Chapter 4

More Resources, More Power?

More electricity, less toil.

The Israeli Electricity Company, 1960s

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 "reasonable" 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 provided in the current chapter.

Summary: 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 problems that are solvable 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 4.1). We then turn to the time complexity of uniform algorithms; indeed, hierarchy and gap theorems for time-complexity, presented in Section 4.2, constitute the main part of the current chapter. We end by mentioning analogous results for space-complexity (see Section 4.3, which may also be read after Section 5.1).

4.1 Non-uniform complexity hierarchies

The model of machines that use advice (cf. §1.2.4.2 and Section 3.1.2) 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 (see Definition 3.5). 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 4.1 *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 $\ell \stackrel{\text{def}}{=} \ell' + \delta$, 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^{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 $f_{\overline{a}}(x) \stackrel{\text{def}}{=} A(a_{|x|}, x)$ 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 4.1), an obvious lower-bound on this number is $2^{s/O(\log s)}$ whereas an obvious upper-bound is $s^{2s} = 2^{2s\log_2 s}$. (Compare these bounds to the lower-bound $2^{\ell'(n)}$ and the upper-bound $2^{\ell'(n)+((\delta(n)-2)/2)}$, which were used in the proof of Theorem 4.1.)

4.2 Time Hierarchies and Gaps

In this section we show that in the "reasonable 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 introductory comments to the current chapter, the "reasonable cases" correspond to time bounds that can be determined by the algorithm itself within the specified time complexity.

We stress that also in the aforementioned "reasonable cases", the added power does not necessarily refer to natural computational problems. That is, like in the case of non-uniform complexity (i.e., Theorem 4.1), the hierarchy theorems are proved by introducing artificial computational problems. Needless to say, we do not know of natural problems in $\mathcal{P}$ that are provably unsolvable in cubic (or some other fixed polynomial) time (on, say, a two-tape Turing machine). Thus, although $\mathcal{P}$ contains an infinite hierarchy of computational problems, each requiring significantly more time than the other, we know of no such hierarchy of natural computational problems. In contrast, so far it has been the case that any natural problem that was shown to be solvable in polynomial-time was eventually followed by algorithms having running-time that is bounded by a moderate polynomial.

[1] It suffices to show that this measure is strictly less than one. This is easily done by considering, for every n, the performance of any algorithm A' having description of length shorter than $(\delta(n)-2)/2$ on all inputs of length n.

4.2.1 Time Hierarchies

Note that the non-uniform computing devices, considered in Section 4.1, were explicitly given the relevant resource bounds (e.g., the length of advice). Actually, they were given the resources themselves (e.g., the advice itself) and did not need to monitor their usage of these resources. In contrast, when designing algorithms of arbitrary 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 is to have the algorithm compute this bound (i.e., $t(|x|)$) before doing anything else. This, in turn, requires the algorithm to read the entire input (see Exercise 4.3) 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. [119, Sec. 12.3])).

Definition 4.2 (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 $\mathrm{poly}(\log t(n))$ 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 $\mathrm{poly}(\log t_i(n))$ 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* $\mathrm{DTIME}(t)$ *the class of decision problems that are solvable in time complexity t*. We call the reader's attention to Exercise 4.7 that asserts that in many cases $\mathrm{DTIME}(t) = \mathrm{DTIME}(t/2)$.

4.2.1.1 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 in terms of 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 Theorem 4.3 asserts that *for any time constructible 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* $\mathrm{DTIME}(t_1)$ *is strictly contained in* $\mathrm{DTIME}(t_2)$. The current version is only slightly weaker, but it allows a somewhat simpler and more intuitive proof. We comment on the proof of the standard version of Theorem 4.3 after proving the current version.

Theorem 4.3 (time hierarchy for two-tape Turing machines): *For any time constructible 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* $\mathrm{DTIME}(t_1)$ *is strictly contained in* $\mathrm{DTIME}(t_2)$.

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, where the constant in the O-notation depends on the emulated machine. Before proving Theorem 4.3, we derive the following corollary.

Corollary 4.4 (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* $\text{DTIME}(t_1)$ *is strictly contained in* $\text{DTIME}(t_2)$.

