TURING MACHINES AND AUTOMATIC FUNCTIONS AS MODELS OF COMPUTATION

SEAH CHENG'EN SAMUEL

(B.Sc.(Hons)), NUS

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF MATHEMATICS

NATIONAL UNIVERSITY OF SINGAPORE

2012

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submited for any degree in any university previously.

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

Seah Cheng'En Samuel

19 July 2012

[illegible]

Acknowledgements

I would like to say a very big thank you to my supervisor, Professor Frank Stephan, for his guidance and help during this research period and writing of the thesis. He has been a great mentor to me and his input has been very valuable. He has been very patient during discussion sessions and I have benefitted a lot from his expertise in this field. He has inspired me to push myself a little harder to "keep up" with his ever-increasing wealth of knowledge.

I would also like to thank my family for their support and encouragement. Special thanks also goes out to my graduate coursemates, Tze Siong, Charlotte, Yanjun and Yu Jie. Thank you all for helping me over the past few semesters of coursework and for your constructive comments with regards to any research ideas I had. Your knowledge and interest in mathematics have made learning together more exciting.

Contents

Summary

Prior work by Case, Jain, Le, Ong, Semukhin and Stephan showed that automatic functions can be computed in linear time without specifying the underlying model of computation. In the present report it is shown that a function is automatic if and only if a one-tape Turing machine can compute it in linear time where input and output have to start at the left end of the tape. It is also shown that the same equivalence holds when nondeterministic one-tape Turing machines are considered.

This result is extended to the study of the complexity class $AF[\log n]$ which is roughly the class of functions which can be computed by a machine with automatic functions as primitive operations in logarithmic time. It is shown that $AF[\log n]$ is a proper subclass of $DTIME[n \log n]$ but still significantly larger than the class $AF[1]$ of automatic functions.

We will investigate a little the class of $NAF[\log n]$ which is the class of functions which can be computed non-deterministically by a machine with automatic functions as primitive operations in logarithmic time. The models AF, DTIME and NAF are compared to find as much as possible, the relations between them.

List of Figures

Chapter 1

Mathematical Preliminaries

The thesis will investigate the notion of automatic functions and Turing machines and find the relation between them. One central result is that a function is automatic if and only if it is computed in linear time by a one-tape Turing machine with the input and output starting at the same position. This result was also generalised to non-deterministic one-tape Turing machines. In both cases, the restriction of the input and output starting at the same position is an important condition.

In the next part of this thesis, it is investigated which complexity class arises if a machine uses automatic functions as a primitive to manipulate its data and has a computational runtime of $O(\log n)$. We will call this model AUTOFUNC. This new model will be compared to a one-tape Turing machine and it will be shown that there are some computations computed by a one-tape Turing machine in $O(n \log n)$ that cannot by computed by our new model in $O(\log n)$ i.e. AF$[\log n] \subset$ DTIME $[n \log n]$.

In addition, we will investigate the model NAF which is the nondeterministic variation of AUTOFUNC and see how non-determinism affects computational ability.

In this chapter, we first recall some basic definitions and review the concepts of Turing machines, Crossing Sequences, Automata and the Pumping Lemma as they will be used frequently in the Chapters 2 and 3.

For more detailed definitions, references on these background topics include [1], [2] and [6].

As we will be talking about machines, in particular, a finite automaton or a Turing machine, we first define what it means for an input to be accepted or rejected by a machine.

Definition 1.0.1. *An input, x is said to be rejected by a machine M if for a given set F of final states of M, the computation of x by M halts at a state $q_R \in F$. We call q_R the rejecting state.*

Definition 1.0.2. *An input, x is said to be accepted by a machine M if the computation of x by M halts at a state $q_A \in F$ i.e. $M(x) = q_A$. We call q_A the accepting state. The set of all inputs accepted by M is called the language of M and denoted by $L(M)$.*

As we will be dealing with strings as the input, we define its corresponding structure.

Definition 1.0.3. *The free monoid on a set S is the monoid whose elements are all the finite sequences (or strings) of zero or more elements from S. It is usually denoted S^*. The identity element is the unique sequence of zero elements, often called the empty string and denoted by ϵ and the monoid operation is string concatenation.*

The idea of a regular language is often used when talking about automata and computations.

Definition 1.0.4. *A regular language is a language that can be generated by a regular expression or accepted by a finite automata.*

Lastly, we will use "$Big - O$" to talk about the time complexity of computations.

Definition 1.0.5. *Given $f, g : \mathbb{N} \to \mathbb{N}$, f is an asymptotic upper bound for g, denoted by $g \in O(f)$ if*

$$\exists C > 0, x_0 \in \mathbb{R} \textit{ such that } \forall x \geq x_0, g(x) \leq Cf(x)$$

1.1 Turing machines

The Turing machine can be seen as a thought experiment. Developed by Alan Turing in 1935, this is a theoretical model of computation that plays the role of an ideal computer. The main components of a Turing machine include the (semi-) infinite tape, a head and a set of instructions.

The tape represents the memory of the computer and is divided into cells. The head is the control and can be positioned on any cell of the tape. Each of these cells can contain only one symbol of a given alphabet, which is read by and/or written by the head. The set of instructions, also known as tuples, plays the role of determining the performed actions. We will use the symbols o and # to denote the start of the tape and the delimiter of the input tape respectively, i.e.# occurs to the right of the last symbol of the input.

What does a Turing machine do? The Church-Turing Thesis states that "All computable functions can be computed on Turing machines." Computable functions are functions that can be calculated using a mechanical calculation device given unlimited amounts of time and storage space. Essentially, a Turing machine can compute all functions that our laptop can do and perhaps even more! We will now study the Turing machines in more detail.

1.1.1 One-tape Deterministic Turing machine

A computation of a Turing machine M on an input x is a (possibly infinite) sequence of all the acitivity carried out by M on x. The Turing machine M is deterministic if and only if for all inputs x there is exactly one computation which M can do on this input. In this case the sequence is called a deterministic computation. Often it is enough to consider the sequence of states of the Turing machine starting with the initial state and each subsequent state that gives rise to the next state to have enough information. However, sometimes there is more activity and we may need more information such as the change in symbols, what is left on the tape and also their respective cell positions.

Definition 1.1.1. *A one-tape deterministic Turing machine DM is a tuple* $(Q, \Sigma, \delta, q_0, F)$ *where*

- Q *is a finite set of states*
- Σ *is a finite set of symbols (to be written on the tape)*
- δ *is the transition function:* $Q \times \Sigma \cup \{\#^1\} \to Q \times \Sigma \cup \{\#\} \times (RLS)$
- $q_0 \in Q$ *is the start state of DM*
- $F \subset Q$ *is the set of final states and* $q_R, {q_A}^2 \in F$

The cells of the tape are conventionally numbered as $1, 2, \ldots$ and each cell can store precisely one of the alphabet symbols.

The transition function δ performs the following operations given any input $(q_i, s_k) \in Q \times \Sigma \cup \{\#\}$ with its corresponding output as $(q_j, t_k, R) \in Q \times \Sigma \cup \{\#\} \times (RLS)$. (see Figure 1.1).

Given that the head is currently at the k-th cell of the tape,

1. q_i is the current state of the one-tape deterministic Turing machine and the head reads the symbol s_k in the k-th cell.
2. According to the transition function δ, the old symbol s_k is changed to the new symbol t_k.

[1]Note that the symbol # is not considered as part of the set of alphabets but can be written in order to represent the erasing of symbols.
[2]q_R, q_A are defined as in 1.0.1 and 1.0.2

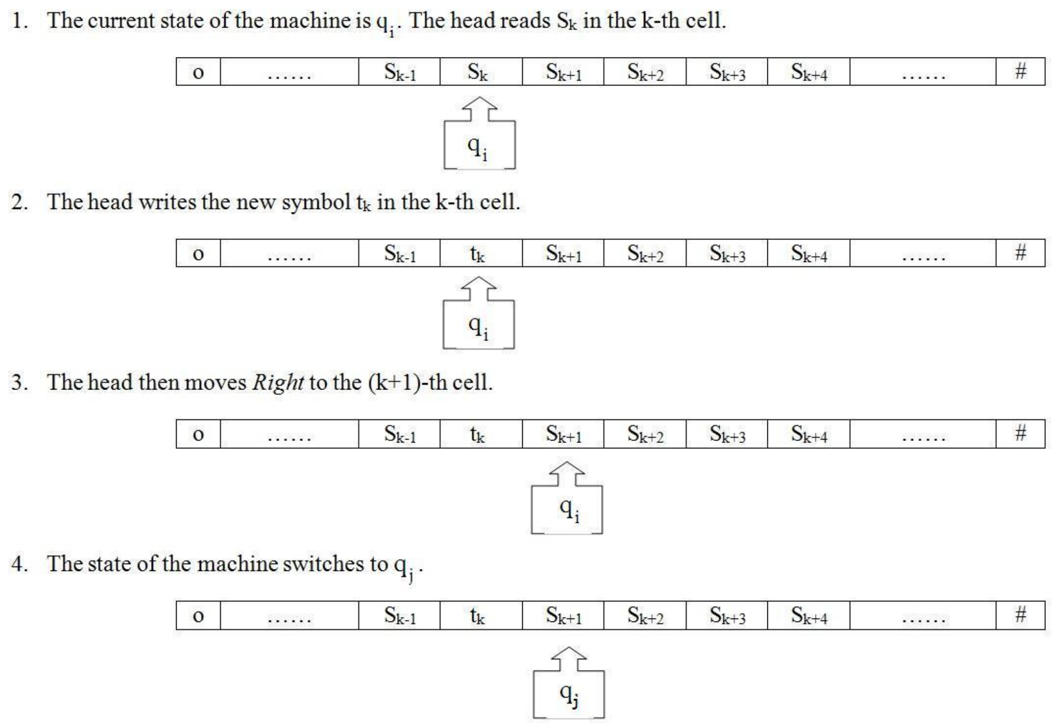

1. The current state of the machine is q_i. The head reads S_k in the k-th cell.
o | ...... | S_{k-1} | S_k | S_{k+1} | S_{k+2} | S_{k+3} | S_{k+4} | ...... | #
q_i
2. The head writes the new symbol t_k in the k-th cell.
o | ...... | S_{k-1} | t_k | S_{k+1} | S_{k+2} | S_{k+3} | S_{k+4} | ...... | #
q_i
3. The head then moves *Right* to the (k+1)-th cell.
o | ...... | S_{k-1} | t_k | S_{k+1} | S_{k+2} | S_{k+3} | S_{k+4} | ...... | #
q_i
4. The state of the machine switches to q_j.
o | ...... | S_{k-1} | t_k | S_{k+1} | S_{k+2} | S_{k+3} | S_{k+4} | ...... | #
q_j

Figure 1.1: Deterministic Turing machine - How it works

3. The letter R represents the movement of the read/write head. In this case, the head moves $Right$ to the $(k+1)$-th cell. It can also move $Left$ to the $(k-1)$-th cell or $Stay$ at the k-th cell.

