D(pal_n) \leq 5\sqrt{n}$.

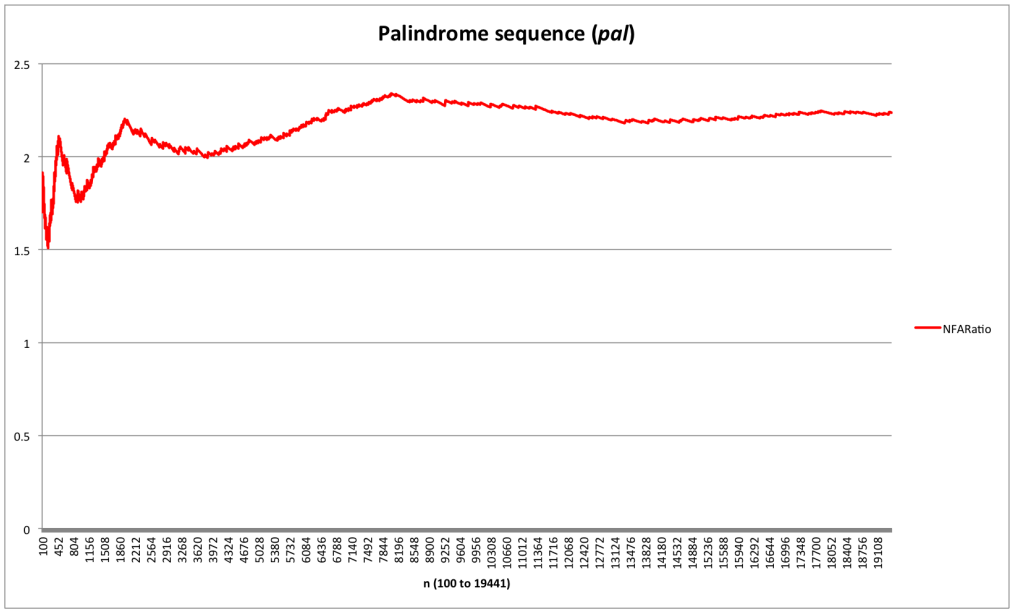


Figure 5.4: Output of program on the Palindrome sequence (pal)

From Figure 5.4, we see that $NFARatio \in [1.503557036, 2.341004746]$ for $100 \leq n \leq 19441$ and $NFARatio \in [2.004541431, 2.341004746]$ for $4000 \leq n \leq 19441$. This empirically agrees with our theorem, albeit our upper bound was proven rather generously.

Power length sequence (pow)

Recall the definition of pow: $pow[i] = 1 \iff |ll(i)|$ is a power of 2. For pow, we have proved the following theoretical result: (Theorem 4.4.4) $\mathcal{O}(\log_2(n)) \leq D(pow_n) \leq \mathcal{O}(\log_2(n))$. To be more precise, $\max\{1, 2^{m^*-1}\} \leq D(pow_n) \leq 2^{m^*} + 2$.

Our program gives us the chart in Figure 5.5. Notice that $y \in [0.5 \cdot 2^{m^*}, 2^{m^*} + 2] = [0.5 \cdot \lfloor \log_2(n) \rfloor, \lfloor \log_2(n) \rfloor + 2]$ for $100 \leq n \leq 76432$. This corresponds empirically with what we have proven earlier.