It follows that, for every such model and every polynomial t (such that $t(n) > n$), there exist problems in $\mathcal{P}$ that are not in $\text{DTIME}(t)$. It also follows that $\mathcal{P}$ is a strict subset of $\mathcal{E}$ and even of "quasi-polynomial time"; moreover, $\mathcal{P}$ is a strict subset of $\text{DTIME}(q)$, where $q(n) = n^{\log_2 n}$ (and even $q(n) = n^{\log_2 \log_2 n}$).

Proof of Corollary 4.4: Letting DTIME_2 denote the classes that correspond to two-tape Turing machines, we note that $\text{DTIME}(t_1) \subseteq \text{DTIME}_2(t_1')$ and $\text{DTIME}(t_2) \supseteq \text{DTIME}_2(t_2')$, where $t_1' = \text{poly}(t_1)$ and t_2' is defined such that $t_2(n) = \text{poly}(t_2'(n))$. 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 (i.e., $p(p_1^{-1}(m)) \geq p_2(m^2)$), it holds that

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

where the last inequality holds by the choice of p and the first inequality holds by the corollary's hypothesis (i.e., $t_2 > p(t_1)$). Invoking Theorem 4.3 (while noting that $t_2'(n) > t_1'(n)^2$), we have $\text{DTIME}_2(t_2') \supset \text{DTIME}_2(t_1')$. Combining this with the aforementioned relations between DTIME and DTIME_2, the corollary follows. ■

Proof of Theorem 4.3: The idea is constructing 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 $f(x_M) \neq M'(x_M)$, *where $M'(x)$ denotes an emulation of $M(x)$ that is suspended after $t_1(|x|)$ steps.* For example, we may define $f(x_M) = 1 - M'(x_M)$. We note that M' is used instead of M in order to allow computing f in time that is related to t_1. The point is that M is just an arbitrary machine that is associated to the input x_M, and so M does not necessarily run in time t_1 (but, by construction, the corresponding M' does run in time t_1).

Implementing the foregoing idea calls for an efficient association of machines to inputs as well as for a relatively efficient emulation of t_1 steps of an arbitrary machine. As shown next, both requirements can be met easily. Actually, we are going to use a mapping μ of inputs to machines (i.e., μ will map the aforementioned x_M to M) such that each machine is in the range of μ and μ is very easy to compute (e.g., indeed, for starters, assume that μ is the identity mapping). Thus, by construction, $f \notin \text{DTIME}(t_1)$. The issue is presenting a relatively efficient algorithm for computing f; that is, showing that $f \in \text{DTIME}(t_2)$.

The algorithm for computing f as well as the definition of f (sketched in the first paragraph) are straightforward: On input x, the algorithm 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 $1 - M'(x)$. Recall that $M'(x)$ denotes the time-truncated emulation of $M(x)$ (i.e., this emulation suspended after t steps). Thus, $f(x) = 1 - M'(x)$ if $M = \mu(x)$ and (say) $f(x) = 0$ otherwise.

Using the time-constructability of t_1 and ignoring the easy computation of μ, we focus on the question of how much time is required for emulating t steps of machine M (on input x). We should bear in mind that the time-complexity of our algorithm needs to be analyzed in the two-tape Turing-machine model, whereas M itself is a two-tape Turing-machine. We start by implementing our algorithm on a three-tape Turing-machine, and next emulate this machine on a two-tape Turing-machine.

The obvious implementation of our algorithm on a three-tape Turing-machine uses two tapes for the emulation itself and designates the third tape for the actions of the emulation procedure (e.g., storing the code of the emulated machine and maintaining a step-counter). Thus, each step of the the two-tape machine M is emulated using $O(|\langle M \rangle|)$ steps on the three-tape machine.[2] This includes also the amortized complexity of maintaining a step-counter for the emulation (see Exercise 4.4).

Next, we need to emulate the foregoing three-tape machine on a two-tape machine. This is done by using the fact (cf., e.g., [119, Thm. 12.6]) that t' steps of a three-tape machine can be emulated on a two-tape machine in $O(t' \log t')$ steps. Thus, the complexity of computing f on input x is upper-bounded by $O(T_{\mu(x)}(|x|) \log T_{\mu(x)}(|x|))$, where $T_M(n) = O(|\langle M \rangle| \cdot t_1(n))$ represents the cost of emulating $t_1(n)$ steps of the two-tape machine M on a three-tape machine (as in the foregoing discussion).