4. Finally, the one-tape deterministic Turing Machine reaches it new state q_j after the new symbol is written.

The symbol # has a second role: We can also use # to represent all empty (unused) cells in our (semi-)infinite tape. We say that an input x is accepted by DM if the computation of x by DM results in some accepting state q_A i.e $DM(x) = q_A$, .

1.1.2 One-tape Non-deterministic Turing machine

Definition 1.1.2. *A one-tape non-deterministic Turing machine NM is a tuple* $(Q, \Sigma, \delta, q_0, F)$ *where*

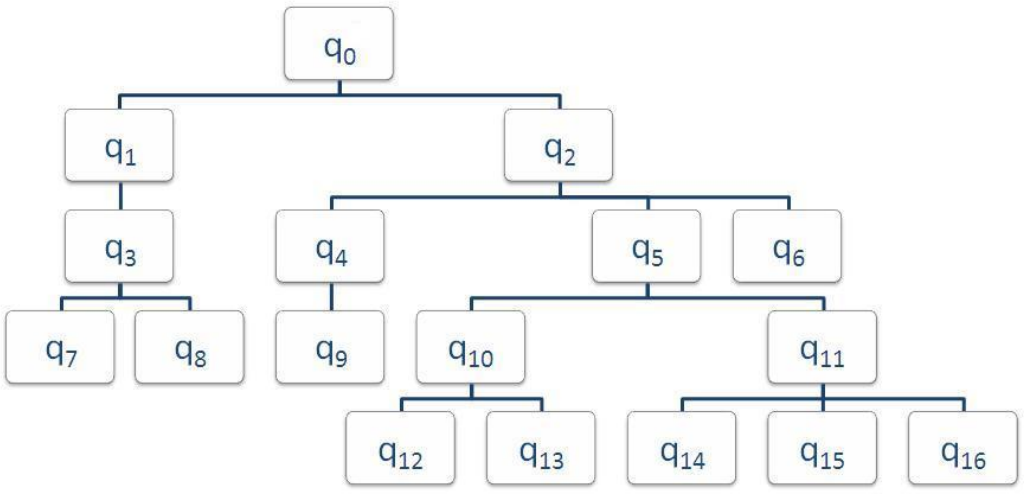


Figure 1.2: Non-deterministic Turing machine - First four levels viewed as a computation tree

- *Q is a finite set of states*
- *Σ is a finite set of symbols (to be written on the tape)*
- *δ is written as a 5-tuple that represents a transition relation i.e. $\delta \subset Q \times \Sigma \cup \{\#\} \times Q \times \Sigma \cup \{\#\} \times (RLS)$*
- *$q_0 \in Q$ is the start state of NM*
- *$F \subset Q$ is the set of final states and $q_R, q_A \in F$*

One way to view the non-deterministic computation would be to look at it as a computation tree (see Figure 1.2).

The nodes of the tree represent each of the states reached and the edges represent the transition from one state to another as the read/write head passes from one cell to the next. A computation path is a path that start from the root of the tree q_0 and ends at one of the final states $q_f \in F$. NM can "guess" a path to an accepting state if one exists and it subsequently carries out the computation of that path. We say that an input x is accepted by NM if the computation tree of NM has *at least one* accepting computation path.

We state now three preliminary results relating deterministic Turing machines and non-deterministic Turing machines.

Theorem 1.1.3. *It is always possible to obtain a deterministic Turing machine DM from a non-deterministic Turing machine NM (without the condition of keeping the same time bounds of each machine)*[3].

Proof. The idea of the proof relies on a technique that does a breadth-first search of NM's computation tree. The DM that we want is the one that consists of only one of the accepted paths of NM. □

Example 1.1.4. *Given an integer n, one wants to find its prime factorisation, that is, the prime numbers $p_1, p_2, \dots, p_k$ such that*

$$\prod_{i=1}^{k} p_i = n$$

To simulate NM that does this computation by DM, let $n_1 := n$. One would need DM to divide n_1 by each prime number $2, 3, 5, \dots$ in this particular order. The first prime number that divides n_1 would be p_1. Then proceed to the next level of NM's computation tree $n_2 = \frac{n_1}{p_1}$. We divide n_2 by each prime number $2, 3, 5, \dots$ again until we find p_2. For each level $l \in \{2, 3, \dots, k+1\}$, repeat the division of $n_l := \frac{n_{l-1}}{p_{l-1}}$ by each prime number $2, 3, 5, \dots$ again, p_{l-1} being the first prime number that divides n_{l-1} at the $(l-1)$-th level. Note that p_a can equal p_b, for some $a, b \in \{1, 2, 3, \dots, k\}$. The process continues until we get $n_{k+1} = 1$ so we have altogether k prime numbers, i.e. $\prod_{i=1}^{k} p_i = n$ is found.

Theorem 1.1.5. *Let q_A be the accepting state of DM and q'_A be that of NM. There exists a constant c such that if $NM(x) = q'_A$ after l steps for any input x, then one knows that $DM(x) = q_A$ in at most c^l steps for the same input x.*

[3]In fact, most variations of Turing machines can be simulated by a one-tape Turing machine, proving that the notion is robust. However, the efficiency of computation might be different.

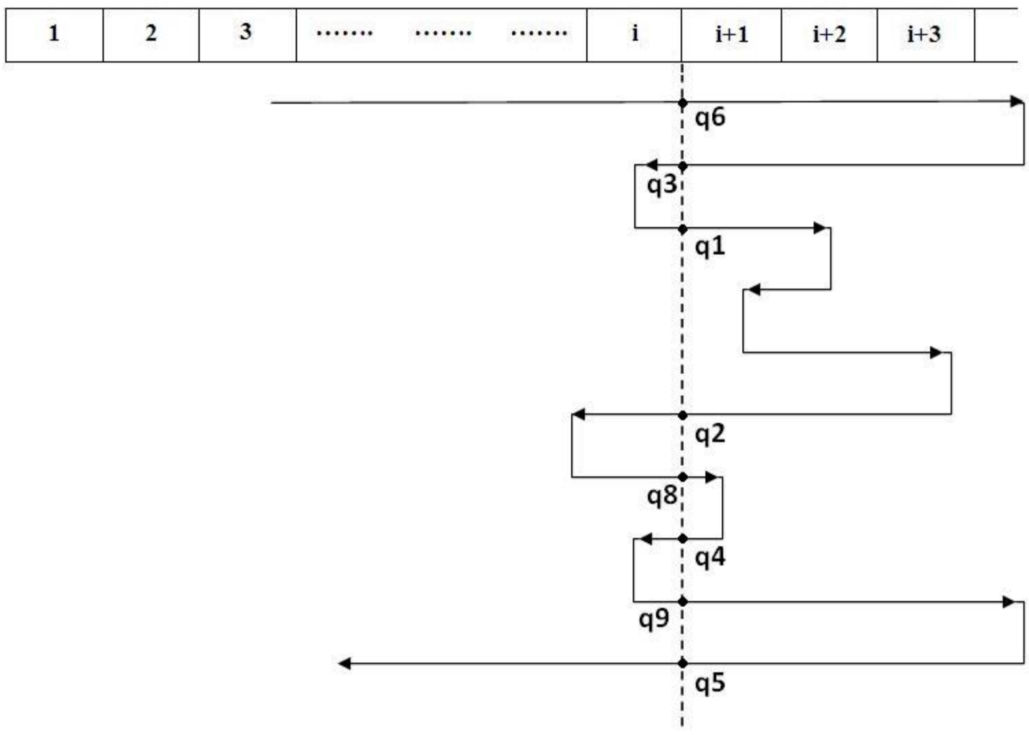


Figure 1.3: Crossing Sequence - Cr_i at the i-th boundary between cell i and cell $i + 1$

1.2 Crossing Sequences

Definition 1.2.1. *Given a one-tape Turing machine and an input n, we define a crossing sequence at the i-th cell Cr_i as the sequence of states in which the Turing machine finds itself when crossing the boundaries between the i-th cell and the $(i+1)$-th cell. These states can also include those that correspond to moves where the head stays in the same position*[4].

The crossing sequence at boundary i as seen in Figure 1.3 is $q_6, q_3, q_1, q_2, q_8, q_4, q_9, q_5$. As mentioned earlier, the state q_1 is the state of the machine before it reads cell $i + 1$; and likewise in the other direction, q_4 is the state of the machine before it reads cell i.

We now state three propositions relating to crossing sequences.

[4]These states are possibly repeated.

Proposition 1.2.2. *If a one-tape Turing machine with inputs x_1 and x_2 gives outputs y_1 and y_2 respectively, and satisfies the following conditions*

1. *the crossing sequences at the i-th cell of both inputs are of equal lengths*
2. *the input symbols after the i-th cell for both x_1 and x_2 are the same*
3. *the first i symbols of y_1 and y_2 are the same*

then $y_1 = y_2$.

Proposition 1.2.3. *The computational time by the machine that is computing a function $f(x)$ is at least the sum of the the lengths of the crossing sequences at each position of x.*

Proposition 1.2.4. *Given a Turing machine TM and input w_m of length n, let $Cr(w_m)$ be some crossing sequence of the input w_m. If w_m is the shortest input such that $Cr(w_m) \geq m$, then every crossing sequence can occur at most twice for the cells of w_m.*

Proof. This is based on the proof in [12, Proposition VIII.1.3]. Let w_m be the shortest word on which the Turing machine has a computation with a crossing sequence $Cr = Cr(w_m)$ of length at least m. In addition, the following argument shows that the Turing machine does not halt in between the two identical crossing sequences that make up the omitted section. Otherwise, the section cannot be omitted.

This is because without loss of generality, there would be a situation where the crossing sequence on the right would have as many forward and backward passes while the one on the left have one more forward pass than backward pass. This immediately means that the crossing sequences cannot be identical.

With that, we suppose now that on the contrary, all words w_m exist, that is, w_m is not the shortest input. We want to arrive at a contradiction by showing that there are no three identical crossing sequences in the word w_m.