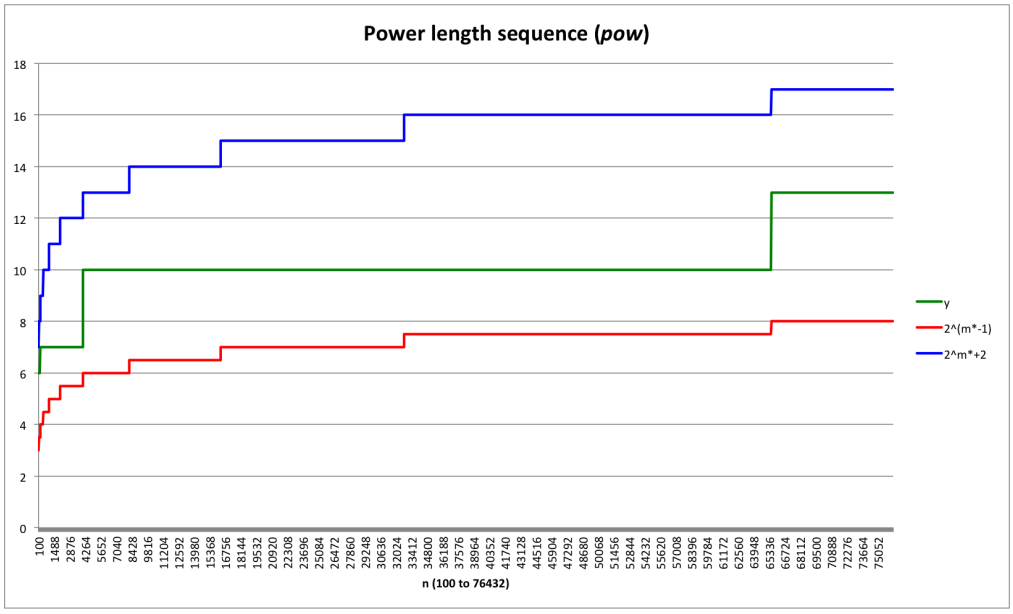


Figure 5.5: Output of program on the Power length sequence (pow)

Champernowne sequence ($cham$)

Here, we study the famous Champernowne sequence, which we denote by $cham$. The Champernowne sequence is formed by concatenating representations of positive natural numbers, one by one. In base 10, we have: $0.12345678910111213...$ The sequence has has many fascinating properties related to normality and randomness.[3] In our report, we study the Champernowne sequence in the $\mathbb{L}$ representation. That is to say, we concatenate consecutive elements of $\mathbb{L}$ and get $cham = 0100011011000001010011....$

Figure 5.6 shows us that $DFARatio \in [1.436432038, 1.702981861]$ for $100 \leq n \leq 12364$. Based on this empirical evidence, we conjecture that $cham$ is one of the languages that meets the existential lower bound of DFAs that we have proved earlier. Unfortunately, we were unable to provide a proof to show that $\mathcal{O}(\frac{|x|}{\log_2(|x|)}) \leq cham \leq \mathcal{O}(\frac{|x|}{\log_2(|x|)})$.

[3]We point readers to `https://en.wikipedia.org/wiki/Champernowne_constant` and `http://mathworld.wolfram.com/ChampernowneConstant.html` to learn more about the properties of the Champernowne sequence.

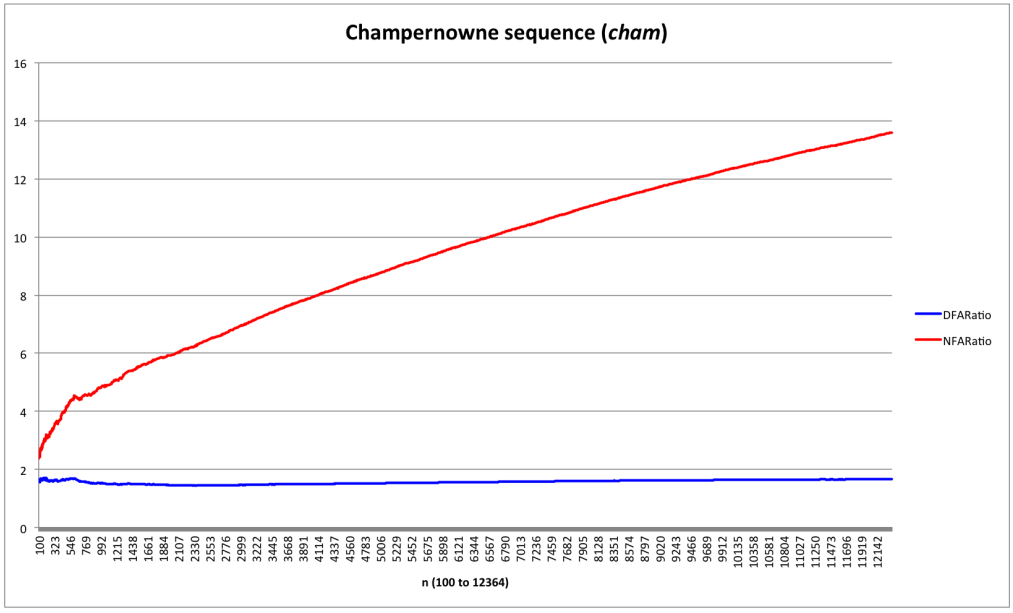


Figure 5.6: Output of program on the Champernowne sequence ($cham$)

