

# A Different Proof of the Time Hierarchy Theorem

András Z. Salamon

School of Computer Science  
University of St Andrews  
Scotland, UK

Andras.Salamon@st-andrews.ac.uk

Michael Wehar

Computer Science Department  
Bryn Mawr College  
PA, USA

mwehar@brynmawr.edu

We reinvestigate the classical time hierarchy theorem in computational complexity theory from Hartmanis and Stearns (1965). We provide a seemingly different proof of the time hierarchy theorem by defining a slow growing computable function that is infinitely often smaller than any  $t(n)$ -time computable superconstant function. The time complexity lower bound is obtained by considering whether the function itself is  $t(n)$ -time computable.

## 1 Introduction

### 1.1 Background

The classical time hierarchy theorem was introduced in Hartmanis and Stearns (1965) [6]. It can be expressed as  $\text{DTIME}(t_1(n)) \subsetneq \text{DTIME}((t_2(n))^2)$  for all well-behaved time bounds  $t_1(n)$  and  $t_2(n)$  such that  $t_1(n)$  is  $o(t_2(n))$ . In 1966, it was improved by Hennie and Stearns using tape-reduction [7]. In particular, we have  $\text{DTIME}(t_1(n)) \subsetneq \text{DTIME}(t_2(n) \log(t_2(n)))$  when  $t_1(n)$  is  $o(t_2(n))$ . Later in 1982, tightness was achieved by Fürer when restricted to a fixed number of tapes [4]. That is, we have  $\text{DTIME}(t_1(n)) \subsetneq \text{DTIME}(t_2(n))$  when  $t_1(n)$  is  $o(t_2(n))$ . Achieving tightness for an unbounded number of tapes seems difficult, as that appears to require an improvement to the tape reduction theorem which currently introduces a  $\log(t_2(n))$  factor in the runtime upper bound.

The proof of the time hierarchy theorem has two components. The first part is universal simulation. Essentially, one shows that the acceptance language  $A_{TM} = \{i\#x \mid M_i \text{ accepts } x \text{ in } t_2(|x|) \text{ steps or less}\}$  is decidable in  $O(t_2(n) \log(t_2(n)))$  steps. In other words,  $A_{TM} \in \text{DTIME}(t_2(n) \log(t_2(n)))$ . The second part is diagonalization. Consider the diagonal language  $D_{TM} = \{i \mid M_i \text{ does not accept } i \text{ in } t_2(|i|) \text{ steps or less}\}$ . Suppose for sake of contradiction that  $D_{TM} \in \text{DTIME}(t_1(n))$ . If it is, then there is a machine  $M_i$  that decides  $D_{TM}$  in  $o(t_2(n))$  time. This leads to a contradiction because  $i \in D_{TM} \Leftrightarrow i \notin D_{TM}$ . It follows that  $D_{TM} \in \text{DTIME}(t_2(n) \log(t_2(n)))$  and  $D_{TM} \notin \text{DTIME}(t_1(n))$ .

### 1.2 Terminology and Notation

In this work, we adopt the following terminology and notation. Much of the terminology is either borrowed from or a slight variation to that from [8]. In addition, all logarithms will be base 2.

We index Turing machines using a form of Gödel numbering [5, 8]. Each Turing machine is represented as  $M_i$  for some number  $i$  such that we can efficiently compute from the number  $i$  the states and transitions of the machine  $M_i$ . Turing machines decide languages while Turing transducers compute functions. The Turing machines that we consider in the following all have a fixed number of tapes (specifically, two tapes). In particular, each Turing machine has a designated input tape where the input (represented as a string) is initially written. Each machine also has a work tape. Both tapes are two-way, use a binary alphabet, have a leftmost initial cell, and extend infinitely to the right. Each Turing

transducer similarly has a designated input tape and an output tape. The string on the output tape at the moment when the machine halts represents the transducer's output.

A language  $L$  is  $t(n)$ -time decidable if there exists a Turing machine that decides  $L$  and always halts in at most  $O(t(n))$  steps where  $n$  is the input length. More simply, we write  $L \in \text{DTIME}(t(n))$ . A function  $f : \mathbb{N} \rightarrow \mathbb{N}$  is said to be  $t(n)$ -time computable if there exists a Turing transducer that computes  $f(n)$  in at most  $t(n)$  steps for any given input number  $n$ . By input number we mean the number represented by the input string. We will represent input numbers in unary for convenience, although we suggest that the results in this work could be adapted for binary representation. Furthermore, a function  $t(n)$  is said to be time-constructible if it is  $O(t(n))$ -time computable. In addition, we consider the following properties of functions. We say that  $f(n)$  is superconstant if  $f(n)$  is  $\omega(1)$ . We say that  $f(n)$  is non-decreasing if for all  $n_1, n_2 \in \mathbb{N}$ , we have  $n_1 < n_2$  implies  $f(n_1) \leq f(n_2)$ . We say that  $f(n)$  is superadditive if for all  $n_1, n_2 \in \mathbb{N}$ , we have  $f(n_1 + n_2) \geq f(n_1) + f(n_2)$  [1]. We say that  $t(n)$  is a *well-behaved* time bound if it is non-decreasing, superadditive, time-constructible, and  $t(n) \geq n$  for all  $n \in \mathbb{N}$ .

