JOURNAL OF COMPUTER AND SYSTEM SCIENCES 44, 168–192 (1992)

The Complexity Types of Computable Sets

WOLFGANG MAASS*

Institutes for Information Processing, Graz University of Technology, A-8010 Graz, Austria

AND

THEODORE A. SLAMAN†

Department of Mathematics, University of Chicago, Chicago, Illinois 60637

Received January 25, 1991

We analyze the fine structure of time complexity classes for RAMs, in particular the equivalence relation $A =_C B$ ("A and B have the same time complexity") $\Leftrightarrow$ (for all time constructible f: $A \in \mathrm{DTIME}(f) \Leftrightarrow B \in \mathrm{DTIME}(f)$). The $=_C$-equivalence class of A is called its *complexity type*. Characteristic sequences of time bounds are introduced as a technical tool for the analysis of complexity types. We investigate the relationship between the complexity type of a set and its polynomial time degree, as well as the structure of complexity types inside P with regard to linear time degrees. Furthermore, it is shown that every complexity type contains a sparse set and that the complexity types with their natural partial order form a lattice. © 1992 Academic Press, Inc.

1. INTRODUCTION

In the subsequent analysis of the fine structure of time complexity classes we consider the following set of time bounds:

$$T := \{f : \mathbf{N} \to \mathbf{N} \mid f(n) \geqslant n \text{ and } f \text{ is time constructible on a RAM}\}.$$

f is called time constructible on a RAM if some RAM can compute the function $1^n \mapsto 1^{f(n)}$ in $O(f(n))$ steps. We do not allow arbitrary recursive functions as time bounds in our approach in order to avoid pathological phenomena (e.g., gap theorems [HU; HH]). In this way we can focus on those aspects of complexity classes that are relevant for concrete complexity (note that all functions that are actually used as time bounds in the analysis of algorithms are time constructible).

* Written under partial support by NSF Grant CCR 8903398. Part of this research was carried out during a visit of the first author at the University of Chicago. The first author thanks the Department of Computer Science at the University of Chicago for its hospitality.

† Written under partial support by Presidential Young Investigator Award DMS-8451748 and NSF Grant DMS-8601856.

0022-0000/92 $3.00

We use the random access machine (RAM) with uniform cost criterion as machine model (see [CR; AHU; MY; P]) because this is the most frequently considered model in algorithm design, and because a RAM allows more sophisticated diagonalization constructions than a Turing machine. It does not matter for the following which of the common versions of the RAM model with instructions for ADDITION and SUBTRACTION of integers is chosen (note that it is common to exclude MULTIPLICATION of integers from the instruction set in order to ensure that the computation time of the RAM is polynomially related to time on a Turing machine). In order to be specific we consider the RAM model as it was defined by Cook and Reckhow [CR] (we use the common "uniform cost criterion" [AHU], i.e., $l(n) = 1$ in the notation of [CR]). This model consists of a finite program, an infinite array $X_0, X_1, \ldots$ of registers (each capable of holding an arbitrary integer), and separate one-way input and output tapes. The program consists of instructions for ADDITION and SUBTRACTION of two register contents, the conditional jump "TRA m if $X_i > 0$" which causes control to be transferred to line m of the program if the current content of register X_i is positive, instructions for the transfer of register contents with indirect addressing, instructions for storing a constant, and the instruction "READ X_i" (transfer the content of the next input cell on the input tape to register X_i) and "PRINT X_i" (print the content of register X_i on the next cell of the output tape). The relationship between computation time on RAMs and Turing machines is discussed in [CR, Theorem 2; AHU, Section 1.7; and P, Chap. 3]. It is obvious that a multi-tape Turing machine of time complexity $t(n)$ can be simulated by a RAM of time complexity $O(t(n))$. With a little bit more work (see [P]) one can construct a simulating RAM of time complexity $O(n + (t(n)/\log t(n)))$ (assuming that the output has length $O(n + (t(n)/\log t(n)))$). We define

$$\mathrm{DTIME}(f) := \left\{ A \subseteq \{0,1\}^* \;\middle|\; \begin{array}{l}\text{there is a RAM of time complexity} \\ O(f) \text{ that computes } A\end{array} \right\}.$$