It turns out that the quality of the result we obtain depends on the choice of the mapping μ (of inputs to machines). Using the naive (identity) mapping (i.e., $\mu(x) = x$) we can only establish the theorem for $t_2(n) = \widetilde{O}(n \cdot t_1(n))$ rather than $t_2(n) = \widetilde{O}(t_1(n))$, because in this case $T_{\mu(x)}(|x|) = O(|x| \cdot t_1(|x|))$. (Note that in this case $x_M = \langle M \rangle$ is a description of $\mu(x_M) = M$.) The theorem follows by associating the machine M with the input $x_M = \langle M \rangle 01^m$, where $m = 2^{|\langle M \rangle|}$; that is, we may use the mapping μ such that $\mu(x) = M$ if $x = \langle M \rangle 01^{2^{|\langle M \rangle|}}$ and $\mu(x)$ equals some fixed machine otherwise. In this case $|\mu(x)| < \log_2 |x| < \log t_1(|x|)$ and so $T_{\mu(x)}(|x|) = O((\log t_1(|x|)) \cdot t_1(|x|))$. The theorem follows. ■

[2] This overhead accounts both for searching the code of M for the adequate action and for the effecting of this action (which may refer to a larger alphabet than the one used by the emulator).

Teaching note: Proving the standard version of Theorem 4.3 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 in $T_{\mu(x)}(|x|)$ (i.e., the $|\mu(x)|$ factor that multiplies $t_1(|x|)$). 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. [119, Thm. 12.9]). Instead, a different approach is taken (see Footnote 3).

Technical Comments. The proof of Theorem 4.3 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). Specifically, we used the mapping μ such that $\mu(x) = M$ if $x = \langle M\rangle 01^{2^{|\langle M\rangle|}}$ and $\mu(x)$ equals some fixed machine otherwise. We comment that each machine can be made to err on infinitely many inputs by redefining μ such that $\mu(x) = M$ if $\langle M\rangle 01^{2^{|\langle M\rangle|}}$ is a suffix of x (and $\mu(x)$ equals some fixed machine otherwise). We also comment that, in contrast to the proof of Theorem 4.3, the proof of Theorem 1.5 utilizes a rigid mapping of inputs to machines (i.e., there $\mu(x) = M$ if $x = \langle M\rangle$).

Digest: Diagonalization. The last comment highlights the fact that the proof of Theorem 4.3 is merely a sophisticated version of the proof of Theorem 1.5. Both proofs refer to versions of the universal function, which in the case of the proof of Theorem 4.3 is (implicitly) defined such that its value at $(\langle M\rangle, x)$ equals $M'(x)$, where $M'(x)$ denotes an emulation of $M(x)$ that is suspended after $t_1(|x|)$ steps.[4] Actually, both proofs refers to the "diagonal" of the aforementioned function, which in the case of the proof of Theorem 4.3 is only defined implicitly. That is, the value of the diagonal function at x, denoted $d(x)$, equals the value of the universal function at $(\langle\mu(x)\rangle, x)$. This is actually a definitional schema, as the choice of the function μ remains unspecified. Indeed, setting $\mu(x) = x$ corresponds to a "real" diagonal in the matrix depicting the universal function, but any other choice of a 1-1 mappings μ also yields a "kind of diagonal" of the universal function. Either way, the function f is defined such that for every x it holds that $f(x) \neq d(x)$. This guarantees that no machine of time-complexity t_1 can compute f, and the focus is on presenting an algorithm that computes f (which, needless to say, has time-complexity greater than t_1). Part of the proof of Theorem 4.3 is devoted to

[3] In the standard proof 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 $\text{DTIME}(t_2)$, and "pushes the problem" to showing that f is not in $\text{DTIME}(t_1)$. It also explains why t_2 (rather than t_1) is assumed to be time constructible. As for the foregoing problem, it is resolved by observing that for each relevant machine (i.e., having time complexity t_1) the executions on any sufficiently long input will be fully emulated. 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.

[4] Needless to say, in the proof of Theorem 1.5, $M' = M$.

selecting μ in a way that minimizes the time-complexity of computing f, whereas in the proof of Theorem 1.5 we merely need to guarantee that f is computable.

4.2.1.2 Impossibility of speed-up for universal computation

The Time Hierarchy Theorem (Theorem 4.3) 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 4.3 implies that this value cannot be computed within significantly less steps.