5.3 Conclusions

As seen in the previous section, our program is able to provide us an empirical but numerically tighter bound than the generous bounds that we have proven in earlier chapters. We list down the numerical bounds below:

- For $100 \leq n \leq 106901$,
  $4 \leq D(MT_n) \leq 4$
- For $100 \leq n \leq 19441$,
  $1.50 \cdot \sqrt{n} \leq D(pal_n) \leq 2.35 \cdot \sqrt{n}$
  For $4000 \leq n \leq 19441$,
  $2.00 \cdot \sqrt{n} \leq D(pal_n) \leq 2.35 \cdot \sqrt{n}$
- For $100 \leq n \leq 76432$,
  $0.5 \cdot \lfloor \log_2(n) \rfloor \leq D(pow_n) \leq \lfloor \log_2(n) \rfloor + 2$
- For $100 \leq n \leq 12364$,
  $1.436 \cdot \frac{|x|}{\log_2(|x|)} \leq D(cham_n) \leq 1.703 \cdot \frac{|x|}{\log_2(|x|)}$

Appendix A

Appendix A: Side Proofs

Theorem A.0.1. *Kolmogorov complexity is not computable.*

Proof. Suppose, for a contradiction, that Kolmogorov complexity is computable.

Let $P(x)$ be the program that computes $K(x)$ for any given string x.

Consider the following program $Q(n)$ that returns the first binary string that has complexity greater than the input n:

Algorithm 3 $Q(n)$

```
for x ∈ {0,1}* do
  if P(x) > n then
    return x
  end if
end for
```

The programs P and Q is finite, so it $K(P) = j$ and $K(Q) = k$ for some $j, k \in \mathbb{N}$. Let $n = j + k + 2\log_2(j + k)$. Consider the output string $x_n = Q(n)$. By the program definition of $Q(n)$, $K(x_n) > n$.

But we could have described x_n by utilising the programs P, Q and the number n. That is to say, $K(x_n) \leq K(P) + K(Q) + K(n) = j + k + \log_2(n) \leq n$.

Contradiction! □

Appendix B

Appendix B: Repeated subwords

In this chapter, we explore languages that are made up of repeated subwords.

For instance, the language $A_{0^k} = \{w^k : w \in \{0,1\}^*\}$ consists of binary strings that are made up with k repeated smaller substrings. If $k = 3$, $\epsilon, 111, 101010 \in A_{000}$ and $11, 101, 1111 \notin A_{000}$. We may also choose to reverse or bit flip some of the substrings.

B.1 Definitions

In this chapter, we index languages A by a binary string $i \in \{0,1,2,3\}^+$ to indicate the sequence of substrings in the language. We will also often encounter a phenomenon we call *prefix nesting*.

Definition B.1.1. (Representation of index $i \in \{0,1,2,3\}^+$)

- The length of i indicates the number of substrings in A
- 0 represents the original subword (w)
- 1 represents the reverse of the subword (w^R)
- 2 represents the bit flipped version of the subword ($\overline{w}$)
- 3 represents the reversed, bit flipped version of the subword ($\overline{w^R} = \overline{w}^R$)

Definition B.1.2. (Type function $t(w, x)$)

$$t(w,x) = \begin{cases} w & \text{, if } x = 0 \\ w^R & \text{, if } x = 1 \\ \overline{w} & \text{, if } x = 2 \\ \overline{w^R} = \overline{w}^R & \text{, if } x = 3 \\ \uparrow & \text{, otherwise} \end{cases}.$$

Definition B.1.3. (Index language A_i)
Suppose $i = a_0^{b_0} a_1^{b_1} ... a_m^{b_m}$,

$$\begin{array}{rrcl} \text{where} & k & \in & \{0, 1, ..., m\}, \\ & a_k & \in & \{0, 1, 2, 3\}, \\ & b_k & \in & \mathbb{N} \setminus \{0\} \end{array}$$

Then, the language $A_i := \{(t(w, a_0))^{b_0} ... (t(w, a_m))^{b_m} : w \in \{0, 1\}^*\}$.

