Texts in Computational Complexity: More Resources, More Power?

Oded Goldreich
Department of Computer Science and Applied Mathematics
Weizmann Institute of Science, Rehovot, Israel.

December 5, 2005

More electricity, less toil.

The Israeli Electricity Company, 1960's

A brief introduction

Is it indeed the case that the more resources one has, the more one can achieve? The answer may seem obvious, but the obvious answer (of yes) actually presumes that the worker knows how much resources are at his/her disposal. In this case, when allocated more resources, the worker (or computation) can indeed achieve more. But otherwise, nothing may be gained by adding resources.

In the context of computational complexity, an algorithm knows the amount of resources that it is allocated if it can determine this amount without exceeding the corresponding resources. This condition is satisfies in all natural cases, but it may not hold in general. The latter fact should not be that surprising: we already know that some functions are not computable and if these functions are used to determine resources then the algorithm may be in trouble. Needless to say, this discussion requires some formalization, which is will provided below.

When using "nice" functions to determine the algorithm's resources, it is indeed the case that more resources allow for more tasks to be performed. However, when "ugly" functions are used for the same purpose, increasing the resources may have no effect. By nice functions we mean functions that can be computed without exceeding the amount of resources that they specify (e.g., $t(n) = n^2$ or $t(n) = 2^n$). Naturally, "ugly" functions do not allow to present themselves in such nice forms.

The forgoing discussion refers to a uniform model of computation and to (natural) resources such as time and space complexities. Thus, we get results asserting, for example, that there are functions computable in cubic-time but not in quadratic-time. In case of non-uniform models of computation, the issue of "nicety" does not arise, and it is easy to establish separations between levels of circuit complexity that differ by any unbounded amount.

Results that *separate* the class of problems solvable within one resource bound from the class of problems solvable within a larger resource bound are called hierarchy theorems. Results that indicate the non-existence of such separations, hence indicating a "gap" in the growth of computing power (or a "gap" in the existence of algorithms that utilize the added resources), are called gap theorems. A somewhat related phenomenon, called speed-up theorems, refers to the inability to define the complexity of some problems.

Caveat: Uniform complexity classes based on specific resource bounds (e.g., cubic-time) are model dependent. Furthermore, the tightness of separation results (i.e., how much more time is required to solve an additional computational problem) is also model dependent. Still the existence of such separations is a phenomenon common to all reasonable and general models of computation (as referred to in the Cobham-Edmonds Thesis). In the following presentation, we will explicitly differentiate model-specific effects from generic ones.

Organization: We will first demonstrate the "more resources yield more power" phenomenon in the context of non-uniform complexity. In this case the issue of "knowing" the amount of resources allocated to the computing device does not arise, because each device is tailored to the amount of resources allowed for the input length that it handles (see Section 1).

1 Non-uniform complexity hierarchies

The model of machines that use advice offers a very convenient setting for separation results. We refer specifically, to classes of the form $\mathcal{P}/\ell$, where $\ell : \mathbb{N} \to \mathbb{N}$ is an arbitrary function. Recall that every Boolean function is in $\mathcal{P}/2^n$, by virtue of a trivial algorithm that is given as advice the truth-table of the function restricted to the relevant input length. An analogous algorithm underlies the following separation result.

Theorem 1 *For any two functions $\ell', \delta : \mathbb{N} \to \mathbb{N}$ such that $\ell'(n) + \delta(n) \leq 2^n$ and δ is unbounded, it holds that $\mathcal{P}/\ell'$ is strictly contained in $\mathcal{P}/(\ell' + \delta)$.*

Proof: Let $\ell \stackrel{\rm def}{=} \ell' + \delta$, and consider the algorithm A that given advice $a_n \in \{0,1\}^{\ell(n)}$ and input $i \in \{1, ..., 2^n\}$ (viewed as an n-bit long string), outputs the $i^{\rm th}$ bit of a_n if $i \leq |a_n|$ and zero otherwise. Clearly, for any $\overline{a} = (a_n)_{n\in\mathbb{N}}$ such that $|a_n| = \ell(n)$, it holds that the function $f_{\overline{a}}(x) \stackrel{\rm def}{=} A(a_{|x|}, x)$ is in $\mathcal{P}/\ell$. Furthermore, different sequences $\overline{a}$ yield different functions $f_{\overline{a}}$. We claim that some of these functions $f_{\overline{a}}$ are not in $\mathcal{P}/\ell$, thus obtaining a separation.