### 1.3 Fixed Points and Anti-Fixed Points

The time hierarchy theorem has a strong connection with fixed points. In particular, in [11, Theorem 6.20], it was shown how the diagonalization argument from the classic proof of the time hierarchy theorem can be reframed as a fixed point argument. Another relationship that we have observed is that the time hierarchy theorem itself, when expressed in terms of function classes instead of language classes, can be viewed as a result about anti-fixed points. We present this observation in Theorem 1.2 below.

We proceed by introducing additional terminology and notation. Let  $\text{FDTIME}(t(n))$  denote the class of functions  $f : \mathbb{N} \rightarrow \mathbb{N}$  such that  $f(n)$  is  $O(t(n))$ -time computable. Consider the set of functions  $\mathbb{F} = \{f \mid f : \mathbb{N} \rightarrow \mathbb{N}\}$  and the function  $\text{compose} : \mathcal{P}(\mathbb{F}) \rightarrow \mathcal{P}(\mathbb{F})$  such that for all  $X \subseteq \mathbb{F}$ ,

$$\text{compose}(X) = \{f \circ g \mid f \in \text{FDTIME}(n^2) \text{ and } g \in X\}.$$

**Lemma 1.1.** *For any well-behaved time bound  $t(n)$ ,  $\text{compose}(\text{FDTIME}(t(n))) = \text{FDTIME}((t(n))^2)$ .*

*Proof.* First, let  $f \circ g \in \text{compose}(\text{FDTIME}(t(n)))$  be given. Observe that  $g(n)$  is computed in  $O(t(n))$  time, the output of  $g(n)$  is  $O(t(n))$ , and  $f$  evaluated on an input number that is  $O(t(n))$  takes  $O((t(n))^2)$  time. Therefore, we have that  $f \circ g$  is  $O((t(n))^2)$ -time computable. Second, let  $h \in \text{FDTIME}((t(n))^2)$  be given. We will represent  $h$  as  $f \circ g$  such that  $g \in \text{FDTIME}(t(n))$  and  $f \in \text{FDTIME}(n^2)$ . We define  $g(n)$  in such a way that  $g(n)$  is  $\Theta(t(n))$  and we can efficiently recover  $n$  given  $g(n)$ . We accomplish this by setting  $g(n) = 2^{\lfloor \log(t(n)) \rfloor + 1} + n$ . We define  $f$  so that it extracts  $n$  from an input  $g(n)$  and then evaluates  $h(n)$ . We extract  $n$  by getting the binary representation of  $g(n)$  and then removing the leading 1 bit.  $\square$

We consider a variation of the time hierarchy theorem for computable functions rather than decidable languages. In particular, notice<sup>1</sup> that  $\text{FDTIME}(t(n)) \subsetneq \text{FDTIME}((t(n))^2)$  for all well-behaved time bounds  $t(n)$ . By combining with Lemma 1.1, we have the following anti-fixed point result.

**Theorem 1.2** (Anti-Fixed Points). *For any well-behaved time bound  $t(n)$ ,*

$$\text{FDTIME}(t(n)) \neq \text{compose}(\text{FDTIME}(t(n))).$$

*Proof.* By combining the time hierarchy theorem for function classes with Lemma 1.1, we have that

$$\text{FDTIME}(t(n)) \subsetneq \text{FDTIME}((t(n))^2) = \text{compose}(\text{FDTIME}(t(n))).$$

It follows that all  $\text{FDTIME}(t(n))$  classes are anti-fixed points for the function  $\text{compose}$ .  $\square$

<sup>1</sup>Technically, this holds trivially because any function that is  $\Theta((t(n))^2)$  cannot be computed in  $O(t(n))$  time.