These indexed languages are not necessarily unique. For example, $A_{00} = A_{11}$ and $A_{010} = A_{101}$, etc.

Definition B.1.4. (Source string x_w)
Given $x \in A_i$ for some i, define the x_w as the substring w that makes up x.

Definition B.1.5. (Prefix nested)
We say x is prefix nested in y when:
(i) $x, y \in A$, (ii) $x \prec y$, and (iii) $|x| < |y|$.

In other words, "Both x and y are in the language, and x is a proper prefix of y".
Define $pn(A, x, y)$: In the language A, x is prefix nested in y.

B.2 The language $A_{a_0 a_1}$

Without loss of generality, we may assume that $a_0 = 0$. It may be useful to note that all strings in these languages have even lengths.

In this section, we investigate encoding techniques for prefix nested strings for languages $A_{a_0 a_1}$. We will first describe the encoding, then explain how to recover both x and y from the encoded string. The 2 methods presented below only make sense when $|x| \geq |y_w|$.

Encoding methods

Suppose $pn(A_{a_0 a_1}, x, y)$. Let $|x| = m, |y| = n$.

By exploiting the fact that x is a prefix of y, we can encode both x and y into a string shorter than $|y|$. Since $pn(A_{a_0a_1}, x, y)$, we have:
(i) $x[0 : m-1] = y[0 : m-1]$ and (ii) $y[m : n-1]$ is related to x_w.

Method 1: Store $\langle x_w, b(m), b(n)\rangle = \langle x[0 : \frac{m}{2} - 1], b(m), b(n)\rangle$

Method 2: Store $\langle y[m : n-1], b(m), b(n)\rangle$

Encoding using Method 1 uses $\frac{m}{2} + \log_2(m) + \log_2(n)$ bits.
Encoding using Method 2 uses $n - m + \log_2(m) + \log_2(n)$ bits.

For example, let A_{01}, $x = 110011$, $y = 1100110011$.
We have $m = 6$, $n = 10$, $x_w = 110$, $y_w = 11001$.

Method 1 will store: $\langle x_w, m, n\rangle = \langle 110, 110, 1010\rangle$.
Method 2 will store: $\langle y[m : n-1], m, n\rangle = \langle 0011, 110, 1010\rangle$.

Decoding methods

Method 1

Since we stored $\langle x_w, m, n\rangle = \langle x[0 : \frac{m}{2} - 1], m, n\rangle$, it suffices to recover $y[\frac{m}{2} : n]$.

- Since we have x_w, we can recover the entire x
- Since $|x| > y_w$, we can recover the entire y_w
- Since we have y_w, we can recover the entire y

Method 2

Since we stored $\langle y[m : n-1], m, n\rangle$, it suffices to recover $y[0 : m-1]$.

There are 4 different possible A_{0a_1} and the decoding is not the same for all of them. To be precise, A_{00} and A_{02} will be decoded in one way while A_{01} and A_{03} will be decoded in another. This is because, in our encodings, bit flips contain the same amount of information as not flipping the bits.

In the following, we will explain how to decode for A_{00} and A_{01}. Decoding for A_{02} and A_{03} will require additional bit flips after each *mirroring* step. For a string x, *mirroring* is the process of recovering bits of a source string x_w from another, across the midway point of x .

Decoding for A_{00}:

- Since we have $y[m : n-1]$, we can recover $y[m - \frac{n}{2} : \frac{n}{2}]$

- Claim: $m - \frac{n}{2} < \frac{m}{2} < \frac{n}{2}$
  Since $n > m$, then $\frac{m}{2} < \frac{n}{2}$.
  If $m - \frac{n}{2} \geq \frac{m}{2} \Rightarrow 2m - n \geq m \Rightarrow m - n \geq 0$.
  Contradiction, since $n > m$.

- By the claim, we have recovered at least 1 bit from the back of 1^{st} copy of x and at least 1 bit from 2^{nd} copy of x

- Using these bits, we can extract more bits as illustrated in Figure B.1 by repeated iterations of *mirroring*.

- Since each iteration of *mirroring* gains us more bits, we can repeat until we recover the entire x_w or y_w.