The claim is proved by considering all possible (polynomial-time) algorithms A' and all possible sequences $\overline{a}' = (a'_n)_{n\in\mathbb{N}}$ such that $|a'_n| = \ell'(n)$. Fixing any algorithm A', we consider the number of n-bit long functions that are correctly computed by $A'(a'_n, \cdot)$. Clearly, the number of these functions is at most $2^{\ell'(n)}$, and thus A' may account for at most $2^{-\delta(n)}$ fraction of the functions $f_{\overline{a}}$ (even when restricted to n-bit strings). This consideration holds for every n and every possible A', and thus the measure of the set of functions that are computable by algorithms that take advice of length ℓ is zero.[1] ■

A somewhat less tight bound can be obtained by using the model of Boolean circuits. In this case some slackness is needed in order to account for the gap between the upper and lower bounds regarding the number of Boolean functions over $\{0,1\}^n$ that are computed by Boolean circuits of size $s < 2^n$. Specifically (see Exercise 10), an obvious lower-bound on this number is $2^{s/O(\log s)}$ whereas an obvious upper-bound is $\binom{s^2}{s} \approx 2^{2s\log_2 s}$. (Compare these to the lower-bound 2^s, and the upper-bound $2^{s+((\delta(n)-2)/2)}$ used in the proof of Theorem 1.)

[1]It suffices to show that this measure is strictly less than one. This is easily done by considering, for every algorithm A', the performance of A' on inputs of length n such that $\delta(n) > 2|\langle A' \rangle| + 2$.

2 Time Hierarchies and Gaps

In this section we show that in the "natural cases" increasing time-complexity allows for more problems to be solved, whereas in "pathological cases" it may happen that even a dramatic increase in the time-complexity provides no additional computing power. As hinted in the introduction, the "natural cases" correspond to time bounds that can be determined by the algorithm itself within the specified time complexity.

2.1 Time Hierarchies

Note that the non-uniform computing devices considered in the previous section were explicitly given the relevant resource bounds. Actually, they were given the resources themselves and did not need to monitor their usage of these resources. In contrast, when designing algorithms of time complexity $t : \mathbb{N} \to \mathbb{N}$, we need to make sure that the algorithm does not exceed the time bound. Furthermore, when invoked on input x, the algorithm is not given the time bound $t(|x|)$ explicitly, and a reasonable design methodology (for dealing with generic algorithms as we will in the proof of the following separation results) is to let the algorithm compute this bound (i.e., $t(|x|)$). This, in turn, requires to read the entire input (see Exercise 11) as well as to compute $t(n)$ using $O(t(n))$ (or so) time. The latter requirement motivates the following definition (which is related to the standard definition of "fully time constructibility" (cf. [4, Sec. 12.3])).

Definition 2 (time constructible functions): *A function $t : \mathbb{N} \to \mathbb{N}$ is called* time constructible *if there exists an algorithm that on input n outputs $t(n)$ using at most $t(n)$ steps.*

Equivalently, we may require that the mapping $1^n \mapsto t(n)$ be computable within time complexity t. We warn that the foregoing definition is model dependent; however, typically nice functions are computable even faster (e.g., in $\text{poly}(\log t(n))$ steps), in which case the model-dependency is irrelevant (for reasonable and general models of computation, as referred to in the Cobham-Edmonds Thesis). For example, in any reasonable and general model, functions like $t_1(n) = n^2$, $t_2(n) = 2^n$, and $t_3(n) = 2^{2^n}$ are computable in $\text{poly}(\log t_i(n))$ steps.