For A and B contained in $\{0,1\}^*$ we define

$$A \leqslant_C B :\Leftrightarrow \forall f \in T (B \in \mathrm{DTIME}(f) \Rightarrow A \in \mathrm{DTIME}(f))$$

and

$$A =_C B :\Leftrightarrow \forall f \in T (A \in \mathrm{DTIME}(f) \Leftrightarrow B \in \mathrm{DTIME}(f)).$$

Intuitively, $A =_C B$ if A and B have the same deterministic time complexity. The $=_C$-equivalence classes are called *complexity types*. We write $\mathcal{O}$ for $\mathrm{DTIME}(n)$, which is the smallest complexity type. Note that for every complexity type $\mathscr{C}$ and every $f \in T$ one has either $\mathscr{C} \subseteq \mathrm{DTIME}(f)$ or $\mathscr{C} \cap \mathrm{DTIME}(f) = \varnothing$.

Remark. Geske, Huynh, and Selman [GHS] have considered the related partial order "the complexity of A is polynomially related to the complexity of B," and the associated equivalence classes ("polynomial complexity degrees"). Our results for complexity types (e.g., Theorems 5 and 6 below) provide corresponding results also for polynomial complexity degrees.

In order to prove results about the structure of complexity types one needs a technique to construct a set within a given time complexity while simultaneously controlling its other properties. It is less difficult to ensure that a constructed set is of complexity type $\mathscr{C}$ if one can associate with $\mathscr{C}$ an "optimal" time bound $f_{\mathscr{C}} \in T$ such that for all sets $X \in \mathscr{C}$,

$$\{f \in T \mid X \in \mathrm{DTIME}(f)\} = \{f \in T \mid f = \Omega(f_{\mathscr{C}})\}.$$

In this case, we call $\mathscr{C}$ a *principal* complexity type. Blum's speedup theorem [B] asserts that there are complexity types that are not principal. For example, there is a complexity type $\mathscr{C}$ such that for every $X \in \mathscr{C}$,

$$\{f \in T \mid X \in \mathrm{DTIME}(f)\} = \left\{f \in T \mid \exists i \in \mathbf{N}\left(f(n) = \Omega\left(\frac{n^2}{(\log n)^i}\right)\right)\right\}.$$

Note that this effect occurs even if one is only interested in time constructible time bounds (and sets X of "low" complexity).

In order to prove our results also for complexity types which are non-principal, we show that in some sense the situation of Blum's speedup theorem (where we can characterize the functions f with $X \in \mathrm{DTIME}(f)$ with the help of a "cofinal" sequence of functions) is already the worst that can happen (unfortunately this is not quite true, since we cannot always obtain a cofinal sequence of functions f_i with the same nice properties as in Blum's speedup theorem). More precisely: we will show that each complexity type can be characterized by a cofinal sequence of time bounds with the following properties.

DEFINITION. $(t_i)_{i \in \mathbf{N}} \subseteq \mathbf{N}$ is called a *characteristic sequence* if $t\colon i \mapsto t_i$ is recursive and

(1) $\forall i \in \mathbf{N}(\{t_i\} \in T$ and program t_i is a witness for the time-constructibility of $\{t_i\})$;

(2) $\forall i, n \in \mathbf{N}(\{t_{i+1}\}(n) \leqslant \{t_i\}(n))$.

DEFINITION. Assume that $A \subseteq \{0, 1\}^*$ is an arbitrary set and $\mathscr{C}$ is an arbitrary complexity type. One says that $(t_i)_{i \in \mathbf{N}}$ is *characteristic for A* if $(t_i)_{i \in \mathbf{N}}$ is a characteristic sequence and

$$\forall f \in T(A \in \mathrm{DTIME}(f) \Leftrightarrow \exists i \in \mathbf{N}(f(n) = \Omega(\{t_i\}(n)))).$$

One says that $(t_i)_{i \in \mathbf{N}}$ is *characteristic for $\mathscr{C}$* if $(t_i)_{i \in \mathbf{N}}$ is characteristic for some $A \in \mathscr{C}$ (or equivalently: for all $A \in \mathscr{C}$).