Decoding for A_{01}:

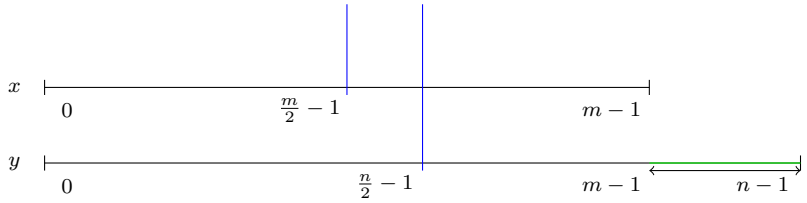
- Since we have $y[m : n-1]$, we can recover $y[0 : n-m-1] = x[0 : n-m-1]$.

- Since we have $x[0 : n-m-1]$, we can recover $x[2m-n : m-1] = y[2m-n : m-1]$.

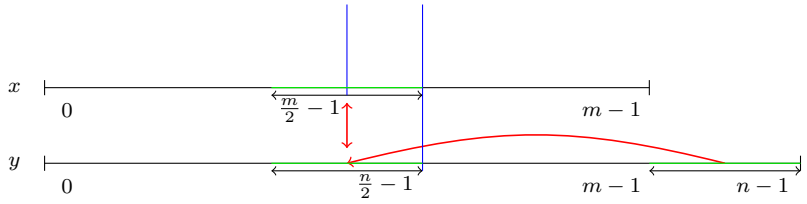
- Then we recurse the decoding on $x[n-m : 2m-n-1]$ and $y[n-m : m-1]$.

Comments

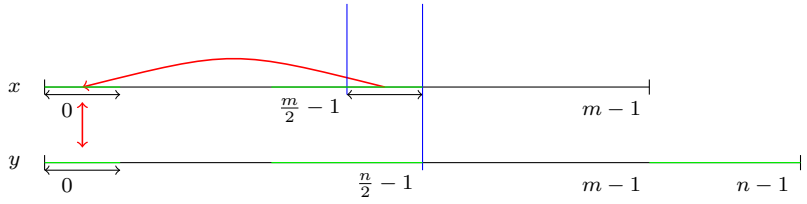
As of now, it is not clear how we can extend these methods to languages A_i for general i. However, it is useful to note that we can modify the encoding/decoding to work for the languages $\{wyw : w \in \{0,1\}^*, y \in \{0,1\}^k\}$ and $\{wyw^R : w \in \{0,1\}^*, y \in \{0,1\}^k\}$ for fixed $k \in \mathbb{N}$.



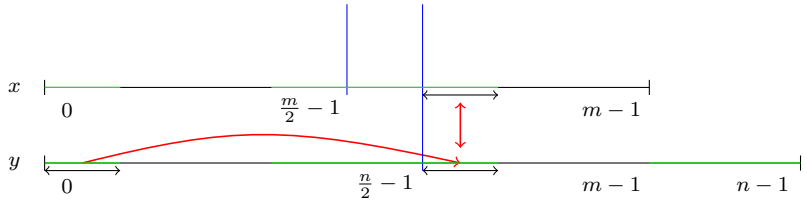
(a) We start off with $y[m : n-1]$



(b) Since $y \in A_{00}$, we recover the last bits of the first copy of y_w. Then, by prefix nesting property, we get some new bits of x_w.

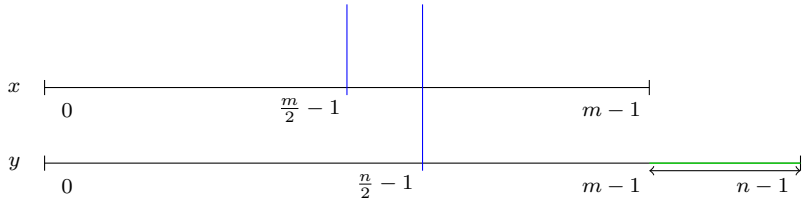


(c) Since $x \in A_{00}$, we recover the first bits of the first copy of x_w. Then, by prefix nesting property, we get the first bits of y_w.