## 1.4 Slow Growing Functions

In this subsection, we introduce a template such that for any well-behaved time bound  $t(n)$ , we have a slow growing function  $SF_t(n)$ . We will see that these functions are so slow growing that  $SF_t(n)$  is infinitely often smaller than any superconstant  $o(t(n))$ -time computable function. In the following, we denote by  $\text{time}(M_i)$  the number of steps taken before halting by a Turing machine  $M_i$  when given an empty input. If it does not halt, then  $\text{time}(M_i)$  takes the value  $\infty$ . Now, we proceed with the definition.

**Definition 1.3.** For any well-behaved time bound  $t(n)$ , we define the function  $SF_t : \mathbb{N} \rightarrow \mathbb{N}$  such that

$$SF_t(n) = \min\{i \in \mathbb{N} \mid n \leq \text{time}(M_i) \leq t(n)\}.$$

Next, we prove that these functions are growing. That is, we observe that  $SF_t(n)$  is superconstant.

**Proposition 1.4.** For any well-behaved time bound  $t(n)$ , we have that  $SF_t(n)$  is  $\omega(1)$ .

*Proof.* Let  $k \in \mathbb{N}$  be given. Consider  $m = \max\{\text{time}(M_i) \mid i \leq k \text{ and } \text{time}(M_i) < \infty\} + 1$ . Hence, there cannot exist a Turing machine  $M_i$  with  $i \leq k$  such that  $m \leq \text{time}(M_i) < \infty$ . Therefore,  $SF_t(m) > k$ . Since  $k$  was an arbitrary natural number, the proposition holds.  $\square$

**Remark 1.5.** The  $SF_t(n)$  functions grow, but do so slowly. An upper bound of  $O(\text{poly}(\log(n)))$  holds when  $t(n) \geq 2 \cdot n$ . In particular, given  $n \in \mathbb{N}$ , there exists  $2^m \in \mathbb{N}$  such that  $n \leq 2^m \leq 2 \cdot n \leq t(n)$ . By constructing small machines that count to fixed numbers [10, 3, 2], it can be shown that there exists an  $O(\frac{\log(m)}{\log \log(m)})$  state Turing machine  $M_i$  such that  $n \leq \text{time}(M_i) \leq 2^m$ . Since  $M_i$  has  $O(\frac{\log(m)}{\log \log(m)})$  states,  $i$  must be  $O(\text{poly}(m))$  which is  $O(\text{poly}(\log(n)))$ . It follows that  $SF_t(n)$  is  $O(\text{poly}(\log(n)))$ . In future work, we plan to more carefully investigate and prove bounds for the  $SF_t(n)$  functions.

## 2 Reproducing the Time Hierarchy Theorem

In this section, we provide our proof of the time hierarchy theorem using the slow growing  $SF_t(n)$  functions. We start by proving the following proposition which states that these functions are computable. This proposition is similar to the universal simulation part of the classic proof of the time hierarchy theorem. Note that because we focus on Turing machines with a fixed number of tapes (specifically, two tapes), the following proof works without a  $\log(t(n))$  factor.

**Proposition 2.1.** For any well-behaved time bound  $t(n)$ , we have that  $SF_t(n)$  is  $t'(n)$ -time computable where  $t'(n)$  is  $O(t(n) \cdot SF_t(n) \cdot \log(SF_t(n)))$ .

*Proof.* Start with a value  $i = 0$  and simulate the two-tape Turing machine  $M_i$  for at most  $t(n)$  steps on an empty input. Then, increment  $i$  and repeat. If  $i$  is found such that  $n \leq \text{time}(M_i) \leq t(n)$ , then  $SF_t(n) = i$ . In summary, we simulate  $SF_t(n) + 1$  many machines each for at most  $t(n)$  steps where each simulation takes  $O(t(n) \cdot \log(SF_t(n)))$  time. Therefore, the total runtime is  $O(t(n) \cdot SF_t(n) \cdot \log(SF_t(n)))$ .

The reason that each simulation has a factor of  $\log(SF_t(n))$  is because we represent  $M_i$  as a bit string  $s$  of length at most  $O(\log(SF_t(n)))$  and we initially write down  $t(n)$  equally-spaced copies of  $s$  across the tape so that we always have quick access to look up the next transition to take during the simulation.  $\square$

The following lemma is our main result about the slow growing  $SF_t(n)$  functions and is of independent interest. We will apply this lemma to prove our version of the time hierarchy theorem.

**Lemma 2.2.** For any well-behaved time bound  $t(n)$  and  $o(t(n))$ -time computable superconstant function  $f(n)$ , we have that  $SF_t(n) < f(n)$  for infinitely many input values  $n \in \mathbb{N}$ .