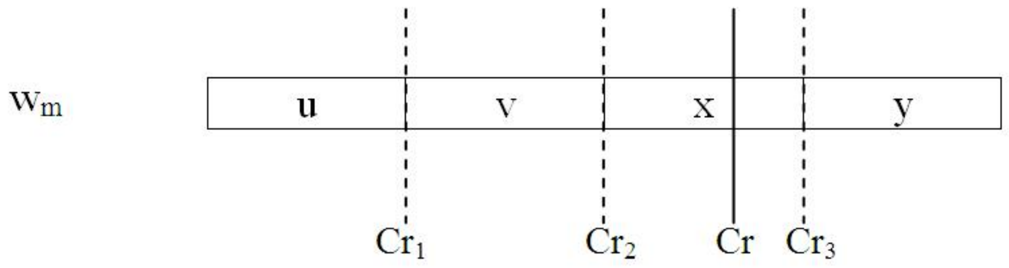


Figure 1.4: There cannot be three identical crossing sequences in the word w_m

Suppose the 3 identical crossing sequences Cr_1, Cr_2, Cr_3 separate w_m into 4 parts, $uvxy$ and Cr the crossing sequence of length at least m in one of the sections (see Figure 1.4). If Cr is not in the section v, then v can be omitted by pumping down the word to give uxy. On uxy, some computation still has the crossing sequence Cr. But $|uxy|$ is shorter than $|w_m|$. If Cr is in the section v, then the pumping can be done on section x and $|uvy|$ is still shorter than $|w_m|$. Either way, we get a contradiction. So we can say that on w_m, the same crossing sequence can occur at most twice. □

Theorem 1.2.5. *Given a one-tape Turing machine TM and constant c for which every non-deterministic run on input of length n needs at most cn steps, then there is a constant c' such that every crossing sequence has at most length c'.*

Proof. This result is also seen in [5] and [14]. By Theorem 1.2.4 and using the same notations, we know that each crossing sequence $Cr(w_m)$ can occur at most twice. Consider now 2 cases.

Case 1:

Suppose there exists a word x for infinitely many m such that $w_m = x$. Then computations on x can be arbitrarily long. This implies that the Turing machine is not running in linear time.

Case 2:

Let d be such that there are at most d^m crossing sequences of length $m, d \geq 8$. This bound exists because for each crossing sequence, there are finitely many symbols. For a given constant c, choose m such that

$$|w_m| \geq 4d^{3c+1}$$

Now, half of the crossing sequences are of length longer that $3c$. Therefore, the runtime of Turing machine is at least $|w_m|\frac{3c}{2}$. The runtime is scaled up by a factor and so Turing machine does not run in time $|w_m| \cdot c$. This means that only finitely many w_m exist and hence the length of the crossing sequence must be bounded. □

1.3 Automata

Informally, an automaton works in a similar way to a Turing machine because it reads an input and moves in the forward direction. However, this movement is only in one direction over the input from left to right and it does not modify the input. At the end of the input, the automaton either accepts or rejects the input.

1.3.1 Deterministic Finite Automata

Definition 1.3.1. *A deterministic finite automaton is defined formally as a 5-tuple DFA* $= (Q, \Sigma, \delta, q_0, F)$ *where*

- Q *is the (finite) set of states*
- Σ *is the (finite) set of symbols*
- δ *is the transition function:* $Q \times \Sigma \rightarrow Q$
- q_0 *is the start state of the automaton*
- $F \subset Q$ *is the set of all final states*

Let q_j be the state reached after the first j symbols $x_1x_2\dots x_j$ of the input string x. Upon reading the next symbol x_{j+1}, the DFA makes a transition into the new state $\delta(q_j, x_{j+1}) := q_{j+1}$. The DFA said to accept the input string if the state reached after the last symbol x_n, q_f is in the set F of final states.

Because we are dealing with strings, one useful extension that is considered is the transition function $\hat{\delta}$, that we call the extended transition function. Intuitively, $\hat{\delta}$ is a multi-step version of δ. This function accepts input strings and not just symbols (of a string) and goes directly from the start state to the final state.

Definition 1.3.2. *The language accepted by a deterministic finite automaton is the set $L(DFA) = \{x \in \Sigma^* : \hat{\delta}(q_0, x) \in F\}$.*

1.3.2 Non-deterministic Finite Automata

Definition 1.3.3. *A non-deterministic finite automaton is defined formally as a 5-tuple NFA $= (Q, \Sigma, \delta, q_0, F)$ where*

- *Q is the (finite) set of states*
- *Σ is the (finite) set of symbols*
- *δ is the transition function: $Q \times \Sigma \to \mathcal{P}(Q)$, where $\mathcal{P}(Q)$ is the power set of Q*
- *q_0 is the start state of the automaton*
- *$F \subset Q$ is the set of all final states*

Because we are dealing with non-determinism, we need to consider a slight variation of the extended transition function. For each $S \subseteq Q$, let $\widehat{\delta}(S, \epsilon) = S$ and define inductively for $x \in \Sigma^*$ and $a \in \Sigma$, $\widehat{\delta}(S, xa) := \bigcup_{q \in \widehat{\delta}(S,x)} \delta(q, a)$

Definition 1.3.4. *The non-deterministic automaton NFA accepts x if and only if $\widehat{\delta}(\{q_0\}, x) \cap F \neq \emptyset$.*

1.4 Pumping Lemma

The main use of the Pumping Lemma is to prove that certain languages are not regular.

Lemma 1.4.1. *The Pumping Lemma states that if R is a regular language, then there is a number $p \in \mathbb{N}$ which represents the pumping length, such that if a string $s \in R$ is of length at least p, then s may be divided into three shorter words $s = xyz$, satisfying the following 3 properties:*

1. *for each $i \in \mathbb{Z}_{\geq 0}, xy^iz \in R$*
2. $|y| > 0$
3. $|xy| \leq p$

Proof. Let DFA be a deterministic finite automaton $(Q, \Sigma, \delta, q_0, F)$ that accepts R, and $p = |Q|$. Suppose $s = t_1t_2 \dots t_n \in L(DFA)$ and $n \geq p$. One has

$$\overbrace{q_0 \xrightarrow{t_1} q_1 \xrightarrow{t_2} q_2 \dots q_{p-1} \xrightarrow{t_p} q_p}^{p+1 \text{ states}} \dots q_n \in F$$

By the pigeonhole principle, $q_0, q_1, \dots, q_p$ must have repeat states, that is, $q_j = q_k$ for some $0 \leq j < k \leq p$. One can write the above as

$$q_0 \xrightarrow{x} q_j \xrightarrow{y} (q_k = q_j) \xrightarrow{z} q_n \in F$$

where $x = t_1t_2 \dots t_j$, $y = t_{j+1}t_{j+2} \dots t_k$ and $z = t_{k+1} \dots t_n$. Then one can see that $|xy| \leq p$ and $|y| > 0$. Also, for every $i \geq 0$, since one has

$$q_0 \xrightarrow{x} \overbrace{q_j \xrightarrow{y} q_j \dots \xrightarrow{y} q_j}^{i \text{ times}} \xrightarrow{z} q_n \in F$$

the automaton has to accept all words of the form xy^iz, i.e. $xy^iz \in L(DFA)$. $\square$

Remark 1.4.2. *Note that one can in the above proof choose j, k such that $n - p \leq j < k \leq n$. Then the x, y, z selection satisfies $s = xyz$ with the following 3 conditions:*

1. *for each $i \in \mathbb{Z}_{\geq 0}, xy^iz \in R$*
2. *$|y| > 0$*
3. *$|yz| \leq p$*

This permits one to do pumping near the end of the word.

Chapter 2

Automatic Functions and Linear Time

Informally, an automatic function is a function from strings to strings whose graph is recognised by a finite automaton. More formally, this is based on the notion of convolution. In this chapter, we will introduce the concept of convolution of strings and proceed to show two key results between automatic functions and one-tape Turing machines.

2.1 Convolution

Definition 2.1.1. *Suppose x and y are two strings such that $x = x_1x_2 \dots x_m$ and $y = y_1y_2 \dots y_n$. Let $x' = x'_1x'_2 \dots x'_r$ and $y' = y'_1y'_2 \dots y'_r$, where*

i) $r = \max(m, n)$,

ii) $x'_i = \begin{cases} x_i, & \text{if } i \leq m \\ \#, & \text{otherwise} \end{cases}$

iii) $y'_i = \begin{cases} y_i & \text{if } i \leq n \\ \# & \text{otherwise} \end{cases}$

Then the convolution of the two strings x and y is

$$conv(x, y) = (x'_1, y'_1)(x'_2, y'_2) \dots (x'_r, y'_r).$$

We can define the convolution of a fixed number of strings similarly in order to define functions which have a fixed number of inputs instead of one. We can use convolutions also to define functions with several inputs computed by one-tape Turing machines. The exposition in this chapter just follows the basic case of mapping strings to strings.

Definition 2.1.2. *A function f is called automatic if and only if there is a deterministic finite automaton which recognises the convoluted input-output pairs; that is, given $conv(x, y)$, the automaton accepts if and only if x is in the domain of f and $f(x) = y$.*

Note that in the above Definition 2.1.2, one could replace the requirement that the automaton is deterministic by working with non-deterministic finite automata. The class of functions defined would then be the same.

Theorem 2.1.3 states the importance of the concept of automatic functions and automatic relations.

Theorem 2.1.3. *Every function, which is first-order definable from finite number of automatic functions and relations, is automatic again and the corresponding automaton can be computed effectively from the other automata.*

This gives the second nice fact that structures consisting of automatic functions and relations have a decidable first-order theory. This is presented in [8] and [9].

Throughout the thesis, we only consider one-tape Turing machines where the tape content before the computation is the input and the tape content after the computation is the output. These Turing machines can be either deterministic or non-deterministic. We define that a Turing machine computes a function f as follows.

Definition 2.1.4. *We say that a (not necessarily deterministic) Turing machine computes a function f if and only if the following two conditions are satisfied:*

- *for each x in the domain of f, there is a computation which replaces x on the tape by the output $f(x)$ and which halts.*
- *if a computation of the machine starts with input x on the tape and halts with output y on the tape, then x is in the domain of f and $f(x) = y$.*

Note that every deterministic Turing machine can by definition also be viewed as a non-deterministic one, hence every function computed by a deterministic Turing machine is also computed by a non-deterministic one satisfying the same complexity bounds.

The main result of this chapter is that the following three models are equivalent:

- automatic functions
- functions computed in deterministic linear time by a one-tape Turing machine where input and output start at the same position
- functions computed in non-deterministic linear time by a one-tape Turing machine where input and output start at the same position

This equivalence is shown in the following two results, where the first one generalises a remark in [3] stating that an automatic function can be computed in linear time.

2.2 Automatic Functions and Deterministic Linear Time One-tape Turing machines

Theorem 2.2.1. *Let f be an automatic function. Then there is a deterministic linear time one-tape Turing machine which replaces any legal input x on the tape by the output $f(x)$ starting at the same position as x before.*

Proof. Assume that a deterministic automaton with c states (numbered 1 to c, where 1 is the starting state) accepts a word of the form $\text{conv}(x, y) \cdot (\#, \#)$ if and only if x is in the domain of f and $y = f(x)$; the automaton rejects any other sequence. Suppose input is $x = x_1 x_2 \ldots x_r$. Now one considers the following additional work-tape symbols consisting of all tuples $(a, s_1, s_2, \ldots, s_c)$:

- a is $\#$ or one of x_k's
- for $d \in \{1, 2, \ldots, c\}$, s_d takes the values $-, +$ or $*$

The symbols $-, +$ and $*$ are used in 2 different ways. First, consider the k-th cell:

- $s_d = -$ if and only if there is **no** word of the form $y_1 y_2 \ldots y_{k-1}$ such that the automaton on input $(x_1, y_1)(x_2, y_2) \ldots (x_{k-1}, y_{k-1})$[1] reaches the state d
- $s_d = +$ if and only if there is **exactly one** such word
- $s_d = *$ if and only if there are **at least two** such words.