Likewise, for a fixed model of computation (to be understood from the context) and for any function $t : \mathbb{N} \to \mathbb{N}$, we *denote by* $\textsc{Dtime}(t)$ *the class of decision problems that are solvable in time complexity t.*

2.1.1 The Time Hierarchy Theorem

In the following theorem, we refer to the model of two-tape Turing machines. In this case we obtain quite a tight hierarchy, when referring to the relation between t_1 and t_2. We stress that using the Cobham-Edmonds Thesis, this results yields (possibly less tight) hierarchy theorems for any reasonable and general model of computation.

> **Teaching note:** The standard statement of the following result asserts that *for any time computable function t_2 and every function t_1 such that $t_2 = \omega(t_1 \log t_1)$ and $t_1(n) > n$ it holds that* $\textsc{Dtime}(t_1)$ *is strictly contained in* $\textsc{Dtime}(t_2)$. We find the proof of the current version more intuitive, and comment on the proof of the standard version after proving the current one.

Theorem 3 (time hierarchy for two-tape Turing machines): *For any time computable function t_1 and every function t_2 such that $t_2(n) \geq (\log t_1(n))^2 \cdot t_1(n)$ and $t_1(n) > n$ it holds that* $\textsc{Dtime}(t_1)$ *is strictly contained in* $\textsc{Dtime}(t_2)$.

As will become clear from the proof, an analogous result holds for any model in which a universal machine can emulate t steps of another machine in $O(t\log t)$ time. Before proving Theorem 3, we derive the following corollary.

Corollary 4 (time hierarchy for any reasonable and general model): *For any reasonable and general model of computation there exists a positive polynomial p such that for any time-computable function t_1 and every function t_2 such that $t_2 > p(t_1)$ and $t_1(n) > n$ it holds that* $\mathrm{D}\textsc{time}(t_1)$ *is strictly contained in* $\mathrm{D}\textsc{time}(t_2)$.

Proof of Corollary 4: Letting $\mathrm{D}\textsc{time}_2$ denote the classes that correspond to two-tape Turing machines, we have $\mathrm{D}\textsc{time}(t_1) \subseteq \mathrm{D}\textsc{time}_2(t_1')$ and $\mathrm{D}\textsc{time}(t_2) \supseteq \mathrm{D}\textsc{time}_2(t_2')$, where $t_1' = \mathrm{poly}(t_1)$ and t_2' is defined such that $t_2(n) = \mathrm{poly}(t_2'(n))$. The latter unspecified polynomials, hereafter denoted p_1 and p_2 respectively, are the ones guaranteed by the Cobham-Edmonds Thesis. Also, the hypothesis that t_1 is time-computable implies that $t_1' = p_1(t_1)$ is time-constructible with respect to the two-tape Turing machine model. Thus, for a suitable choice of the polynomial p, it holds that

$$t_2'(n) = p_2^{-1}(t_2(n)) > p_2^{-1}(p(t_1(n))) > p_2^{-1}(p(p_1^{-1}(t_1'(n)))) > (t_1'(n))^2 .$$

Invoking Theorem 3, we have $\mathrm{D}\textsc{time}_2(t_2') \supset \mathrm{D}\textsc{time}_2(t_1')$, and the corollary follows. ■

Proof of Theorem 3: The idea is to construct a Boolean function f such that all machines having time complexity t_1 fail to compute f. This is done by associating each possible machine M a different input x_M (e.g., $x_M = \langle M\rangle$), and making sure that $f(x_M) \neq M'(x)$, where $M'(x)$ denotes an emulation of $M(x)$ that is suspended after $t_1(|x|)$ steps. Actually, we are going to use a mapping μ of inputs to machines (i.e., $\mu(x_M) = M$), such that each machine is in the range of μ and μ is very easy to compute.