Theorem 4.5 *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 4.4). In particular, it follows that $\mathtt{u}'$ is not computable in polynomial time (because the input t is presented in binary). In fact, one can show that *deciding whether or not M halts on input x in t steps* (i.e., membership in the set $\{(\langle M\rangle, x, t) : \mathtt{u}'(\langle M\rangle, x, t) \neq \perp\}$) is not in $\mathcal{P}$; see Exercise 4.5.

Proof: Suppose (towards the contradiction) that, for every fixed 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, where the o-notation hides a constant that may depend on M. Consider an arbitrary time constructible t_1 (s.t. $t_1(n) > n$) and an arbitrary set $S \in \text{DTIME}(t_2)$, 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 $\text{DTIME}(2t_1) = \text{DTIME}(t_1)$, implying that $\text{DTIME}(t_2) = \text{DTIME}(t_1)$, which in turn contradicts Theorem 4.3. ■

4.2.1.3 Hierarchy theorem for non-deterministic time

Analogously to 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* NTIME($t$) *the class of sets that are accepted by some non-deterministic machine of time complexity t.* Alternatively, analogously to the definition of $\mathcal{NP}$, a set $S \subseteq \{0,1\}^*$ is in 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 4.6). 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:

Theorem 4.6 (non-deterministic time hierarchy for two-tape Turing machines): *For any time-constructible and monotonicly non-decreasing 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: We cannot just apply the proof of Theorem 4.3, because the Boolean function f defined there requires the ability to determine whether there exists a computation of M that 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 question (both for a "yes" and "no" answers) is to try all the $(2^{t_1(|x_M|)})$ relevant executions.[5] 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 (non-deterministic) machine M, a large interval of strings (viewed as integers), denoted $I_M = [\alpha_M, \beta_M]$, such that the various intervals do not intersect and such that it is easy to determine for each string x in which interval it resides. For each $x \in [\alpha_M, \beta_M - 1]$, 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 $t_1(|x'|) \leq t_1(|x|+1)$ steps. Thus, unless either M (non-deterministically) accepts each string in the interval I_M or M (non-deterministically) accepts no such string, it (i.e., M) fails to (non-deterministically) accept $\{x : f(x)=1\}$ (because M must non-deterministically accept $x' = x+1$ if and only if it non-deterministically accepts x). So it is left to deal with the case that M is invariant on I_M, which is where the definition of the value of $f(\beta_M)$ comes into play: We define $f(\beta_M)$ to equal *zero* if and only if there exists a non-deterministic computation of M that accepts the input α_M in $t_1(|\alpha_M|)$ steps. We shall select β_M to be large enough relative to α_M such that we can afford to try all possible computations of M on input α_M. Details follow.

We present the following non-deterministic machine for accepting the set $\{x : f(x) = 1\}$. We assume that, on input x, it is easy to determine the machine M that corresponds to the interval $[\alpha_M, \beta_M]$ in which x reside.[6] On input $x \in [\alpha_M, \beta_M - 1]$, this non-deterministic machine emulates a (single) non-deterministic computation of M on input $x' = x+1$, and decides accordingly. Indeed, this emulation can be performed in time $(\log t_1(|x+1|))^2 \cdot t_1(|x+1|) \leq t_2(|x|)$. On input $x = \beta_M$, our machine just tries all $2^{t_1(|\alpha_M|)}$ executions of M on input α_M and decides in a suitable manner; that is, our machine emulates all $2^{t_1(|\alpha_M|)}$ possible executions of

[5] Indeed, we can non-deterministically recognize "yes" answers in $\widetilde{O}(t_1(|x_M|))$ steps, but we cannot do so for "no" answers.

[6] For example, we may partition the strings to consequetive intervals such that the i^{th} interval, denoted $[\alpha_i, \beta_i]$, corresponds to the i^{th} machine and for $T_1(m) = 2^{2t_1(m)}$ it holds that $\beta_i = 1^{T_1(|\alpha_i|)}$ and $\alpha_{i+1} = 0^{T_1(|\alpha_i|)+1}$. Note that $|\beta_i| = T_1(|\alpha_i|)$, and thus $t_1(|\beta_i|) > t_1(|\alpha_i|) \cdot 2^{t_1(|\alpha_i|)}$.