Remark. The idea of characterizing the complexity of a recursive set by a sequence of "cofinal" complexity bounds is rather old (see, e.g., [MF; L; Ly; SS; MW]). However, none of our predecessors exactly characterized the time complexity of a recursive set in terms of a uniform cofinal sequence of time constructible time bounds. This is the form of characterization which we employ in later proofs. [L; MW] give corresponding results for space complexity of Turing machines. Their results exploit the linear speedup theorem for space complexity on Turing machines, which is not available for time complexity on RAMs. Time complexity on RAMs has been considered in [SS], but only sufficient conditions are given for the cofinal sequence of time bounds (these conditions are stronger than ours). The more general results on complexity sequences in axiomatic complexity theory [MF; SS] involve "overhead functions," or deal with nonuniform sequences, which makes the specialization to the notions considered here impossible. Because of the lack of a fine time hierarchy theorem for multi-tape Turing machines, it is an open problem whether one can give a similar characterization for the Turing machine time complexity of a recursive set.

The relatonship between complexity types and characteristic sequences is clarified in Section 2 of this paper. In Theorem 1, we show that every complexity type $\mathscr{C}$ is associated with a sequence $(t_i)_{i\in\mathbf{N}}$ that is characteristic for $\mathscr{C}$. This fact will be used in most of the subsequent results. We show in Theorem 2 that the converse of Theorem 1 is also true: for every characteristic sequence $(t_i)_{i\in\mathbf{N}}$ there exists a complexity type $\mathscr{C}$ such that $(t_i)_{i\in\mathbf{N}}$ is characteristic for $\mathscr{C}$.

As an immediate consequence of the proofs of these results we show in Theorem 3 that every complexity type contains a sparse set. As another consequence we obtain in Theorem 4 that the complexity types of computable sets form a lattice under the partial order $\leqslant_C$.

In Section 3 we show that with the help of these tools one is able to prove sharper versions of various familiar results about polynomial time degrees. It is shown in Theorem 5 that every complexity type outside of P contains sets of incomparable polynomial time Turing degree. In Theorem 6 we construct a complexity type that contains a minimal pair of polynomial time Turing degrees. A comparison of the proofs of these results with the proofs of the related results by Ladner [La] and Landweber, Lipton, and Robertson [LLR] shows that these sharper versions pose a more serious challenge to our construction techniques (we apply finite injury priority arguments and a constructive version of Cohen forcing).

In Section 4 we use the concepts and techniques that are introduced in this paper for an investigation of the fine structure of P. We show in Theorem 7 that in P each complexity type $\mathscr{C} \neq 0$ contains a rich structure of linear time degrees. In Theorem 8, we characterize those $\mathscr{C}$ that have a complete set under linear time many–one reductions. The proofs of these two theorems provide evidence that finite injury priority arguments are not only relevant for the investigation of sets "higher up," but also for the analysis of the finite structure of P. Finally, in Section 5 we list some open problems.

In this paper we give complete proofs for those results that had been reported earlier in the extended abstract [MS1]. Furthermore, we have added Theorem 4, as well as a list of open problems. Some recent results about complexity types are reported in [MS2].

2. The Relationship between Complexity Types and Characteristic Sequences

Theorem 1 (Inverse of the speedup theorem for time on RAMs). *For every recursive set A there exists a sequence $(t_i)_{i\in \mathbf{N}}$ that is characteristic for A.*

Proof. Fix e_0, t_0, and c_0 in $\mathbf{N}$ such that $\{e_0\} = A$; for all $n \in \mathbf{N}$, $n \leqslant \{t_0\}(n)$; for all $x \in \{0, 1\}^*$, $\{e_0\}(x)$ converges in $\leqslant \{t_0\}(|x|)$ steps; and for all $n \in \mathbf{N}$, $\{t_0\}(n)$ converges in $\leqslant c_0 \cdot \{t_0\}(n)$ steps.

For $i = \langle (i)_0, (i)_1, (i)_2 \rangle \in \mathbf{N}$ and $y \in \{0, 1\}^*$, we define $Q(i, y)$ to be the property defined by the following conditions:

(1) If $\{(i)_0\}(y)\downarrow$ then $\{(i)_0\}(y) = \{e_0\}(y)$.