Now the Turing machine simulating the automaton replaces the cell to the left of the input by o,[2] the cell containing x_1 by $(x_1, +, -, \ldots, -)$. Then, for each new cell with entry x_k (from the input or $\#$ if that has been exhausted) the Turing machine replaces x_k by $(x_k, s_1, s_2, \ldots, s_c)$ under the following conditions, (where the entry in the previous cell was $(x_{k-1}, s'_1, s'_2, \ldots, s'_c)$):

- $s_d = +$ if and only if there is exactly one (y_{k-1}, d') such that $s'_{d'}$ is $+$ and the Turing machine transfers on (x_{k-1}, y_{k-1}) from state d' to d and there is no pair (y_{k-1}, d') such that $s'_{d'}$ is $*$ and the Turing machine transfers on (x_{k-1}, y_{k-1}) from d' to d;
- $s_d = *$ if and only if there are at least two pairs (y_{k-1}, d') such that $s'_{d'}$ is $+$ and the Turing machine transfers on (x_{k-1}, y_{k-1}) from state d' to d or

[1] Here the x_i and y_i can also be $\#$ when a word has been exhausted.
[2] Recall that this just marks the start position of the input on the tape.

there is at least one pair (y_{k-1}, d') such that $s'_{d'}$ is $*$ and the Turing machine transfers on (x_{k-1}, y_{k-1}) from d' to d;

- $s_d = -$ if and only if for all pairs (y_{k-1}, d') such that the Turing machine transfers on (x_{k-1}, y_{k-1}) from d' to d, it holds that $s'_{d'}$ is $-$.

These 3 cases are mutually exclusive.

The Turing machine replaces each symbol in the input as above until it reaches the cell where the intended symbol $(a, s_1, s_2, \ldots, s_c)$ has $s_d = +$ for some accepting state d. If this happens, the Turing machine memorises the state d, turns around, erases this cell and goes backward.

When the Turing machine comes backward from the cell $k + 1$ to the cell k, where the state memorised for the cell $k + 1$ is d', then it determines the unique (d, y_k) such that $s_d = +$ (as stored in cell k) and the Turing machine transfers from d to d' on (x_k, y_k); now the Turing machine replaces the symbol on cell k by y_k (if $y_k \neq \#$) and by the blank symbol (if $y_k = \#$). Then the Turing machine keeps the state d in the memory and goes to the left and repeats this process until it reaches the cell which has the symbol o on it. Once the Turing machine reaches there, it replaces this symbol by the blank and terminates.

For the verification, note that the output $y = y_1 y_2 \ldots$ (with # appended) satisfies that the Turing machine, after reading $(x_1, y_1)(x_2, y_2) \ldots (x_k, y_k)$, is always in a state d with $s_d = +$ (as written in cell $k + 1$ in the algorithm above), as the function value y is unique in x; thus, whenever the Turing machine ends up in an accepting state d with $s_d = +$ then the input-output-pair $\mathrm{conv}(x, y) \cdot (\#, \#)$ has been completely processed and $x \in \mathit{dom}(f) \wedge f(x) = y$ has been verified. Therefore, the Turing machine can turn and follow the unique path, marked by + symbols, backwards in order to reconstruct the output from the input and the markings. All superfluous symbols and markings are removed from the tape in this process. As y depends uniquely on x, the Turing machine accepting $\mathrm{conv}(x, y)$ can accept at most c symbols after the word x; hence the runtime of the Turing machine is bounded by $2 \cdot (|x| + c + 2)$, that is, the runtime is linear. $\square$

Note that this deterministic Turing machine makes two passes: one from the origin to the end of the word and one back. These two passes are needed as guessing is not possible. To see this, consider the function $f(x_1x_2\ldots x_{k-1}x_k) = x_k\,x_2\ldots x_{k-1}x_1$, where the first and last symbol are exchanged and the others remain unchanged. For a deterministic one-tape Turing machine, one can observe that this action of swapping the first and last symbol cannot be done in one pass.

2.3 Automatic Functions and Non-Deterministic Linear Time One-tape Turing Machines

Theorem 2.3.1. *Let f be a function computed by a non-deterministic one-tape Turing machine in linear time, with the input and output starting at the same position. Then f is automatic.*

Proof. The proof is based on crossing sequences. We can also read about this in [7].

Without loss of generality, one can assume that there is a special symbol o to the left of the input occurring only there and that the automaton each time turns when it reaches this position. Furthermore, it starts there and returns to that position at the end. A computation accepts only when the full computation has been accomplished and the automaton has returned to its origin o.

The proof method of the above result is done by showing that if it is computable in time $|x|$, there is a constant c' so that every computation visits each cell at most c' times, otherwise the function f would not be linear time computable. This is possible by Theorem 1.2.5. This permits to represent the computation locally by storing for each visit to a cell — the direction from which the Turing machine entered the cell, in which state it was, what activity it did and in which direction it left the cell. This gives, for each cell, only a constant amount of information which can be stored in the cell using a sufficiently large alphabet.

Now a non-deterministic automaton can recognise the set $\{conv(x, y) : x \in$

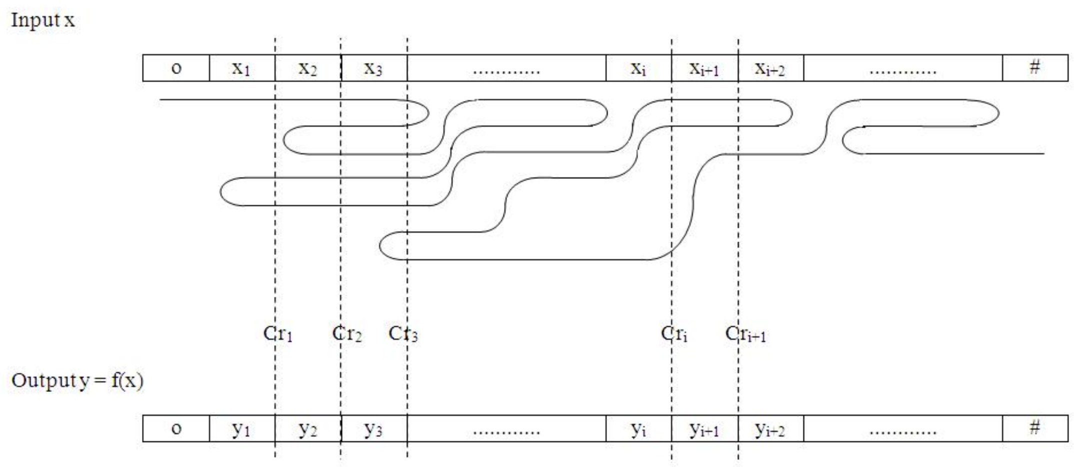


Figure 2.1: Non-deterministic Turing machine recognising $conv(x, y)$

$dom(f) \wedge y = f(x)\}$ by guessing on each cell, the local information of the visits of the Turing machine. It compares it with the information from the previous cell and checks whether it is consistent. So, suppose the input $x = x_1x_2x_3\ldots$ is transformed to output $f(x) = y = y_1y_2y_3\ldots$. The machine goes over the string and at each time, it guesses the crossing sequence at the i-th cell Cr_i. It then guesses the crossing sequence at the $(i+1)$-th cell Cr_{i+1} and sees if the activities match and whether x_{i+1} gets transformed to y_{i+1}. This is only a finite amount of information as the finite number of crossing sequences have finite length.

If you go from the left to right, the computation is that in each cell, it can visit at most finitely many times. So the non-deterministic computation looks like Figure 2.1.

The automaton passes over the full word and when it reaches the end, the machine should have reached an accepting state, somewhere along the computation. This occurs when the input matches the output and the computation is verified at every step from beginning to end. However, the computations are rejected if there is inconsistency. As per all non-deterministic automata, there needs to be at least one accepting run and this run corresponds to the non-deterministic Turing machine that terminates. If there is a run with a wrong result, then the non-deterministic Turing machine cannot terminate and this implies that there was a problem in the verification in the automaton. So the automaton accepts $conv(x, y)$ if and only if the non-deterministic computation transforms some input of the form $ox\#^*$ into some output of the form $oy\#^*$.

During the verification process, certain conditions must be noted, namely, when changing direction, the computation must move onto the next layer (see Figure 2.1)[3] below the preceding computation, so that when reading the crossing sequences, one can read them off systematically in order of occurrence. □

Lastly, the equivalence of the three models is seen as a result of Theorem 2.2.1 and Theorem 2.3.1.

2.4 Importance of the input-output start position

One might ask whether the condition on the input and output starting at the same position is really needed. The answer is "yes".

Suppose that on the contrary, this condition is not needed and that all functions linear time computable by a one-tape Turing machine without any restrictions on output positions are automatic. Then one could consider the free monoid over $\{0, 1\}$. For this monoid $\{0, 1\}^*$, the following function could be computed from $\text{conv}(x, y)$.

Let the output be

$$f(x, y) = \begin{cases} z & \text{if } y = xz \\ \# & \text{if such a } z \text{ does not exist} \end{cases}$$

For this, the machine just compares x_1 with y_1 and erases (x_1, y_1), x_2 with y_2 and erases (x_2, y_2) and so on, until it reaches

[3]This just ensures that the states of the crossing sequences are represented in the right order.

i) a pair of the form (x_m, y_m) with $x_m \neq y_m$

ii) a pair of the form $(x_m, \#)$

iii) a pair of the form $(\#, y_m)$

iv) the end of the input

In cases i) and ii), the output has to be # and the machine just erases all remaining input symbols and puts the special symbol # to denote the special case. In case iii), the value z is just obtained by changing all remaining input symbols $(\#, y_k)$ to y_k and the Turing machine terminates. In case iv), the valid output is the empty string and the Turing machine codes it adequately on the tape. Hence f would be automatic.

But now one could first-order define concatenation g by letting $g(x, z)$ be that y for which $f(x, y) = z$; this would give that the concatenation is automatic, which is known to be a false statement.

Proof. If f is automatic, then $\exists c \forall x[|f(x)| \leq |x| + c]$. However, when we consider the set $\{conv(x, f(x)) : x \in dom(f)\}$ and use Lemma 1.4.1, we can see that given any 2 strings x and y and a pumping constant c, there is the case when the concatenation of these 2 strings might result in the difference between the lengths of the input and output to be much larger than the pumping constant, i.e. $|x + y| - |x| >> c$. However, the output of an automatic function can only be a constant longer than the input. For any string y of arbitrary length, we are not guaranteed of this condition. Hence concatenation is not automatic. □

Thus, the condition on the starting-positions cannot be dropped.