$M(\alpha_M)$ and accepts β_M if and only if all the emulated executions ended rejecting α_M. Note that this part of the emulation is deterministic, and it amounts to emulating $T_M \stackrel{\text{def}}{=} 2^{t_1(|\alpha_M|)} \cdot t_1(|\alpha_M|)$ steps of M. By a suitable choice of the interval $[\alpha_M, \beta_M]$, this number (i.e., T_M) is smaller than $t_1(|\beta_M|)$ (e.g., $|\beta_M| \geq T_M$ implies $T_M \leq t_1(|\beta_M|)$), and it follows that T_M steps of M can be emulated in time $(\log_2 t_1(|\beta_M|))^2 \cdot t_1(|\beta_M|) \leq t_2(|\beta_M|)$. Thus, f is in $\text{NTIME}(t_2)$.

Finally, we show that defining f as in the foregoing indeed guarantees that f 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 by the contradiction hypothesis $A_M(x) = f(x)$. Focusing on the interval $[\alpha_M, \beta_M]$, we have $A_M(x) = f(x)$ for every $x \in [\alpha_M, \beta_M]$, which (combined with the definition of f) implies that $A_M(x) = f(x) = A_M(x+1)$ for every $x \in [\alpha_M, \beta_M - 1]$ and $A_M(\beta_M) = f(\beta_M) = 1 - A_M(\alpha_M)$. Thus, we reached a contraction (because we got $A_M(\alpha_M) = \cdots = A_M(\beta_M) = 1 - A_M(\alpha_M)$). ■

4.2.2 Time Gaps and Speed-Up

In contrast to Theorem 4.3, 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 4.3). 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 4.7 (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), which also includes machines that do not halt on some inputs. (Recall that we cannot enumerate only all machines that halt on every input.) Let t_i denote the time complexity of the i^{th} algorithm; that is, $t_i(n) = \infty$ if the i^{th} machine does not halt on some n-bit long input and otherwise $t_i(n) = \max_{x \in \{0,1\}^n}\{T_i(x)\}$, where $T_i(x)$ denotes the number of steps taken by the i^{th} machine on input x.

The basic idea is to define t such that no t_i is "sandwiched" between t and $g(t)$, and thus no algorithm will have time complexity between t and $g(t)$. Intuitively, if $t_i(n)$ is finite, then we may define t such that $t(n) > t_i(n)$ and thus guarantee that $t_i(n) \notin [t(n), g(t(n))]$, whereas if $t_i(n) = \infty$ then any finite value of $t(n)$ will do (because then $t_i(n) > g(t(n))$). Thus, for every m and n, we can define $t(n)$ such that $t_i(n) \notin [t(n), g(t(n))]$ for every $i \in [m]$ (e.g., $t(n) = \max_{i \in [m]: t_i(n) \neq \infty}\{t_i(n)\} +$

1).[7] This yields a weaker version of the theorem, in which the function t is not computable.

The problem is that we want t to be computable, whereas given n we cannot tell whether or not $t_i(n)$ is finite. However, we do not 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 $g(v)+1$ steps (on each n-bit long string). That is, as far as the i^{th} machine is concerned, we should just find a value v such that either $v > t_i(n)$ or $g(v) < t_i(n)$ (which includes the case $t_i(n) = \infty$). This can be done by starting with $v = v_0$ (where, say, $v_0 = n+1$), and increasing v until either $v > t_i(n)$ or $g(v) < t_i(n)$. The point is that if $t_i(n)$ is infinite then we we output $v = v_0$ after emulating $2^n \cdot (g(v_0)+1)$ steps, and otherwise we output output $v \leq t_i(n)+1$ after performing at most $\sum_{j=v_0}^{t_i(n)} 2^n \cdot j$ emulation steps. 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 $v = n+1$, 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 each of the first $\mu(n)$ machines on each of the n-bit long strings for $g(v)+1$ steps. The procedure sets $t(n)$ to equal the first value v satisfying the aforementioned condition, and halts.

To show that the foregoing procedure halts on every n, consider the set $H_n \subseteq \{1, ..., \mu(n)\}$ of the indices of the (relevant) machines that halt on all inputs of length n. Then, the procedure definitely halts before reaching the value $v = T_n + 2$, where $T_n = \max_{i \in H_n}\{t_i(n)\}$. (Indeed, the procedure may halt with a value $v \leq T_n$, but this will happen only if $g(v) < T_n$.)