(2) If $\{(i)_1\}(|y|)\downarrow$ then $\{(i)_0\}(y)$ converges in less than or equal to $\{(i)_1\}(|y|)$ steps and $\{(i)_1\}(|y|)$ converges in less than or equal $(i)_2 \cdot \{(i)_1\}(|y|)$ steps.

Note that $\forall y\, Q(i, y)$ does not imply that $\{(i)_0\}$ or $\{(i)_1\}$ are total functions.

One defines by recursion, functions $i \mapsto t_i$ and $i \mapsto c_i$ such that for all $n \in \mathbf{N}$, $\{t_i\}(n)$ converges in $\leqslant c_i \cdot \{t_i\}(n)$ steps. As a first approximation, one would like to define $\{t_i\}(n)$ by cases: If for every string y, $\{(i)_1\}(|y|)\downarrow$ and $Q(i, y))$ then define $\{t_i\}(n)$ to be $\max(n, \min(\{t_{i-1}\}(n), \{(i)_1\}(n)))$. Otherwise, define $\{t_i\}(n)$ to be $\{t_{i-1}\}(n)$.

Unfortunately, the required decision as to which case applies is not recursive. However, it is possible to replace the immediate definition by cases with a recursive "looking back" procedure. One defines t_i to be a program that acts on input n as follows: It first spends n steps to check whether "so far" it appears that $\forall y\, Q(i, y)$. More precisely one fixes a canonical order of all $y \in \{0, 1\}^*$ and sequentially in y verifies $Q(i, y)$. Thus, if there is some y with $\neg\, Q(i, y)$, then such y will be seen during the "lock back" for all sufficiently large inputs.

If t_i on input n finds during the "lock back" some $y \in \{0, 1\}^*$ for which $Q(i, y)$ does not hold, then t_i simulates t_{i-1} on input n and outputs $\{t_{i-1}\}(n)$.

Otherwise, t_i continues as follows with n. It simulates t_{i-1} and $(i)_1$ on input n. If $\{(i)_1\}(n) < \{t_{i-1}\}(n)$ and $\{(i)_1\}(n)$ converges in $\leqslant (i)_2 \cdot \{(i)_1\}(n)$ steps (both can be verified in $O(\{(i)_1\}(n))$ steps with the help of the constant c_{i-1}) then t_i outputs $\max(n, \{(i)_1\}(n))$. Otherwise it outputs $\{t_{i-1}\}(n)$. The constant c_i is defined from c_{i-1} and $(i)_2$ in a straightforward manner.

This finishes the recursive definition of the programs $(t_i)_{i\in\mathbf{N}}$ and the constants $(c_i)_{i\in\mathbf{N}}$. One can easily verify by induction on i that t_i is the program for a total recursive function, and that $\{t_i\}(n)$ converges in $\leqslant c_i \cdot \{t_i\}(n)$ steps with $n \leqslant \{t_i\}(n) \leqslant \{t_{i-1}\}(n)$ for all $n \in \mathbf{N}$. Hence, $(t_i)_{i \in \mathbf{N}}$ is a characteristic sequence.

In order to prove that $(t_i)_{i \in \mathbf{N}}$ is characteristic for A one first shows by induction on i that $A \in \mathrm{DTIME}(\{t_i\})$ for all $i \in \mathbf{N}$. Assume that $A = \{e_{i-1}\}$ and that $\{e_{i-1}\}(x)$ converges in $\leqslant d_i \cdot \{t_{i-1}\}(|x|)$ steps for all $x \in \{0, 1\}^*$. If $\exists y \neg Q(i, y)$ then $\{t_i\}(n) = \{t_{i-1}\}(n)$ for almost all $n \in \mathbf{N}$, and nothing remains to be shown. Assume then that $\forall y\, Q(i, y)$. We define e_i to be a program that simulates for every input $x \in \{0, 1\}^*$ both $\{e_{i-1}\}(y)$ and $\{(i)_0\}(y)$. Program e_i outputs the result of that computation which converges first (actually if $\{(i)_0\}(y)\downarrow$ then $\{(i)_0\}(y) = \{e_{i-1}\}(y)$). Note that by the definition of Q if $\{(i)_1\}(n)\downarrow$ then $\{(i)_0\}(y)$ converges for all $y \in \{0, 1\}^n$. Therefore $\{e_i\} = A$ and $\{e_i\}$ is of time complexity $O(\min(\{t_{i-1}\}(n), \{(i)_1\}(n))) = O(\{t_i\}(n))$ (we set $\min(\{t_{i-1}\}(n), \{(i)_1\}(n) := \{t_{i-1}\}(n)$ if $\{(i)_1\}(n)\uparrow$).