The issue is presenting an algorithm for computing f. This algorithm is straightforward: On input x, it computes $t = t_1(|x|)$, determines the machine $M = \mu(x)$ that corresponds to x (outputting a default value of no such machine exists), emulates $M(x)$ for t steps, and returns the value $1 - M'(x)$. The question is how much time is required for this emulation. We should bear in mind that the time complexity of our algorithm needs to be evaluated in the two-tape Turing machine model, whereas M itself is a two-tape Turing machine. The obvious approach is to implement our algorithm on a three-tape Turing-machine, using two tapes for the emulation itself and another tape for the emulation procedure. Using this approach, each step of M is emulated in $O(|\langle M\rangle|)$ steps (on the three-tape machine). Emulating t' steps of a three-tape machine on a two-tape machine requires $O(t'\log t')$ steps, and so the emulation costs $O(T_M(|x|)\log T_M(|x|))$, where $T_M(n) = O(|\langle M\rangle| \cdot t_1(n))$.

It turns out that the quality of the result we obtain depends on the mapping μ of inputs to machines. Using the naive (identity) mapping of the input x_M to the machine M (such that $x_M = \langle M\rangle$ is a description of M), we can only establish the theorem for $t_2(n) = \omega(n\cdot t_1(n)\log(n\cdot t_1(n)))$. The theorem follows by associating with machine M the input $x_M = \langle M\rangle^m$, where $m = 2^{|\langle M\rangle|}$, because in this case $|\langle M\rangle| = o(\log|x_M|) = o(\log t_1(|x_M|))$. In other words, we may use the mapping μ such that $\mu(x) = M$ if $x = \langle M\rangle^{2^{|\langle M\rangle|}}$ and $\mu(x)$ equals some fixed machine otherwise. ■

Teaching note: Proving the standard version of Theorem 3 cannot be done by associating a sufficiently long input x_M with each machine M, because this does not allow to get rid from an additional unbounded factor multiplier of $t_1(n)\log t_1(n)$. Note that the latter factor needs to be computable (at the very least) and thus cannot be accounted for by the generic ω-notation that appears in the standard version (cf. [4, Thm. 12.9]). Instead, a different approach is taken (see Footnote 2).

Comment: The proof of Theorem 3 associates with each potential machine an input and makes this machine err on this input. The aforementioned association is rather flexible: it should merely be efficiently computed (in the direction from the input to a possible machine) and should be sufficiently shrinking (in that direction).

2.1.2 Impossibility of speed-up for universal computation

The Time Hierarchy Theorem (Theorem 3) implies that the computation of a universal machine cannot be significantly sped-up. That is, consider the function $\mathtt{u}'(\langle M\rangle, x, t) \stackrel{\text{def}}{=} y$ if on input x machine M halts within t steps and outputs the string y, and $\mathtt{u}'(\langle M\rangle, x, t) \stackrel{\text{def}}{=} \perp$ if on input x machine M makes more than t steps. Recall that the value of $\mathtt{u}'(\langle M\rangle, x, t)$ can be computed in $\widetilde{O}(|x| + |\langle M\rangle| \cdot t)$ steps. Theorem 3 implies that this value cannot be computed with significantly less steps.

Theorem 5 *There exists no two-tape Turing machine that, on input $\langle M\rangle, x$ and t, computes $\mathtt{u}'(\langle M\rangle, x, t)$ in $o((t + |x|) \cdot f(M)/\log^2(t + |x|))$ steps, where f is an arbitrary function.*

A similar result holds for any reasonable and general model of computation (cf., Corollary 4).

Proof: Suppose (towards the contradiction) that, for every M, given x and $t > |x|$, the value of $\mathtt{u}'(\langle M\rangle, x, t)$ can be computed in $o(t/\log^2 t)$ steps. Consider an arbitrary time computable t_1 (s.t. $t_i 1(n) > n$) and an arbitrary set $S \in \textsc{Dtime}(t_2)$, where $t_2(n) = t_1(n) \cdot \log^2 t_1(n)$. Let M be a machine of time complexity t_2 that decides membership in S, and consider an algorithm that, on input x, first computes $t = t_1(|x|)$, and then computes (and outputs) the value $\mathtt{u}'(\langle M\rangle, x, t \log^2 t)$. By the time constructibility of t_1, the first computation can be implemented in t steps, and by the contradiction hypothesis the same holds for the second computation. Thus, S can be decided in $\textsc{Dtime}(t_1)$, implying that $\textsc{Dtime}(t_2) = \textsc{Dtime}(t_1)$, which in turn contradicts Theorem 3. ■