Finally, for the foregoing function t, we prove that $\text{DTIME}(t) = \text{DTIME}(g(t))$ holds. Indeed, let $S \in \text{DTIME}(g(t))$ and suppose that the i^{th} algorithm decides S in time at most $g(t)$; that is, for every n, it holds that $t_i(n) \leq g(t(n))$. Then (by the construction of t), for every n satisfying $\mu(n) \geq i$, it holds that $t_i(n) < t(n)$. It follows that the i^{th} algorithm decides S in time at most t on all but finitely many inputs. Combining this algorithm with a "look-up table" machine that handles the exceptional inputs, the theorem follows. ■

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 4.7 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

[7] We may assume, without loss of generality, that $t_1(n) = 1$ for every n; e.g., by letting the machine that always halts after a single step be the first machine in our enumeration.

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 4.7).[8]

Theorem 4.8 (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 4.8 can be applied any (constant) 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 (see Theorem 2.31), nor can the computation of a universal machine (see Theorem 4.5).

We refrain from presenting a proof of Theorem 4.8, but comment on the complexity of the sets involved in this proof. The proof (presented in [119, Sec. 12.6]) provides a construction of a set S in $\text{DTIME}(t') \setminus \text{DTIME}(t'')$ for $t'(n) = h(n - O(1))$ and $t''(n) = h(n - \omega(1))$, where $h(n)$ denoted g iterated n times on 2 (i.e., $h(n) = g^{(n)}(2)$, where $g^{(i+1)}(m) = g(g^{(i)}(m))$ and $g^{(1)} = g$). The set S is constructed such that for every $i > 0$ there exists a $j > i$ and an algorithm that decides S in time t_i but not in time t_j, where $t_k(n) = h(n - k)$.

4.3 Space Hierarchies and Gaps

Hierarchy and Gap Theorems analogous to Theorem 4.3 and Theorem 4.7, 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 4.7).

Before stating the result, we need a few preliminaries. We refer the reader to §1.2.3.4 for a definition of space complexity (and to Chapter 5 for further discussion). 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* $\text{DSPACE}(s)$ *the class of decision problems that are solvable in space complexity* s. Analogously to Definition 4.2, we call a function $s : \mathbb{N} \to \mathbb{N}$ space constructible if there exists

[8] We note that the linear speed-up phenomenon was implicitly addressed in the proof of Theorem 4.3, by allowing an emulation overhead that depends on the length of the description of the emulated machine.

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

Theorem 4.9 is analogous to the traditional version of Theorem 4.3 (rather to the one we presented), and is proven using the alternative approach sketched in Footnote 3. The details are left as an exercise (see Exercise 4.9).

Chapter Notes

The material presented in this chapter predates the theory of NP-completeness and the dominant stature of the P-vs-NP Question. At these early days, the field (to be known as complexity theory) did not yet develop an independent identity and its perspectives were dominated by two classical theories: the theory of computability (and recursive function) and the theory of formal languages. Nevertheless, we believe that the results presented in this chapter are interesting for two reasons. Firstly, as stated up-front, these results address the natural question of under what conditions is it the case that more computational resources help. Secondly, these results demonstrate how far one can get with respect to "generic" questions regarding an arbitrary complexity measure; that is, questions that refer to arbitrary resource bounds (e.g., the relation between $\text{DTIME}(t_1)$ and $\text{DTIME}(t_2)$ for arbitrary t_1 and t_2). We note that the P-vs-NP Question as well as the related questions that will be addressed in the rest of this book are not "generic" since they refer to specific classes (which capture natural computational issues). The foregoing comment many be clarified by the concrete discussion in Section 5.3.3.

The hierarchy theorems (e.g., Theorem 4.3) were proved by Hartmanis and Stearns [110]. Gap theorems (e.g., Theorem 4.7, often referred to as Borodin's Gap Theorem) were proven by Borodin [44]. A axiomatic treatment of complexity measures and corresponding speed-up theorems (e.g., Theorem 4.8, often referred to as Blum's Speed-up Theorem) are due to Blum [36].