Finally, assume that e, t, $c \in \mathbf{N}$ are given such that $\{e\} = A$, $\{t\} \in T$, $\{e\}(x)$ converges in $\leqslant \{t\}(|x|)$ steps for all $x \in \{0, 1\}^*$, and $\{t\}(n)$ converges in $\leqslant c \cdot \{t\}(n)$ steps for all $n \in \mathbf{N}$. Set $i := \langle e, t, c \rangle$. Then $\forall y\, Q(i, y)$ and $\{t_i\}(n) \leqslant \{t\}(n)$ for all $n \in \mathbf{N}$ (by the definition of t_i). Thus $\{t\} = \Omega(\{t_i\})$. ▌

In fact, we can conclude the following stronger version of Theorem 1. Let $A =^* B$ denote the condition that the symmetric difference of A and B is finite.

COROLLARY. *For every recursive set A there exist recursive sequences $(t_i)_{i \in \mathbf{N}}$, $(c_i)_{i \in \mathbf{N}}$, $(d_i)_{i \in \mathbf{N}}$, $(e_i)_{i \in \mathbf{N}}$ such that*

(1) *$(t_i)_{i \in \mathbf{N}}$ is characteristic for A*

(2) *$\{t_i\}(n)$ converges in $\leqslant c_i \cdot \{t_i\}(n)$ steps for all i, $n \in \mathbf{N}$*

(3) *$\{e_i\} =^* A$ and $\{e_i\}(x)$ converges in $\leqslant d_i \cdot \{t_i\}(|x|)$ steps, for all $i \in \mathbf{N}$, $x \in \{0, 1\}^*$.*

Proof. The four sequences are defined simultaneously by recursion. One defines e_i as a program that proceeds for any input $x \in \{0, 1\}^n$ as follows: First e_i spends n steps to check whether "so far" it appears that $\forall y\, Q(i, y)$ (see program t_i in the proof of Theorem 1). If it finds during this "looking back" some y with $\neg Q(i, y)$ then e_i simulates e_{i-1} on input x. Otherwise e_i simulates simultaneously e_{i-1} and $(i)_0$ on input x, and the program $(i)_1$ on input $n = |x|$. e_i outputs the result of that one of the computations $\{e_{i-1}\}(x)$, $\{(i_1)_0\}(x)$ that converges first; except in the case where $\{(i)_1\}(n)$ converges faster than any of these two computations, in which case $\{e_i\}(x)$ gives output 0 (note that in this case $\{(i)_0\}(x)$ does not converge in $\leqslant \{(i)_1\}(|x|)$ steps, thus $\neg Q(i, x)$).

The program t_i and the constant c_i are defined in the same way as in the proof of Theorem 1. We define $d_i := 10 \cdot \max(1 + d_{i-1}, 1 + (i)_2)$.

The claimed properties of t_i and c_i have been verified in the proof of Theorem 1. It remains to be shown (by induction on i) that $\{e_i\} =^* A$ and $\{e_i\}(x)$ converges in $\leqslant d_i \cdot \{t_i\}(|x|)$ steps for all $i \in \mathbf{N}$, $x \in \{0, 1\}^*$.

If there exists some $x \in \{0, 1\}^*$ s.t. $\{e_i\}(x) \neq \{e_{i-1}\}(x)$ and $\{e_i\}(x) \neq A(x)$ then $\neg Q(i, x)$ holds, thus $\{e_i\} =^* \{e_{i-1}\}$. If there is no such x one also has $\{e_i\} =^* \{e_{i-1}\}$.