Chapter 3

Automatic Functions as a Model of Computation

The next model of computation studied is one that uses automatic functions as primitive operations. We will name the model as AUTOFUNC and use the abbreviation AF in the following definition.

Definition 3.0.1. *AF[h(n)] defines the class of all functions which are computed in the following machine model in $O(h(n))$ steps on input of length n.*

Here a machine uses strings as data and can do instructions of the following type:

- *Assignments of the type "$a = f(b, c)$" where f is a fixed automatic function.*
- *Conditional branches of the type "If $a \in R$ then go to ln[1] " where R is a regular set.*
- *Unconditional jumps.*
- *The command halt.*

Initially the input is in a fixed variable x and after the computation the output is in a variable y. The machine can have any finite number of variables and the data

[1]ln will be defined later on in Proposition 3.1.1

type of each variable is a string over a fixed alphabet. The steps of the machine are interpreted in an obvious way, where the functions f for updating variable values and the automatic relations R can have any fixed arity not exceeding the number of variables.

For example, the function $f(x) = xx$ is in $AF[n]$. The algorithm can be written as

1. $y = x$.
2. $z = x$.
3. If $z = \epsilon$ then go to 7.
4. Append the first symbol of z at the end of y.
5. Remove the first symbol from z.
6. Go to 3.
7. Halt.

Note that the instructions in 4 and 5 can be written via automatic functions of the form $y = f(y, z)$ and $z = g(z)$ where $f(y, z)$ consists of y plus the first symbol of z while $g(z)$ consists of all symbols of z except the first one. The run-time is $O(n)$ as the program goes only n rounds through the loop from lines 3 to 6 where n is the number of symbols in x.

A computation halts when the program reaches the halting state and a program is an AF[h(n)] program only if there is a constant c such that the program reaches the halting state for every input $x \in \{0, 1\}^n$ within $h(n) \cdot c + c$ steps.

3.1 How AUTOFUNC works

A Turing machine carries out computations in the local sense. The usefulness of AUTOFUNC is where one can directly process information from the function.

One would use automatic functions as operations on the inputs and can do automatic updates in one step. Thus each step of this model can do something more intelligent than a Turing machine.

In this chapter, we would make a program which uses automatic functions to manipulate its data and do its test (with finitely many string variables) and then compare its complexity with other models of computation, in particular, $DTIME$ and NAF.

Proposition 3.1.1. *Let Σ be the set of symbols. A function f is in $AF[h(n)]$ if and only if there are automatic functions F, G, H such that f is computed by the algorithm*

```
Input $x$ and let $z = F(x)$;
While $G(z) \neq z$,    do $z = G(z)$;
Let $y = H(z)$ and output $y$
```

in a way that the inner loop is run at most $h(n)$ times for input of the length n.

Proof. One direction is clear and follows by definition. If $f \in AF[h(n)]$, then its computational time must be of order $O(h(n))$ for an input of length n.

For the other direction, we need that the simulation uses the convolution of the line number (ln) (for the counting of number of steps carried out) and also all variables involved as values for the configuration. Let $z = conv(ln, x, y, z_1, z_2, \cdots, z_k)$ represent the whole configuration of the machine that simulates the model AUTOFUNC. The function $F(x) = (1, x, \epsilon, \epsilon, \epsilon, \cdots, \epsilon)$ makes the initial configuration with input x.

This algorithm can be written as a program, given as a list of numbered statements. For example,

1. $z_1 = f_1(x)$

2. If $R_2(x, z_1, z_2)$, then go to 8.

3. $z_3 = f_2(z_1, x)$
4. $z_2 = max(z_1, z_3)$
5. ...
6. ..
7. .
8. Halt

The function G is an updating function of the simulation.

$$G\big(conv(ln, x, y, z_1, z_2, \cdots, z_k)\big) = \begin{cases} Halt & \text{if } ln = 8 \\ conv(2, x, f_1(x), z_1, z_2, \cdots, z_k) & \text{if } ln = 1 \\ conv(8, x, y, z_1, z_2, \cdots, z_k) & \text{if } ln = 2 \\ & \text{and } R_2(x, z_1, z_2) \\ conv(3, x, y, z_1, z_2, \cdots, z_k) & \text{if } ln = 2 \\ & \text{and } \neg R_2(x, z_1, z_2) \\ \quad\cdots\cdots & \cdots \\ \quad\vdots & \end{cases}$$

This updating step is carried out until the *While* loop is void, i.e. the configuration does not change. Then the function $H\big(conv(ln, x, y, z_1, z_2, \cdots, z_k)\big) = y$ extracts the output y.

The inner loop runs as long as the algorithm executes a command, until Halt is reach. Each update of $G(z)$ uses up one step of the algorithm. Due to this 1-to-1 correspondence between the number of steps taken and the iterations of the function G, we know that if the algorithm runs $O(h(n))$ times, then the $While$ loop runs $O(h(n))$ steps until it reaches a fixed point in the algorithm. $\square$

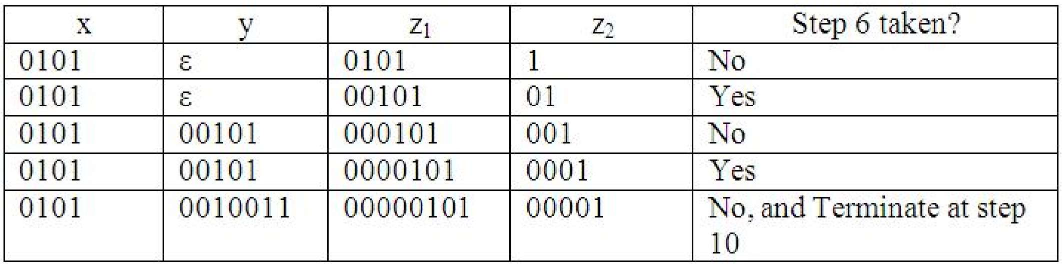
For the following example, one can use the fact that one can implement binary addition, comparison and bit-wise "and" on natural numbers via automatic functions.

Example 3.1.2. *Suppose we have this program that computes the square of the input* x*. The configuration* (ln, x, y, z_1, z_2) *has* $x = 0101$ *being the input to be read,* y *being the output and* z_1 *and* z_2 *being the auxiliary variables.*

Note that one in the field of automatic structures stores binary numbers in reverse order, so 001 represents four and 0101 represents ten. This is the same for $z_2{}^2$. This is because the convolution of automatic functions reads from the left. Hence we need to "reorder" the binary number such that the ones are read first before the tens and so on.

This algorithm runs in $O(n)$ steps and the program is as follows, with + being binary addition operator.

1. $z_1 = x$
2. $z_2 = 1$
3. $y = \epsilon$
4. If z_2 is at a position of a 0 in x, then go to 7.
5. If $|z_2| > |x|$, then go to 10.
6. If z_2 is not at a position of a 0 in x, then $y = y + z_1$.
7. $z_1 = z_1 + z_1$
8. $z_2 = z_2 + z_2$
9. Go to 4.
10. Halt.



x	y	z_1	z_2	Step 6 taken?
0101	ε	0101	1	No
0101	ε	00101	01	Yes
0101	00101	000101	001	No
0101	00101	0000101	0001	Yes
0101	0010011	00000101	00001	No, and Terminate at step 10

Figure 3.1: Example 3.1.2 - How the variables update as the algorithm runs

z_2 plays the role of a pointer to indicate which symbol in x is being read at each step. For example, when $z_2 = 001$, then it is pointing at the third symbol of x. The third symbol of x is a 0, which would bring us to step 4 in the algorithm. When $z_2 = 00001$, its length is longer than $|x|$, which leads us to step 5 and subsequently the halting step 10. We can see from Figure 3.1 that the output is finally 0010011, which in binary form 1100100 stands for one hundred, the square of ten.

3.2 $AF[1] \subset AF[\log n]$

We show in this section that if $f \notin AF[1]$, then f needs at least $O(\log n)$ steps.

Example 3.2.1. *Consider the function $f_{bin}(x)$ that converts an input x of length n into a binary number representing n.*

We first show that this function is not automatic. We can see this by applying Lemma 1.4.1 and refering to Remark 1.4.2

$$\text{Input} \quad x = \underbrace{1010}_{x_1}\underbrace{01}_{x_2}\underbrace{1}_{x_3} \qquad \text{has a length of 7}$$

$$\text{Output} \quad z = \underbrace{111\#}_{z_1}\underbrace{\#\#}_{z_2}\underbrace{\#}_{z_3} \qquad \text{has a length of 3}$$

[2]The binary addition operation is carried out in the usual manner (with the none reversed binary numbers).

Let $z = z_1z_2z_3$ be made up of 3 sections as seen above. We will do pumping on the middle section x_2 and z_2. By Definitions 1.0.4 and 2.1.2, if a function is automatic, then the set of all convolutions of inputs and outputs is regular. We note that the input x is much longer than the output z. Hence in this regular set, after pumping, we get

$$\begin{array}{lll}
\text{Input (pumped)} & x = \underbrace{1010}_{x_1}\underbrace{01}_{x_2}\underbrace{01}_{x_2}\underbrace{1}_{x_3} & \text{has a length of 9} \\
\text{Output (pumped)} & z = \underbrace{\overbrace{111}^{|f_{bin}(x)|}\ \#}_{z_1}\underbrace{\#\#}_{z_2}\underbrace{\#\#}_{z_2}\underbrace{\#}_{z_3} & \text{has a length of 3}
\end{array}$$

Only the input is modified. The output is not changed. Looking at $z_1z_2^*z_3$, we observe that the input gets longer but the output coded inside z is not modified, because only the number of # are increased, which do not affect overall the output[3].

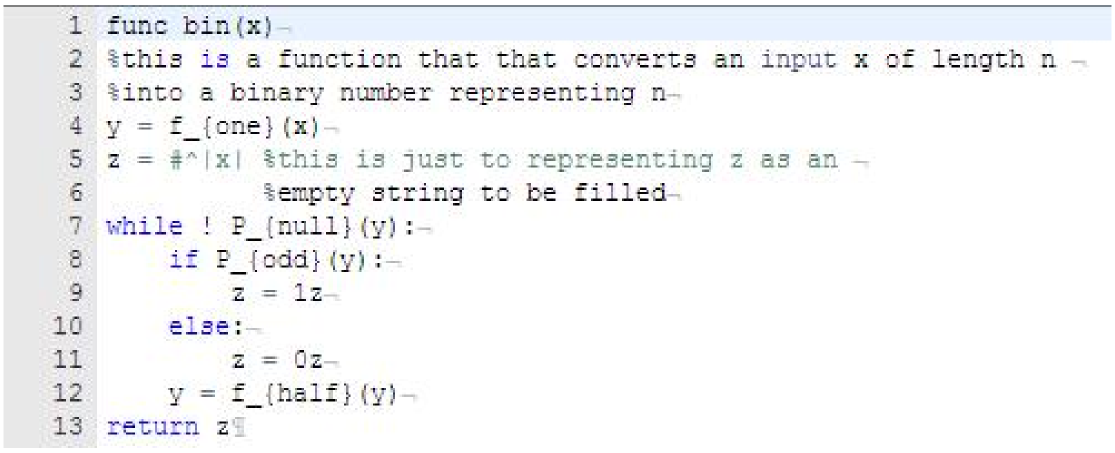
Initially, the function is finite to one, that is, the function has finitely many values that are mapped to the same output. Now, by pumping lemma, there can be infinitely many inputs that go to the same output. We obtain thus a function that is properly infinite to one. So there are infinitely many inputs that are mapped to the same output. This shows that $f_{bin}(x) \notin AF[1]$.