Exercises

Exercise 4.1 Let $F_n(s)$ denote the number of different Boolean functions over $\{0,1\}^n$ that are computed by Boolean circuits of size s. Prove that, for any $s < 2^n$, it holds that $F_n(s) \geq 2^{s/O(\log s)}$ and $F_n(s) \leq s^{2s}$.

Guideline: Any Boolean function $f : \{0,1\}^\ell \to \{0,1\}$ can be computed by a circuit of size $s_\ell = O(\ell \cdot 2^\ell)$. Thus, for every $\ell \leq n$, it holds that $F_n(s_\ell) \geq 2^{2^\ell} > 2^{s_\ell/O(\log s_\ell)}$. On the other hand, the number of circuits of size s is less than $2^s \cdot \binom{s^2}{s}$, where the second factor represents the number of possible choices of pair of gates that feed any gate in the circuit.

Exercise 4.2 (advice can speed-up computation) For every time constructible function t, show that there exists a set S in $\text{DTIME}(t^2) \setminus \text{DTIME}(t)$ that can be decided in linear-time using an advice of linear length (i.e., $S \in \text{DTIME}(\ell)/\ell$ where $\ell(n) = O(n)$).

Guideline: Starting with a set $S' \in \text{DTIME}(T^2) \setminus \text{DTIME}(T)$, where $T(m) = t(2^m)$, consider the set $S = \{x0^{2^{|x|}-|x|} : x \in S'\}$.

Exercise 4.3 Referring to a reasonable model of computation (and assuming that the input length is not given explicitly (e.g., as in Definition 10.10)), prove that any algorithm that has sub-linear time-complexity actually has constant time-complexity.

Guideline: 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 4.4 (constant amortized time step-counter) A step-counter is an algorithm that runs for a number of steps that is specified in its input. Actually, such an algorithm may run for a somewhat larger number of steps but halt after issuing a number of "signals" as specified in its input, where these signals are defined as entering (and leaving) a designated state (of the algorithm). A step-counter may be run in parallel to another procedure in order to suspend the execution after a desired number of steps (of the other procedure) has elapsed. Show that there exists a simple deterministic machine that, on input n, halts after issuing n signals while making $O(n)$ steps.

Guideline: A slightly careful implementation of the straightforward algorithm will do, when coupled with an "amortized" time-complexity analysis.

Exercise 4.5 (a natural set in $\mathcal{E} \setminus \mathcal{P}$) In continuation to the proof of Theorem 4.5, prove that the set $\{(\langle M \rangle, x, t) : \mathtt{u}'(\langle M \rangle, x, t) \neq \perp\}$ is in $\mathcal{E} \setminus \mathcal{P}$, where $\mathcal{E} \stackrel{\text{def}}{=} \cup_c \text{DTIME}(e_c)$ and $e_c(n) = 2^{cn}$.

Exercise 4.6 Prove that the two definitions of NTIME, presented in §4.2.1.3, are related up to logarithmic factors. Note the importance of condition that V has linear (rather than polynomial) time-complexity.

Guideline: When emulating a non-deterministic machine by the verification procedure V, encode the non-deterministic choices in y such that $|y|$ is slightly larger than the number of steps taken by the original machine. Specifically, having $|y| = O(t \log t)$, where t denotes the number of steps taken by the original machine, allows to emulate the latter in linear time (i.e., linear in $|y|$).

Exercise 4.7 (linear speed-up of Turing machine) 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 $t'(n) = O(n) + (t(n)/2)$.

Guideline: Consider a machine that uses a larger alphabet, capable of encoding a constant (denoted c) number of symbols of the original machine, and thus capable of emulating c steps of the original machine in $O(1)$ steps, where the constant in the O-notation is a universal constant (independent of c). Note that the $O(n)$ term accounts to a preprocessing that converts the binary input to work-alphabet of the new machine (which encoding c input bits in one alphabet symbol). Thus, a similar result for one-tape Turing machine seems to require a $O(n^2)$ term.

Exercise 4.8 In continuation to Exercise 4.7, state and prove an analogous result for space complexity, when using the standard definition of space as recalled in Section 4.3. (Note that this result does not hold with respect to "binary space complexity" as defined in Section 5.1.1.)

Exercise 4.9 Prove Theorem 4.9. As a warm-up, assume that s_1 (rather than s_2) is space constructible.

Guideline: Note that providing a space-efficient emulation of one machine by another machine is easier than providing an analogous time-efficient emulation.