In every case the number of computation steps of e_i on input x can be bounded by $10 \cdot (|x| + (\text{number of computation steps of } \{e_{i-1}\}(x))) \leqslant 10 \cdot (1 + d_{i-1}) \cdot \{t_{i-1}\}(|x|)$. Furthermore, if $\{t_i\}(|x|) < \{t_{i-1}\}(|x|)$ then $\{(i)_1\}(|x|) < \{t_{i-1}\}(|x|)$, $\{(i)_1\}(|x|)$ converges in $\leqslant (i)_2 \cdot \{(i)_1\}(|x|)$ steps, and $\{t_i\}(|x|) = \max(|x|, \{(i)_1\}(|x|))$ (by the definition of t_i). In this case the computation of $\{e_i\}(x)$ takes no more than

$$|x| + 10 \cdot (i)_2 \cdot \{(i)_1\}(|x|) \leqslant 10 \cdot (1 + (i)_2) \cdot \{t_i\}(|x|)$$

steps. ▌

THEOREM 2 (Refinement of the speedup theorem for time on RAMs). *For every characteristic sequence $(t_i)_{i \in \mathbf{N}}$ there is a set A such that $(t_i)_{i \in \mathbf{N}}$ is characteristic for A.*

Proof. Fix an arbitrary sequence $(t_i)_{i \in \mathbf{N}}$ and a sequence $(K_i)_{i \in \mathbf{N}}$ of numbers such that $\{t_i\}(n)$ converges in $\leqslant K_i \cdot \{t_i\}(n)$ steps, for all i, $n \in \mathbf{N}$.

The claim of the theorem does not follow from any of the customary versions of Blum's speedup theorem [B], because it need not be the case that $\{t_i\} = o(\{t_{i-1}\})$. In fact it may occur that $\{t_i\} = \{t_{i-1}\}$ for many (or even for all) $i \in \mathbf{N}$. Even worse, one may have that $K_i \cdot \{t_i\}(n) > K_{i-1} \cdot \{t_{i-1}\}(n)$ for many i, $n \in \mathbf{N}$ (this may occur, for example, in the characteristic sequences that arise from the construction of Theorem 1 if i does not encode a "faster" algorithms $(i)_0$ of time complexity $\{(i)_1\}(n) < \{t_{i-1}\}(n)$; in this case one may have that $\{t_i\}(n) = \{t_{i-1}\}(n)$ and the computation of $\{t_i\}(n)$ takes longer than the computation of $\{t_{i-1}\}(n)$ because the former involves simulations of both $\{t_{i-1}\}(n)$ and $\{(i)_1\}(n)$). Therefore it is more difficult than in Blum's speedup theorem to ensure in the following proof that the constructed set A satisfies $A \in \mathrm{DTIME}(\{t_i\})$ for every $i \in \mathbf{N}$. To achieve $A \in \mathrm{DTIME}(\{t_i\})$ it is no longer enough to halt for all $j \geqslant i$ the attempts towards making $A(x) \neq \{j\}(x)$ after $\{t_j\}(|x|)$ steps, because the value of $\{t_j\}(|x|)$ may only be known after $K_j \cdot \{t_j\}(|x|)$ steps and it is possible that

$$\lim_{j, n \to \infty} \frac{K_j \cdot \{t_j\}(n)}{\{t_i\}(n)} = \infty.$$

Instead, in the following construction one simulates, together with $\{j\}(x)$ and $\{t_j\}(|x|)$, also the computations $\{t_0\}(|x|), \ldots, \{t_{j-1}\}(|x|)$; that one of these $j+2$ computations which converges first will halt the computation of $A(x)$. It is somewhat delicate to implement these simultaneous simulations of $j+2$ computations (where j grows with $|x|$) in such a way that for each fixed $i \in \mathbf{N}$ the *portion* of the total computation time for $A(x)$ that is devoted to the simulation of $\{t_i\}(|x|)$ does not shrink when j grows to infinity (obviously this property is needed to prove that $A \in \mathrm{DTIME}(\{t_i\})$).

In the following we construct a RAM R that computes a set A for which $(t_i)_{i \in \mathbf{N}}$

is a characteristic sequence. We fix some coding of RAMs by binary strings analogously as in [CR]. We arrange that each code for a RAM is a binary string without leading zeros, so that one can also view it as binary notation for some number $j \in \mathbf{N}$. We assume that the empty string codes the "empty" RAM (which has no instructions). Thus one can associate with each binary string x the longest initial segment of x that is a code for a RAM, which will be denoted by j_x in the following.