*Proof.* Let  $T$  denote a Turing transducer such that  $T$  computes  $f(n)$  in at most  $\frac{t(n)}{2}$  steps. For each  $d \in \mathbb{N}$ , we define a Turing machine  $\mathcal{M}(d)$  as follows. The machine  $\mathcal{M}(d)$  starts with a value  $i = 0$  and uses  $T$  to evaluate  $f(2^i)$ . If we have  $2^d < f(2^i)$ , then halt. Otherwise, increment  $i$  and repeat.

The machine  $\mathcal{M}(d)$  will eventually halt because  $f(n)$  is  $\omega(1)$  which implies that there will eventually be a value  $k \in \mathbb{N}$  such that  $2^d < f(2^k)$ . The total runtime<sup>2</sup> will be at most

$$\frac{\sum_{i=0}^k t(2^i)}{2} \leq \frac{t(2^k) + t(\sum_{i=0}^{k-1} 2^i)}{2} = \frac{t(2^k) + t(2^k - 1)}{2} \leq t(2^k).$$

This works because of the superadditive property  $t(n_1) + t(n_2) \leq t(n_1 + n_2)$  which holds since  $t(n)$  is a well-behaved time bound. In addition, the runtime will be at least  $2^k$  from evaluating  $f(2^k)$  since every input symbol from the unary input number  $2^k$  must be read.

Also, the machine  $\mathcal{M}(d)$  is straightforward to construct given the Turing transducer  $T$ . If  $T$  has  $m$  states, then there exists a constant  $C$  such that  $\mathcal{M}(d)$  has at most  $O(\frac{\log(d)}{\log \log(d)}) + m + C$  states. The first term comes from counting the number  $d$  as in [10, 3, 2]. Therefore, since  $m + C$  is a constant, there is  $j \leq p(d)$  for a polynomial  $p$ , independent of  $d$ , such that  $\mathcal{M}(d) = M_j$ . Hence,  $SF_t(2^k) \leq j \leq p(d)$ .

Now, we can pick  $d$  sufficiently large so that  $SF_t(2^k) \leq p(d) < 2^d < f(2^k)$ . It follows that we have a single input  $n = 2^k$  such that  $SF_t(n) < f(n)$ . Next, we pick  $d'$  such that  $f(2^k) < 2^{d'}$ . From there, we get  $k' > k$  such that  $SF_t(2^{k'}) < f(2^{k'})$ . That is, we have that  $n' = 2^{k'} > n$  such that  $SF_t(n') < f(n')$ . Moreover, we can repeat this process to get infinitely many values  $n \in \mathbb{N}$  such that  $SF_t(n) < f(n)$ .  $\square$

The following theorem is our lower bound result. It applies the preceding lemma with  $f(n) = SF_t(n)$ . This seems to be a form of self-reference resembling diagonalization.

**Theorem 2.3** (Lower Bound). *For any well-behaved time bound  $t(n)$ , we have that  $SF_t(n)$  is not computable in  $o(t(n))$  time.*

*Proof.* Suppose for sake of contradiction that  $SF_t(n)$  is  $o(t(n))$ -time computable. By Proposition 1.4,  $SF_t(n)$  is  $\omega(1)$ . Hence, we can apply Lemma 2.2 with  $f(n) = SF_t(n)$  to get that  $SF_t(n)$  is infinitely often smaller than itself. This is a contradiction because we cannot have  $SF_t(n) < SF_t(n)$  for any  $n \in \mathbb{N}$ .  $\square$

**Remark 2.4.** *Let us examine how Lemma 2.2 is applied to prove Theorem 2.3. We define a function  $G : \mathbb{F} \rightarrow \mathbb{N}$ . That is,  $G$  maps functions to natural numbers. Given an appropriate function  $f : \mathbb{N} \rightarrow \mathbb{N}$ , we define  $G(f) = n$  where  $n$  is the smallest value such that  $SF_t(n) < f(n)$ . Therefore,  $SF_t(G(f)) < f(G(f))$ . Now, consider  $f = SF_t$ . In this case, we get  $SF_t(G(SF_t)) < SF_t(G(SF_t))$  which would be a contradiction.*

Now, we obtain our version of the Time Hierarchy Theorem by representing the function  $SF_t(n)$  as a language. Then, we combine Proposition 2.1 and Theorem 2.3.