2.1.3 Hierarchy theorem for non-deterministic time

Analogously to $\textsc{Dtime}$, for a fixed model of computation (to be understood from the context) and for any function $t : \mathbb{N} \to \mathbb{N}$, we *denote by* $\textsc{Ntime}(t)$ *the class of decision problems that are solvable by a non-deterministic machine of time complexity* t. Alternatively, analogously to the definition of $\mathcal{NP}$, a set $S \subseteq \{0,1\}^*$ is in $\textsc{Ntime}(t)$ if there exists a *linear-time* algorithm V such that the two conditions hold

1. For every $x \in S$ there exists $y \in \{0,1\}^{t(|x|)}$ such that $V(x, y) = 1$.
2. For every $x \notin S$ and every $y \in \{0,1\}^*$ it holds that $V(x, y) = 0$.

We warn that the two formulations are not identical, but in sufficiently strong models (e.g., two-tape Turing machines) they are related up to logarithmic factors (see Exercise 12). The hierarchy theorem itself is similar to the one for deterministic time, except that here we require that $t_2(n) \geq (\log t_1(n+1))^2 \cdot t_1(n+1)$ (rather than $t_2(n) \geq (\log t_1(n))^2 \cdot t_1(n)$). That is:

[2] The function f is not defined with reference to $t_1(|x_M|)$ steps of $M(x_M)$, but rather with reference to the result of emulating $M(x_M)$ while using a total of $t_2(|x_M|)$ steps in the emulation process (i.e., in the algorithm used to compute f). This guarantees that f is in $\textsc{Dtime}(t_2)$, and pushes the problem to showing that f is not in $\textsc{Dtime}(t_1)$, where the problem is resolved by noting that each candidate machine having time complexity t_1 will have its executions fully emulated on any sufficiently long input. Thus, we merely need to associate with each M a disjoint set of infinitely many inputs and make sure that M errs on each of these inputs.

Theorem 6 (non-deterministic time hierarchy for two-tape Turing machines): *For any time computable function t_1 and every function t_2 such that $t_2(n) \geq (\log t_1(n+1))^2 \cdot t_1(n+1)$ and $t_1(n) > n$ it holds that* $\text{NTIME}(t_1)$ *is strictly contained in* $\text{NTIME}(t_2)$.

Proof Sketch: We cannot just apply the proof of Theorem 3, because the Boolean function f defined there requires the ability to determine whether M accepts the input x_M in $t_1(|x_M|)$ steps. In the current context, M is a non-deterministic machine and so the only way we know how to determine this questions (both for a yes and no answers) is to try all the $(2^{t_1(|x_M|)})$ relevant executions. But this would put f in $\text{DTIME}(2^{t_1})$, rather than in $\text{NTIME}(\widetilde{O}(t_1))$, and so a different approach is needed.

We associate with each machine M, a large interval of strings (viewed as integers), denoted $I_M = [\ell_M, u_M]$, such that the various intervals do not intersect and such that it is easy to determine for each string x in which interval it falls. For each $x \in [\ell_M, u_M - 1]$, we define $f(x) = 1$ if and only if there exists a non-deterministic computation of M that accepts the input $x' \stackrel{\text{def}}{=} x + 1$ in $t_1(|x'|)$ steps. As for $f(u_M)$, we define it as *zero* if and only if there exists a non-deterministic computation of M that accepts the input ℓ_M in $t_1(|\ell_M|)$ steps. This definition is coupled with a non-deterministic machine for accepting the set $\{x : f(x) = 1\}$. On input $x \in [\ell_M, u_M - 1]$, this non-deterministic machine emulates a non-deterministic computation of M, which can be done in time $(\log t_1(|x+1|))^2 \cdot t_1(|x+1|)$. On input $x = u_M$, the machine just tries all $2^{t_1(|\ell_M|)}$ ($t_1(|\ell_M|)$-step) executions of $M(\ell_M)$, which can be done in time $t_1(|u_M|) \cdot t_1(|u_M|)$ provided the interval was chosen to be large enough (i.e., $t_1(|u_M|) \geq 2^{t_1(|\ell_M|)}$, which certainly holds when $|u_M| \geq 2^{t_1(|\ell_M|)}$).