To see that $f_{bin}(x) \subset AF[\log n]$, let the propositions

$$\begin{array}{ll}
P_{odd}(x) & \text{true if } x \text{ has an odd number of 1's} \\
P_{null}(x) & \text{true if } x \in 0^*
\end{array}$$

Subsequently, define the following two functions

[3]The pumping can be placed at any position of the output, for this example, we pump at the section z_2.



```
func bin(x)
%this is a function that that converts an input x of length n
%into a binary number representing n
y = f_{one}(x)
z = #^|x| %this is just to representing z as an
          %empty string to be filled
while ! P_{null}(y):
    if P_{odd}(y):
        z = 1z
    else:
        z = 0z
    y = f_{half}(y)
return z
```

Figure 3.2: $f_{bin}(x)$ code

$$f_{half}(x) = \begin{cases} \text{the function that makes each } (2k+1)\text{-th} & \\ \text{position 1 to 0 for } k \in \mathbb{N} & \text{if } |x| \geq 2 \\ 1 & \text{if } |x| = 1 \end{cases}$$

$$f_{one}(x) = 1^{|x|}$$

The code for $f_{bin}(x)$ would be as seen in Figure 3.1.

Suppose the input

$$x = 1010011$$

When computed, $f_{bin}(x)$ first determines

$$y = f_{one}(x) = 1111111$$
$$z = \#^{|x|} = \#\#\#\#\#\#\#$$

then carries out the following steps:

- Step 1 -

$$\begin{aligned} P_{odd}(y) &= \text{true} \\ z &= 1\#\#\#\#\#\# \\ y &= 0101010 \end{aligned}$$

- Step 2 -

$$\begin{aligned} P_{odd}(y) &= \text{true} \\ z &= 11\#\#\#\#\# \\ y &= 0001000 \end{aligned}$$

- Step 3 -

$$\begin{aligned} P_{odd}(y) &= \text{true} \\ z &= 111\#\#\#\# \\ y &= 0000000 \end{aligned}$$

- Step 4 -

$$z = 111$$

which is the output and also the binary representation of 7, the length of input x. This function runs in at least $\log n$ time.

Example 3.2.2. *The sorting function $f_{sort}(x)$ that sorts symbols from the input $x \in \{0,1\}^*$ such that the number of each symbol remains the same but the output is 0^i1^j where $i + j = |x|$.*

First we show that the sorting function is not automatic.

Proof. To see this, we can consider the sorting function with input 1^n0^n that will be sorted to 0^n1^n with n larger than the pumping constant c. If we apply Lemma

1.4.1 and pump the the front sections of the input and output, the number of 1s of the input increases and the number of 0s of the output increases. That is, 1^{n+c} gets mapped to 0^{n+c}. □

However, we show that f_{sort} can be done in $O(\log n)$.

Suppose that

$$x = 1110100100011101100111111$$

The machine will first count with time $O(\log n)$ in the same way as in Example 3.2.1, the number of zeros and remember its binary code. For our choice of x, there are nine 0's so we get the binary representation 1001. Next the machine copies the whole output but replaces the 0's by 1's, to get

$$1111111111111111111111111$$

Based on the digits of the binary representation read from left to right, and $\forall k \in \mathbb{N}$

- if the machine reads 1 in the binary representation, it replaces every $(2k+1)$-th 1 by a 0
- if the machine reads 0 in the binary representation, it replaces every $(2k)$-th 1 by a 0

Start	:1111111111111111111111111
Step 1 (binary digit read: 1)	:0101010101010101010101010
Step 2 (binary digit read: 0)	:0100010001000100010001000
Step 3 (binary digit read: 0)	:0100000001000000010000000
Step 4 (binary digit read: 1)	:0000000001000000000000000

At Step 4, the machines recognises that there are indeed nine 0's in front of the first 1. What it does next is to replace all subsequent 0's after the first 1 in Step 4 by 1's to get

$$00000000011111111111111$$

The input is now sorted.

Based on Examples 3.2.1 and 3.2.2, since there are functions that can be done in $O(\log n)$ which are not automatic, we conclude that $AF[1] \subset AF[\log n]$.

3.3 $AF[\log n] \subset DTIME[n \log n]$

We consider now the function $f_{dl}(x) = a^{2|x|}$, that produces an output that is twice the length of the input x. Let's first describe how a Turing machine that counts up to twice the length of the input works.

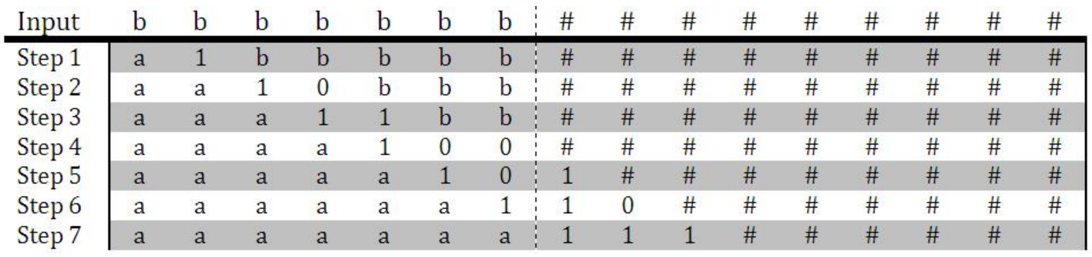
Suppose the input alphabet consists of $\{a, b\}$. We denote the counting alphabet by the set $\{0, 1\}$ and counted in binary numbers. We start with a given input of 7 symbols. We use #s to mark the end of the input and all subsequent cells.

$$bbbbbbb\#\#\#\#\#\#\#\#\#\#\#\#\#\#$$

At each step, the head moves right by one position after changing each b to the output a. This results in the same number of a's as the number of steps carried out so far. The next few cells will then be written with the counting alphabet depending on the binary representation of the number of steps carried out so far. In other words, these two symbols count how many digits have been processed up to the current step. We can consider $\{0, 1\}$ playing the role as the auxiliary memory. What we want to see is that the last line has 14 a's.

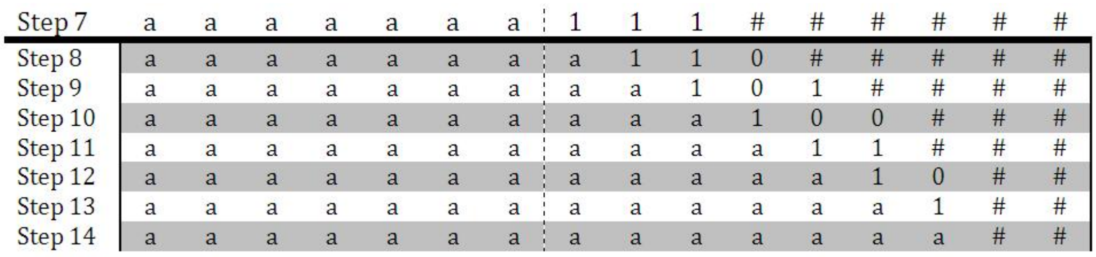
The process of the first seven steps is shown in Figure 3.2.

When the machine reaches the first #, the auxiliary memory starts to count back down but its head continues to move forward in the same way as before,



Input	b	b	b	b	b	b	b	#	#	#	#	#	#	#	#	#
Step 1	a	1	b	b	b	b	b	#	#	#	#	#	#	#	#	#
Step 2	a	a	1	0	b	b	b	#	#	#	#	#	#	#	#	#
Step 3	a	a	a	1	1	b	b	#	#	#	#	#	#	#	#	#
Step 4	a	a	a	a	1	0	0	#	#	#	#	#	#	#	#	#
Step 5	a	a	a	a	a	1	0	1	#	#	#	#	#	#	#	#
Step 6	a	a	a	a	a	a	1	1	0	#	#	#	#	#	#	#
Step 7	a	a	a	a	a	a	a	1	1	1	#	#	#	#	#	#

Figure 3.3: $f_{dl}(x) = a^{2|x|}$ - Steps 1 to 7



Step 7	a	a	a	a	a	a	a	1	1	1	#	#	#	#	#	#
Step 8	a	a	a	a	a	a	a	a	1	1	0	#	#	#	#	#
Step 9	a	a	a	a	a	a	a	a	a	1	0	1	#	#	#	#
Step 10	a	a	a	a	a	a	a	a	a	a	1	0	0	#	#	#
Step 11	a	a	a	a	a	a	a	a	a	a	a	1	1	#	#	#
Step 12	a	a	a	a	a	a	a	a	a	a	a	a	1	0	#	#
Step 13	a	a	a	a	a	a	a	a	a	a	a	a	a	1	#	#
Step 14	a	a	a	a	a	a	a	a	a	a	a	a	a	a	#	#

Figure 3.4: $f_{dl}(x) = a^{2|x|}$ - Steps 8 to 14

adding a's to replace every # after each subsequent step. The machine stops when the (counter) number has vanished. This occurs exactly at the $2|x|$-th step and at this step, one has exactly $2|x|$ a's, in this case

$$aaaaaaaaaaaaaa$$

as seen in Figure 3.3.

For this particular Turing machine computing $f_{dl}(x)$, locally, each step requires the counting of $O(\log n)$ of binary digits. This is because the operations are not atomic and require several steps of the Turing machine which include reading, erasing, writing etc. If there are n symbols in the input, there will be a total of $2n$ steps. Hence, the total run time of the computation is of order $O(n \log n)$.

In addition, we know that this function cannot be computed by a machine using depth of $O(\log n)$ of concatenated automatic functions. This is because an automatic function can increase the input by a length of a constant c number of symbols so that the output is $n + c \log n$. This is not the case since we see that the output of the function $f_{dl}(x)$ has a length $2n$. Thus, it is not within the complexity class of our model $AF[\log n]$[4].

Hence the function $f_{dl}(x) = a^{2|x|}$ proves that it is not possible to speed up the computation of f by our AUTOFUNC. Hence $AF[\log n]$ is a sub class of $DTIME[n \log n]$.

Remark 3.3.1. *The mode of counting can also be done in ternary and decimal mode, as long as the counting alphabets are not part of the input alphabet. On the other hand, using a unary counter would make the machine run back and forth in time $O(n)$, thus making the total run time quadratic $O(n^2)$.*

3.3.1 Importance of the input-output start position

We have seen in section 2.4 that the significance of the having the input-output position restriction. If a function does not have this restriction, then there would be more movement of information if we want to move it back into the time complexity class of $O(n \log n)$.