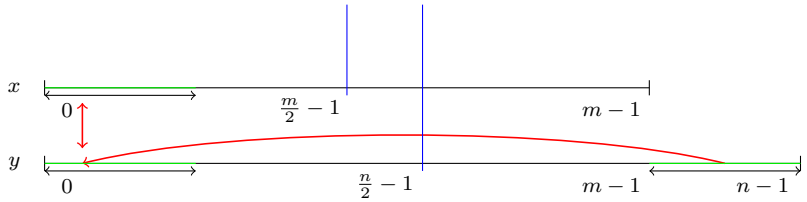


(d) Since $y \in A_{00}$, we recover the first bits of the second copy of y_w. Then, by prefix nesting property, we get new bits of x_w.

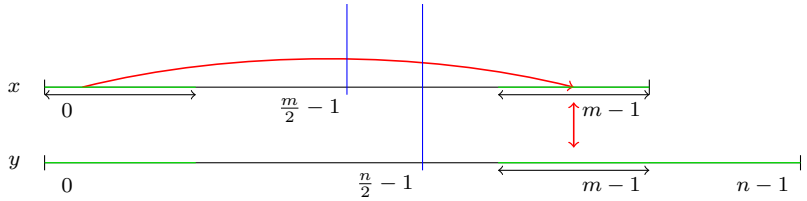
Figure B.1: Illustration for decoding method 2 for A_{00}.
Blue lines indicate repetition point. Green portions are known to us.
We can repeat this *mirroring* process until we recover the entire x_w or y_w.



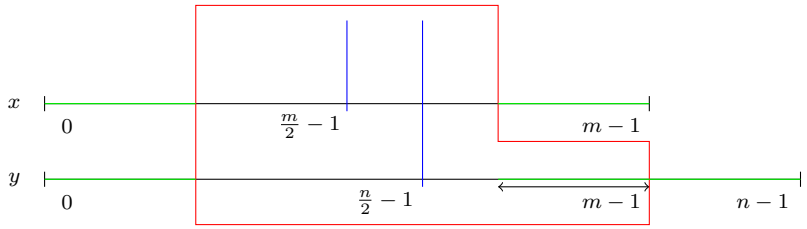
(a) We start off with $y[m : n-1]$



(b) Since $y \in A_{01}$, we recover the first bits of the first copy of y_w. Then, by prefix nesting property, we get the first bits of x_w.



(c) Since $x \in A_{01}$, we recover the last bits of the second copy of x_w. Then, by prefix nesting property, we get some new bits of y_w.



(d) Now we recurse the decoding on $x[n-m : 2m-n-1]$ and $y[n-m : m-1]$ (the substrings of x and y in the red box). Notice that they fulfil the nested property.

Figure B.2: Illustration for decoding method 2 for A_{01}.
Blue lines indicate repetition point. Green portions are known to us.
We can repeat this *mirroring* process until we recover the entire x_w or y_w.

References

[CP89] J. M. Champarnaud and J. E. Pin. "A Maxmin Problem on Finite Automata". In: *Discrete Applied Mathematics* 23 (1989), pp. 91–96.

[Cha94] G. J. Chaitin. "The Berry Paradox". In: *Complexity* (1994), p. 13. arXiv: 9406002 [chao-dyn]. URL: http://arxiv.org/abs/chao-dyn/9406002.

[VL97] Paul Vitányi and Ming Li. *An introduction to Kolmogorov complexity and its applications. (Second edition)*. Vol. 34. 10. 1997, p. 137. ISBN: 9780387339986. DOI: 10.1016/S0898-1221(97)90213-3.

[SW01] Jeffrey Shallit and Ming-wei Wang. "Automatic Complexity of Strings". In: *Journal of Automata, Languages and Combinatorics* 6 (2001), pp. 537–554.

[Hyd13] Kayleigh Hyde. "Nondeterministic finite state complexity". Project for Master of Arts in Mathematics, University of Hawai'i at Mānoa. 2013.

[LW14] Jan Van Leeuwen and Jiri Wiedermann. "Separating the Classes of Recursively Enumerable Languages Based on Machine Size". In: March (2014).

[HKH15] Kayleigh Hyde and Bjørn Kjos-Hanssen. "Nondeterministic Automatic Complexity of Overlap-Free and Almost Square-Free Words". In: *Electron. J. Combin.* 22.3 (2015), Paper 3.22, 18 pp.