Defining f this way guarantees that it is not in $\text{NTIME}(t_1)$. Suppose on the contrary, that some non-deterministic machine M of time complexity t_1 accepts the set $\{x : f(x) = 1\}$. We define a Boolean function A_M such that $A_M(x) = 1$ if and only if there exists a non-deterministic computation of M that accepts the input x, and note that $A_M(x) = f(x)$. Focusing on the interval $[\ell_M, u_M]$, we have $A_M(x) = f(x)$ for every $x \in [\ell_M, u_M]$, which (combined with the definition of f) implies that $A_M(x) = f(x) = A_M(x+1)$ for every $x \in [\ell_M, u_M - 1]$ and $A_M(u_M) = f(u_M) = 1 - A_M(\ell_M)$. Thus, we reached a contraction (i.e., $A_M(\ell_M) = \cdots = A_M(u_M) = 1 - A_M(\ell_M)$). $\square$

2.2 Time Gaps and Speed-Up

In contrast to Theorem 3, there exists functions $t : \mathbb{N} \to \mathbb{N}$ such that $\text{DTIME}(t) = \text{DTIME}(t^2)$ (or even $\text{DTIME}(t) = \text{DTIME}(2^t)$). Needless to say, these functions are not time-constructible (and thus the aforementioned fact does not contradict Theorem 3). The reason for this phenomenon is that, for such functions t, there exists not algorithms that have time complexity above t but below t^2 (resp., 2^t).

Theorem 7 (the time gap theorem): *For every non-decreasing computable function $g : \mathbb{N} \to \mathbb{N}$ there exists a non-decreasing computable function $t : \mathbb{N} \to \mathbb{N}$ such that* $\text{DTIME}(t) = \text{DTIME}(g(t))$.

The forgoing examples referred to $g(m) = m^2$ and $g(m) = 2^m$. Since we are mainly interested in dramatic gaps (i.e., super-polynomial functions g), the model of computation does not matter here (as long as it is reasonable and general).

Proof: Consider an enumeration of all possible algorithms (or machines) and let t_i denote the time complexity of the i^{th} algorithm. Recall that we cannot enumerate only machines that halt on every input, and that $t_i(n) = \infty$ if the i^{th} machine does not halt on some n-bit long input. The basic idea is to define t such that no t_i is "sandwiched" between t and $g(t)$. Intuitively, if $t_i(n)$ is finite, then we may define t such that $t_i(n) < t(n)$, whereas if $t_i(n) = \infty$ then any finite value of $t(n)$ will do (because then $t_i(n) > g(t(n))$). The problem is that we want t to be computable, whereas we cannot tell whether or not $t_i(n)$ is finite. However, we don't really need to make the latter decision: for each candidate value v of $t(n)$, we should just determine whether or not $t_i(n) \in [v, g(v)]$, which can be decided by running the i^{th} machine for at most $g(v)$ steps (on each n-bit long string). Bearing in mind that we should deal with all possible machines, we obtain the following procedure for setting $t(n)$.