We first look at f_{diff} that starts by erasing some of the input symbols.[5] Suppose that we have an input of the form $v2w$ with $v \in \{0\}^n$ and $w \in \{0, 1\}^n$ being mapped to $2w$; inputs without a 2 are mapped to the empty string. The output is relocated compared to the input. The automatic function that can compute this is the one that changes all symbols to a blank until either a 2 or a blank is reached and then halts. This is done in linear time. The Turing machine that can compute this also works in a similar way. It first erases the zeroes in front until it reaches the 2 and stops the computation. This can also be done in linear time. So $f_{diff} \in AF[n] \cap DTIME[n]$.

[4]The observation that any function in $AF[log(n)]$ maps inputs of length n to outputs having at most length $n+c \log n+c'$ was also observed by Chang Moqiao in her honours year project who had studied a preliminary version of this thesis which did not explicitly use this proof method.

[5]Assume output strings are of a shorter length than the input strings.

f_{diff} actually shows that the difference in starting positions of input and output actually saves us computational time. Because if one has a function f_{same} that wants the same starting positions for the input and output, then it would carry out the computations as per f_{diff} and perform additional steps due to the 'shifting' process. This extra work will render $f_{same} \notin AF[\log n]$ and $f_{same} \notin DTIME[n \log n]$.

Without loss of generality, we assume that the starting position is at the front. If we were to add another step to the output of the function f_{diff} to move the output to the front, we observe that one needs to move the output n positions to the start. There is no real way to reduce the computational time of this movement. At most, the run time can be divided by a constant.

We show next how a one tape Turing machine that initially works by erasing zeroes for example, would require $\Omega(n^2)$ time if one wants to subsequently move the output to the same position of the input.

Consider the function f_{same} now to be the function that takes an input $0^{2m-1}2\{0,1\}^m$ and brings the last third of the input to the first third of the output so that we have the restrictive model. They are both of length m. We use 2 as the separating symbol in the string as before. So this time, we do not want to just simply erase the front symbols.

To see that the computational time is of order $\Omega(n^2)$. Observe that the last third section of $\{0,1\}^m$ has to move over the second third of the input. Both have m number of symbols. Let m' be such that $m < m' < 2m$ and $Cr_{m'}$ be the crossing sequence at the m'-th cell. For each $w \in \{0,1\}^m$, we will look at the 3-tuple $(Cr^s_{m'}, m', l)$, where $Cr^s_{m'}$ is the shortest crossing sequence at the m'-th cell and of length l.

Case 1:

$\exists v, w \in \{0,1\}^m$ with the same $(Cr^s_{m'}, m', l)$. Then a problem arises with the convolution of input and output because there can be switching of the different

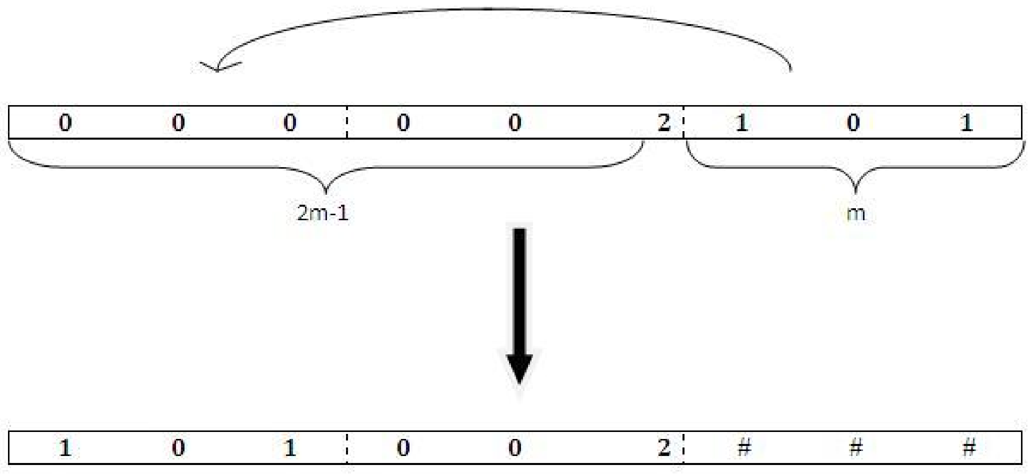


Figure 3.5: f_{same} - Restrictive model requiring input and output starting at the same position

parts and the function is subsequently not well-defined.

Case 2:

Suppose now that $(Cr^s_{m'}, m', l)$ are all different. Then there are m possible values of m', m possible values of l and for L, the longest crossing sequence amongst all values of l for any string $w \in \{0,1\}^m$, c^L possibilities for $Cr^s_{m'}$. Since there are 2^m number of strings w with each their own unique 3-tuple,

$$m \cdot L \cdot c^L \geq 2^m$$

For large m, we can say that $m \leq 2^{\frac{m}{3}}$ and $L \leq 2^{\frac{m}{3}}$. This is because we assume that L is not growing exponentially, otherwise the computation will be at least linear.

$$\begin{aligned} 2^{\frac{2m}{3}} \cdot c^L &\geq 2^m \qquad \text{for some constant } c \\ c^L &\geq 2^{\frac{m}{3}} \\ L \cdot \log c &\geq \frac{m}{3} \\ L &\geq \frac{m}{3 \log c} \end{aligned}$$

So there is indeed a word w that produces crossing sequences with the condition $L \geq \frac{m}{3\log c}$. Hence $\frac{m}{3\log c}$ can be a lower bound for all values of l. Since there are at least m crossing sequences satisfying this lower bound, the whole computation has a runtime of at least $\frac{m^2}{3\log c}$.

Since f_{same} is computed in time $\Omega(n^2)$ by a Turing machine, it must be that $f_{same} \notin DTIME[n \log n]$. By Section 3.3 of this chapter, we deduce that $f_{same} \notin AF[\log n]$.

3.4 Non-deterministic Automatic Functions Computation Model

Definition 3.4.1. *The complexity class of $NAF[h(n)]$ is the class of all functions computable by the machine that uses automatic functions to manipulate its data in a non-deterministic manner. The functions within this complexity class can be computed in the order of $O(h(n))$.*

The main difference between NAF and AF is that for NAF, one can make a program with an additional step which guesses the value of a variable. However, whichever the variable guessed, the algorithm has to terminate in $O(h(n))$ steps, where n is the length of the input x and not any variable guessed. At each step, the value of variables are updated with automatic functions. At the end of the computation, the machine halts and accepts, or halts and rejects.

For $NAF[h(n)]$, given the set $\{conv(x, y, z) : x = \text{input}, y = \text{output}, z = \{accept, reject\}\}$,

- F is multivalued and the set $\{conv(\text{input}, \text{output}) : F(\text{input}) \text{ can be output}\}$ is regular.
- For all x, there is a run ending with $y = f(x)$ and $z = accept$.
- If the process terminates with $conv(x, y, accept)$, then $y = f(x)$
- On input $x \in \Sigma^n$, no run takes longer than $h(n)$ steps and $conv(x, y, z)$ is either $conv(x, f(x), accept)$ or $conv(x, y, reject)$

Example 3.4.2. *This is the same as Example 3.1.2 except that now, the non-determinism would guess a satisfying assignment and the algorithm would verify it. We need to add that G can take different values.*

We can iterate G and after $\log n$ iterations or when G reaches a fixed point of the form $(x, y, accept)$ or $(x, y, reject)$. In line number form, there are 2 special line numbers, one for accept, one for reject corresponding to 2 possible fixed points with halt, where the machine replaces the variable z with *accept* or *reject*.

3.4.1 $AF[1] = NAF[1]$

This result follows from Theorem 2.1.3.

For non-determistic computations, if there are constantly many steps, one can guess for each of these steps, the intermediate results in the variable, that is, make an existential quantifier and then verify that it matches with that of a determistic computation. All computations are accepted, so everything runs in a consistent manner. Hence, for any constant depth computation, they can be captured by automatic functions. This permits a first order definition of the outcome. So if one can non-deterministically guess the outcome, one can also do it without non-determinism.

This means that for constant level, non-determinism does not give something more than one would have with just a deterministic machine. On the other hand, we will see that for logarithmic levels, the non-determinism permits one to compute things that a deterministic model cannot do.

3.4.2 $AF[\log n] \subset NAF[\log n]$

To show that the complexity class of $AF[\log n]$ is a proper subset of $NAF[\log n]$, we will show that there exists a function that can be computed in $NAF[\log n]$ which cannot be computed in time $O(\log n)$ by the AF model.

Proof. Consider the function $f_{dl}(x) = 1^{2|x|}$. We will show that $f_{dl}(x) \in NAF[\log n] \setminus AF[\log n]$. Suppose for example that the input is

$$11111111$$

so $|x| = 8$. The machine now guesses non-deterministically the number of 1's there are in the input and gives the following

$$11111111\dot{}11111111$$

The next few steps taken by the machine would be the deterministic verification of the individual steps of the process that results in the *Accept* or *Reject* of the computation. We start at 11111111˙11111111, whereby the machine will first convert every $(2k+1)$-th digit 1, $k \in \mathbb{N}$ to a zero. Then it compares the parity of 1s before and after the the dot ˙

$$\text{If} \begin{pmatrix} \text{number of 1s before the dot;} \\ \text{number of 1s after the dot} \end{pmatrix} = \begin{cases} \binom{\text{even and } \geq 2;}{\text{even and } \geq 2} & \text{, computation continues} \\ \binom{\text{odd and } \geq 1;}{\text{odd and } \geq 1} & \text{, computation continues} \\ \binom{\text{zero;}}{\text{zero}} & \text{, remove the dot } \dot{} \\ \text{otherwise} & \text{, } \textit{Reject} \end{cases}$$

The process continues with every subsequent line until either a *Reject* is reached or the machine sees

$$00000000\dot{}\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup^{6}$$

It then checks if every $\sqcup$ is a 0 and if so, removes the dot $\dot{}$ and *Accept*.

In the example given, this would be observed

$$11111111\dot{}11111111$$
$$01010101\dot{}01010101$$
$$00010001\dot{}00010001$$
$$00000001\dot{}00000001$$
$$00000000\dot{}00000000$$
$$0000000000000000$$

Accept

Suppose that the input is

$$11111$$

and that the machines guesses wrongly. One would observe for example

$$11111\dot{}11111111$$
$$01010\dot{}10101010$$
$$00010\dot{}00100010$$

Reject

In this case it is the parity of the number of 1s that do not match, causing the rejection of the computation. This comes from the fact that the guessed number of digits was wrong. In fact, for any non-deterministic incorrect guess that the machine makes, it would only take at most $\log n$ steps for the machine to reach a

[6]We are not concerned at this point what symbols are at each $\sqcup$.

Reject state.