**Corollary 2.5** (Time Hierarchy). *For any well-behaved time bounds  $t_1(n)$  and  $t_2(n)$  where  $t_1(n) \cdot SF_{t_2}(n)$  is  $o(t_2(n))$ , we have that  $\text{DTIME}(t_1(n)) \subsetneq \text{DTIME}(t_2(n) \cdot SF_{t_2}(n) \cdot \log(SF_{t_2}(n)))$ .*

*Proof.* By Proposition 2.1, we have that  $SF_{t_2}(n)$  is computable in  $O(t_2(n) \cdot SF_{t_2}(n) \cdot \log(SF_{t_2}(n)))$  time. Also, by Theorem 2.3, we have that  $SF_{t_2}(n)$  is not computable in  $o(t_2(n))$  time. Next, we convert the function  $SF_{t_2}(n)$  into an associated language  $L$  of binary strings that encode ordered pairs of numbers in such a way that an ordered pair  $(n, SF_{t_2}(n)) \in \mathbb{N}^2$  will be encoded as a string of length  $n$ . To do this, we encode both numbers in binary and include a padding to make the first number equal to the encoded string's length. Hence,  $L \in \text{DTIME}(t_2(n) \cdot SF_{t_2}(n) \cdot \log(SF_{t_2}(n)))$ . Also,  $L \notin \text{DTIME}(t_1(n))$  because otherwise,  $SF_{t_2}(n)$  is computable in  $O(t_1(n) \cdot SF_{t_2}(n))$  time which is  $o(t_2(n))$  by searching for the encoded pair in  $L$  with  $n$  as the first number.  $\square$

<sup>2</sup>Any overhead costs can be overcome by picking a transducer that computes  $f(n)$  more efficiently by a constant factor.

### 3 Conclusion and Future Directions

The time hierarchy theorem is a fundamental result in computer science. In Corollary 2.5, we reproduced a variation of this classic theorem. Although the statement of our version offers no real improvement, we have shown a seemingly different approach for proving a time complexity lower bound. Furthermore, we have shown how these slow growing computable functions, denoted by  $SF_t(n)$ , are strongly connected with the complexity classes  $\text{DTIME}(t(n))$ . We plan to expand this work to offer a more intensive study of these functions. In particular, we notice a strong relationship between these functions and the busy beaver functions from [9]. Because of this association, the authors informally refer to the  $SF_t(n)$  functions as the slow sloth functions and continue to explore their seemingly unusual behaviors.

#### Acknowledgments

We greatly appreciate all of the feedback that we have received. We are especially thankful for the helpful and supportive discussions with D. Chistikov, C. Liu, N. Lutz, and C. Xu.

#### References

- [1] Andrew M. Bruckner & E. Ostrow (1962): *Some function classes related to the class of convex functions*. *Pacific Journal of Mathematics* 12, pp. 1203–1215, doi:10.2140/pjm.1962.12.1203.
- [2] Gregory J. Chaitin (1966): *On the Length of Programs for Computing Finite Binary Sequences*. *J. ACM* 13(4), p. 547–569, doi:10.1145/321356.321363.
- [3] Dmitry Chistikov (2014): *Notes on Counting with Finite Machines*. In Venkatesh Raman & S. P. Suresh, editors: *34th International Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS 2014)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 29, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 339–350, doi:10.4230/LIPIcs.FSTTCS.2014.339.
- [4] Martin Fürer (1982): *The tight deterministic time hierarchy*. In: *Symposium on the Theory of Computing*, Association for Computing Machinery, pp. 8–16, doi:10.1145/800070.802172.
- [5] Kurt Gödel (1970): *On formally undecidable propositions of Principia mathematica and related systems I (1931)*. In Jean van Heijenoort, editor: *Frege and Gödel: Two Fundamental Texts in Mathematical Logic*, Harvard University Press, pp. 87–107, doi:10.4159/harvard.9780674864603.c3.
- [6] J. Hartmanis & R. E. Stearns (1965): *On the computational complexity of algorithms*. *Transactions of the American Mathematical Society* 117, pp. 285–306, doi:10.1090/S0002-9947-1965-0170805-7.
- [7] F. C. Hennie & R. E. Stearns (1966): *Two-Tape Simulation of Multitape Turing Machines*. *J. ACM* 13(4), pp. 533–546, doi:10.1145/321356.321362.
- [8] Steve Homer & Alan Selman (2011): *Computability and Complexity Theory*, 2nd edition. Springer, doi:10.1007/978-1-4614-0682-2.
- [9] T. Rado (1962): *On non-computable functions*. *The Bell System Technical Journal* 41(3), pp. 877–884, doi:10.1002/j.1538-7305.1962.tb00480.x.
- [10] Michael Wehar (2013): *Fixed Parameter Inductive Inference*. Manuscript.
- [11] N.S. Yanofsky (2022): *Theoretical Computer Science for the Working Category Theorist*. *Elements in Applied Category Theory*, Cambridge University Press, doi:10.1017/9781108872348.