Let $\mu : \mathbb{N} \to \mathbb{N}$ be any unbounded and computable function (e.g., $\mu(n) = n$ will do). Starting with $v = 1$, we keep incrementing v until v satisfies, for every $i \in \{1, ..., \mu(n)\}$, either $t_i(n) < v$ or $t_i(n) > g(v)$. This condition can be verified by computing $\mu(n)$ and $g(v)$, and emulating the execution of of the $\mu(n)$ machines on each of the n-bit long strings for $g(v)$ steps. The procedure sets $g(n)$ to equal the first value v satisfying the aforementioned condition, and halts. To show that the procedure halts on every n, consider the set $H_n \subseteq \{1, ..., \mu(n)\}$ of indices of the relevant machines that halt on all inputs of length n. Then the procedure definitely halts before reaching the value $v = m + 2$, where $m = \max_{i \in H_n}\{t_i(n)\}$. (Indeed, the procedure may halt with $v \leq m$, provide that $g(v) < m$.) ■

Comment: The function t defined by the foregoing proof is computable in time that exceeds $g(t)$. Specifically, the presented procedure computes $t(n)$ (as well as $g(f(n))$) in time $\widetilde{O}(2^n \cdot g(t(n)) + T_g(t(n)))$, where $T_g(m)$ denotes the number of steps required to compute $g(m)$ on input m.

Speed-up Theorems. Theorem 7 can be viewed as asserting that some time complexity classes (i.e., $\text{DTIME}(g(t))$ in the theorem) collapse to lower classes (i.e., to $\text{DTIME}(t)$). A conceptually related phenomenon is of problems that have no optimal algorithm (not even in a very mild sense); that is, every algorithm for these ("pathological") problems can be drastically sped-up. It follows that the complexity of these problems can not be defined (i.e., as the complexity of the best algorithm solving this problem). The following drastic speed-up theorem should not be confused with the linear speed-up that is an artifact of the definition of a Turing machine (see Exercise 13).[3]

Theorem 8 (the time speed-up theorem): *For every computable* (and super-linear) *function g there exists a decidable set S such that if $S \in \text{DTIME}(t)$ then $S \in \text{DTIME}(t')$ for t' satisfying $g(t'(n)) < t(n)$.*

Taking $g(n) = n^2$ (or $g(n) = 2^n$), the theorem asserts that, for every t, if $S \in \text{DTIME}(t)$ then $S \in \text{DTIME}(\sqrt{t})$ (resp., $S \in \text{DTIME}(\log t)$). Note that Theorem 8 can be applied any number of times, which means that we cannot give a reasonable estimate to the complexity of deciding membership in S. In contrast, recall that in some important cases optimal algorithms for solving computational problems do exist. Specifically, algorithms solving (candid) search problems in NP cannot be speed-up nor can the computation of a universal machine (see Theorem 5).

The proof of Theorem 8 (given in [4, Sec. 12.6]) constructs a set S in $\text{DTIME}(t') \setminus \text{DTIME}(t'')$ for any $t'(n) = t(n - O(1))$ and $t''(n) = t(n - \omega(1))$, where $t(n)$ denoted g iterated n times on 2 (i.e., $t(n) = g^{(n)}(2)$, where $g^{(i+1)}(m) = g(g^{(i)}(m))$ and $g^{(1)} = g$).

[3] We note that this fact was implicitly addressed in the proof of Theorem 3, by allowing an emulation overhead that depends on the length of the description of the emulated machine.

3 Space Hierarchies and Gaps

Hierarchy and Gap Theorems analogous to Theorem 3 and Theorem 7, respectively, are known for space complexity. In fact, since space-efficient emulation of space-bounded machines is simpler than time-efficient emulations of time-bounded machines, the results tend to be sharper. This is most conspicuous in the case of the separation result (stated next), which is optimal (in light of linear speed-up results; see Exercise 13).

Before stating the result, we need a few preliminaries. We assume familiarity with the definition of space complexity. As in case of time complexity, we consider a specific model of computation, but the results hold for any other reasonable and general model. Specifically, we consider three-tape Turing machines, because we designate two special tapes for input and output. For any function $s : \mathbb{N} \to \mathbb{N}$, we *denote by* $\mathrm{DSPACE}(s)$ *the class of decision problems that are solvable in space complexity* s. Analogously to Definition 2, we call a function $s : \mathbb{N} \to \mathbb{N}$ space constructible if there exists an algorithm that on input n outputs $s(n)$ using at most $s(n)$ cells of the work-tape. Actually, functions like $s_1(n) = \log n$, $s_2(n) = (\log n)^2$, and $s_3(n) = 2^n$ are computable using $\log s_i(n)$ space.