Hence the function $f_{dl}(x) = 1^{2|x|}$ is indeed in the class $NAF[\log n]$. And from section 3.3, this same function would not be computable by the model of automatic functions AF in time $O(\log n)$. □

3.4.3 $NAF[\log n]$ and $DTIME[n \log n]$

We now compare the 2 classes $NAF[\log n]$ and $DTIME[n \log n]$. We will show that one of them is not properly contained in the other.

If we have a NAF algorithm, we can simulate it with a non-deterministic Turing machine NM because each step of NM is recognised by an NFA as to whether the step is permitted by an automatic relation. For each step, it goes $\log n$ times from the left end to the right end and back. Hence the length of its crossing sequence has a depth of $\log n$, but this is indeed too short. We need a longer crossing sequence to realise the function we want to compute.

Example 3.4.3. *If x is an input that has length 2^m for some $m \in \mathbb{N}$, then the output $f(x) = 1^{4m}$, else, $f(x) = \epsilon$. This is in $NAF[\log n]$ but not in $DTIME[n \log n]$.*

The algorithm is as follows.

1. $z_1 = 1^{|x|}$
2. Guess for z_2 a value from $\{1\}^*$.
3. If $z_1 \in \{0\}^*1$ and $z_2 \in \{0\}^*1$, then go to 10.
4. If $z_1 \in \{0\}^*1$ and $z_2 \notin \{0\}^*1$, then go to 14.
5. If $z_1 \in \{0,1\}^*\{0\}^+ \cup \{\epsilon\}$, then go to 12.
6. If $z_2 \in \{0,1\}^*\{0\}^+ \cup \{\epsilon\}$, then go to 14.
7. Convert all first 1s in z_1 to 0, keeping all second 1s.
8. Convert all first, second and third 1s in z_2 to 0, keeping all fourth 1s.

9. Go to 3.
10. Let $y = 1^{|z_2|}$.
11. Halt and *Accept*.
12. Let $y = \epsilon$.
13. Halt and *Accept*.
14. Halt and *Reject*.

For this algorithm, the input at every step keeps every $2k$-th position 1, $1 \leq k \leq \frac{n}{2}$. The output keeps every $4j$-th position 1, $1 \leq j \leq \frac{n^2}{4}$. This goes on until the output reaches the form 0^*1. If this does not happen, the output is rejected. If the input reaches 0^*10^+, then the output would be ϵ. This algorithm is done in $\log n$ time and its non-determinism is seen at step 2 where the guessing takes place.

Example 3.4.4. *Suppose we have an input of length $n = 4^m + r$ with $r < 3.4^m$. Consider the function that takes each i-th symbol and swaps it with the $(n-i+1)$-th symbol. This is in $DTIME[n \log n]$ but not in $NAF[\log n]$.*

We can construct a Turing machine that uses a larger alphabet than the input alphabet. This permits the convolution of symbols of the tape and symbols of the counter. So while keeping the symbol of the input on the first entry of the convolution, it can use the second entry to count the number of symbols there are. It counts with a counter with modulo base 4 to get the number m. It then moves back to the beginning and since the counter has only $\log n$ digits, it takes $O(n \log n)$ to move to the front. For the output, the Turing machine counts in modulo base 2 instead and outputs 2^m symbols. This can also be marked off in $O(n \log n)$. The head then moves to the end of the word and erases every thing up to the 2^m-th symbol.

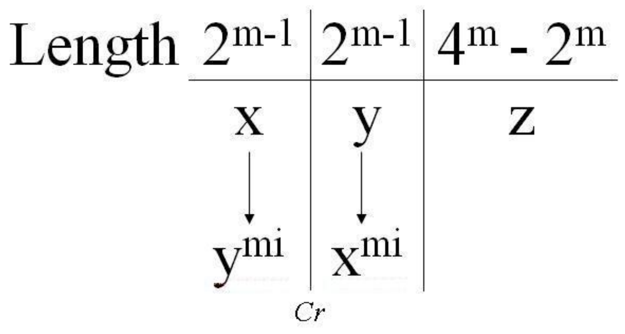


Figure 3.6: Input xyz gets mapped to output $y^{\mathrm{mi}}x^{\mathrm{mi}}$

This cannot be done in $NAF(\log n)$. Suppose we consider the output as 2 sections of length 2^{m-1} and the crossing sequence in the middle Cr (see Figure 3.5), which we get upon doing the automatic function. It has at each non-deterministic step, verifies with an automaton and goes back. Since we know

$$n < 4^m.4$$

$$\log n < 2m + 2$$

For some constant c, Cr has a length of

$$c \cdot \log n = m \cdot 2c$$

The section z of the input is erased, leaving only 2^{m-1} possibilities for characters on each half of the output string and hence $2^{2^{m-1}}$ many possible outputs that must go over Cr when the inverting is done. However there are only $\tilde{c}^{O(m)}$ many crossing sequences for some constant $\tilde{c}$ and for big m,

$$2^{2^{m-1}} > \tilde{c}^{O(m)}$$

So indeed, the number of possible outputs will be more than the number of crossing sequences the longer the input.

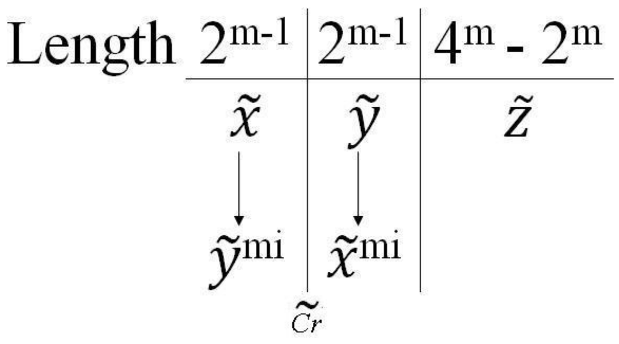


Figure 3.7: Input $\tilde{x}\tilde{y}\tilde{z}$ gets mapped to output $\tilde{y}^{\mathrm{mi}}\tilde{x}^{\mathrm{mi}}$

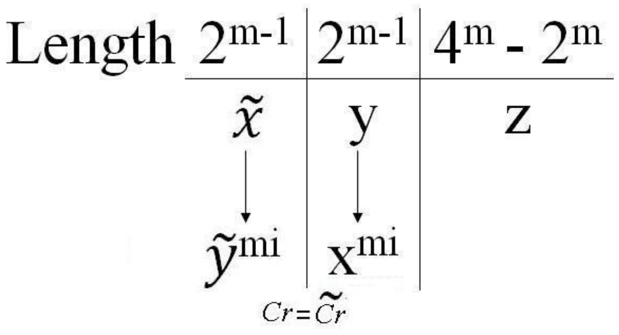


Figure 3.8: Input $\tilde{x}yz$ gets mapped to output $\tilde{y}^{\mathrm{mi}}x^{\mathrm{mi}}$

Suppose there is another input $\tilde{x}\tilde{y}\tilde{z}$ which outputs $\tilde{y}^{\mathrm{mi}}\tilde{x}^{\mathrm{mi}}$ (see Figure 3.6). If $Cr = \tilde{Cr}$, we have 2 different inputs with the same crossing sequence. Then we can have $\tilde{x}yz$ giving an output $\tilde{y}^{\mathrm{mi}}x^{\mathrm{mi}}$ and this will be an accepting computation also (see Figure 3.7).

This means that we can exchange the first half and keep the second half of the word for this particular crossing sequence Cr. But by assumption, $xy \neq \tilde{x}\tilde{y}$ implies $\tilde{x}y \neq x\tilde{y}$. This computation which is accepted is producing an incorrect output. Hence this computation although accepted, is not doing what its supposed to do and hence is not in $NAF[\log n]$.

This means that the crossing sequence in the middle must have a length of order greater than $O(\log n)$. Hence $NAF[\log n]$ is not capturing everything that can be done by $DTIME[n \log n]$.

We have seen a function that can be done in $NAF[\log n]$ but not in $DTIME[n \log n]$ in Example 3.4.3 and vice versa in Example 3.4.4. Hence the two models are not comparable.

Remark 3.4.5. *We also see that from Section 3.3 and Section 3.4.2 that* $AF[\log n] \subset NAF[\log n] \cap DTIME[n \log n]$.

3.5 Some Open Questions

A lot of functions show how we can make outputs of certain lengths in one model but not in the other model. However, we can add one more restriction in trying to study and compare between different models of computation that require the input and output to both have the same lengths.

- Is there $f \in NAF[\log n] \setminus AF[\log n]$ with $\forall x, |f(x)| = |x|$?

- Is there $f \in NAF[\log n] \setminus DTIME[n \log n]$ with $\forall x, |f(x)| = |x|$?

Bibliography

[1] Sanjeev Arora, Boaz Barak. *Computational Complexity: A Modern Approach*, Cambridge University Press, 2009.

[2] Taylor L. Booth. *Sequential Machines and Automata Theory*, John Wiley and Sons, Inc., New York, 1967.

[3] John Case, Sanjay Jain, Trong Dao Le, Yuh Shin Ong, Pavel Semukhin and Frank Stephan. Automatic learning of subclasses of pattern languages, *Language and Automata Theory and Applications*, LATA 2011, Taragona, Spain, May 2011, Proceedings. Springer LNCS 6638:192–203, 2011.

[4] John Case, Sanjay Jain, Samuel Seah and Frank Stephan. *Automatic Functions, Linear Time and Learning*, 2012.

[5] Juris Hartmanis. Computational complexity of one-tape Turing machine computations. *Journal of the Association of Computing Machinery* 15:411–418, 1968.

[6] Juris Hartmanis and Richard E. Stearns. On the computational complexity of algorithms, *Trans. Amer. Math. Soc.*, 117:285306, 1965.

[7] Fred C. Hennie. Crossing sequences and off-line Turing machine computations. Sixth Annual Symposium on *Switching Circuit Theory and Logical Design*, pages 168–172, 1965.

[8] Bernard R. Hodgson, Décidabilité par automate fini, *Annales des sciences mathématiques du Québec*, 7(1):39–57, 1983.

[9] Bakhadyr Khoussainov and Anil Nerode. Automatic presentations of structures. *Logical and Computational Complexity*, (International Workshop LCC 1994). Springer LNCS 960:367–392, 1995.

[10] Bakhadyr Khoussainov, Mia Minnes. *Three Lectures on Automatic Structures*, , 2007.

[11] Bakhadyr Khoussainov, Sasha Rubin. Finite Automata and Isomorphism Types *CDMTCS Research Report Series*, 128:1–4, 2000.

[12] Piergiorgio Odifreddi. *Classical Recursion Theory*, Volume II. Studies in Logic and the Foundations of Mathematics, 143. Elsevier, 1999.

[13] Sasha Rubin. Automata presenting structures: a survey of the finite string case. *The Bulletin of Symbolic Logic*, 14:169–209, 2008.

[14] Boris A. Trakhtenbrot. Turing computations with logarithmic delay. *Algebra i Logika*, 3:33-48, 1964.