Theorem 9 (space hierarchy for three-tape Turing machines): *For any space computable function* s_2 *and every function* s_1 *such that* $s_2 = \omega(s_1)$ *and* $s_1(n) > \log n$ *it holds that* $\mathrm{DSPACE}(s_1)$ *is strictly contained in* $\mathrm{DSPACE}(s_2)$.

Theorem 9 is analogous to the traditional version of Theorem 3 (rather to the one we presented), and is proven using the alternative approach sketched in Footnote 2.

Notes

The hierarchy theorems (e.g., Theorem 3) were proved by Hartmanis and Stearns [3]. Gap theorems (e.g., Theorem 7, often referred to as Borodin's Gap Theorem) were proven by Borodin [2]. A axiomatic treatment of complexity measures and corresponding speed-up theorems (e.g., Theorem 8, often referred to as Blum's Speed-up Theorem) are due to Blum [1].

Exercises

Exercise 10 For $s < 2^n$, prove that the number of Boolean functions over $\{0,1\}^n$ that are computed by Boolean circuits of size s is at least $2^{s/O(\log s)}$ and at most $\binom{s^2}{s}$.

Guideline: Any Bollean function $f : \{0,1\}^\ell \to \{0,1\}$ can be computed by a circuit of size $s = O(\ell \cdot 2^\ell)$. Thus, circuits of size s may compute $2^{2^\ell} > 2^{s/O(\log s)}$ different Boolean functions. On the other hand, the number of circuits of size s is less than $\binom{s^2}{s}$, where the upper-bound represents the number of possible choices of pair of gates that feed any gate in the circuit.

Exercise 11 Referring to a reasonable model of computation (and assuming that the input length is not given explicitly), prove that any algorithm that has sub-linear time-complexity actually has constant time-complexity.

(Hint: Consider the question of whether or not there exists an infinite set of strings S such that when invoked on any input $x \in S$ the algorithm reads all of x. Note that if S is infinite then the algorithm cannot have sub-linear time-complexity, and prove that if S is finite then the algorithm has constant time-complexity.)

Exercise 12 Prove that the two definitions of NTIME, presented at the end of Section 2.1, are related up to logarithmic factors. Note the importance of condition that V has linear (rather than polynomial) time-complexity.
(Hint: when emulating a non-deterministic machine by the verification procedure V, encode the non-deterministic choices in y while using sufficient padding such that t steps of the original machine can be emulated in $|y| = O(t \log t)$ steps of V.)

Exercise 13 Prove that any problem that can be solved by a two-tape Turing machine that has time complexity t can be solved by another two-tape Turing machine having time complexity t', where $t'(n) = O(n) + (t(n)/2)$.
(Hint: Consider a machine that uses a larger alphabet, capable of encoding a constant number of symbols of the original machine, and thus capable of emulating a constant number of steps of the original machine in a smaller constant number of steps. Note that the $O(n)$ term accounts to a preprocessing required to encode the input in the work-alphabet of the new machine, and that a similar result for one-tape Turing machine seems to require a $O(n^2)$ term.)
State and prove an analogous result for space complexity.

References

[1] M. Blum. A Machine-Independent Theory of the Complexity of Recursive Functions. *Journal of the ACM*, Vol. 14 (2), pages 290–305, 1967.

[2] A. Borodin. Computational Complexity and the Existence of Complexity Gaps. *Journal of the ACM*, Vol. 19 (1), pages 158–174, 1972.

[3] J. Hartmanis and R.E. Stearns. On the Computational Complexity of of Algorithms. *Transactions of the AMS*, Vol. 117, pages 285–306, 1965.

[4] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.