The RAM R with input $x \in \{0, 1\}^*$ acts as follows. It first checks via "looking back" for $|x|$ steps whether the requirement "$A \neq \{j\}$" for $j := j_x$ has already been satisfied during the first $|x|$ steps of the construction (for shorter inputs). If the answer is "yes," R decides without any further computation that $x \notin A$. If the answer is "no," R on input x dovetails the simulations of $t_0, ..., t_j$ on input $|x|$ and of j on input x in the following manner. The simultaneous simulation proceeds in phases $p = 0, 1, 2, ...$ At the beginning of each phase p the RAM R simulates one more step of j on input x. If m_j is the number of steps that R has used to simulate this step of j on input x, then R simulates subsequently m_j steps of t_j on input $|x|$. If m_{j-1} is the number of steps that R has used so far in phase p (for the simulation of one step of j on x and m_j steps of t_j on $|x|$), then R simulates subsequently m_{j-1} steps of t_{j-1} on input $|x|$, etc. If the number of steps that R spends in phase p exceeds $|x|$, R immediately halts and rejects x.

If R has finished phase p (by simulating m_0 steps of t_0 on input $|x|$, where m_0 is the total number of steps which R has spent in phase p for the simulation of $t_1, .., t_j$ on $|x|$ and j on x) and none of the $j+2$ simulated computations has reached a halting state during phase p, then R proceeds to phase $p+1$.

If the computation of j on input x has converged during phase p then R puts x into A if and only if $\{j\}(x) = 0$. With this action the requirement "$A \neq \{j\}$" becomes satisfied and the computation of R on input x is finished.

If j on input x has not converged during phase p, but one of the other $j+1$ simulated computations (of $t_0, ..., t_j$ on $|x|$) converges during phase p, then R decides that $x \notin A$ and halts.

We now describe some further details of the program of R in order to make a precise time analysis possible. In order to avoid undesirable interactions R reserves for each simulated RAM a separate infinite sequence ARRAY$_k$ of registers, where ARRAY$_k$ consists of the registers

$$X_{2^{k+1} \cdot (2j+1)}, \qquad j = 0, 1, 2, ...$$

Furthermore, we specify for each $k \in \mathbf{N}$ a sub-array MEMORY$_k$ of ARRAY$_k$, consisting of the registers $X_{2^{k+1} \cdot (2j+1)}$ for $j = 0, 2, 4, ...$ in ARRAY$_k$. For each $k \in \mathbf{N}$ the constructed RAM R uses ARRAY$_k$ for the simulation of the RAM coded by t_k (more precisely, R uses MEMORY$_k$ to simulate the registers of the RAM t_k, and it uses the remaining registers in ARRAY$_k$ to store the program that is coded by t_k). Furthermore, R uses ARRAY$_{j+1}$ to simulate the RAM coded by $j = j_x \in \mathbf{N}$ (this does not cause a conflict since j and t_{j+1} are never simultaneously simulated).

We have made these arrays explicit because in order to simulate a step of the kth one of these machines that involves, say, the ith register of that machine, R has to compute on the side (via iterated additions) the "real" address $2^{k+1} \cdot (2 \cdot 2i + 1)$ of the corresponding register in MEMORY$_k$. The number of steps required by this side-computation depends on k, but is independent of i. Therefore there exist constants $c(k)$, $k \in \mathbf{N}$, that bound the number of steps that R has to spend to simulate a single step of the kth one of these machines. It is essential that for any fixed $k \leqslant j_x + 1$ this constant $c(k)$ is independent of the number $j_x + 2$ of simulated machines for input x (this follows from the fact that ARRAY$_k$ is independent of j_x).

At the beginning of its computation on input x the RAM R computes $j := j_x$. Then R "looks back" for $|x|$ steps to check whether "$A \neq \{j\}$" has already been satisfied at some argument x' such that j_x is a prefix of x' and x' is a prefix of x. Let $x_1, ..., x_k$ be a list of all such binary strings x' (ordered according to their length). We implement the "looking back" for argument x as follows. For each of the shorter inputs $x_1, x_2, ...$ (one after the other) R carries out the same computation as discussed below for x, but without any "looking back" procedure. If it turns out after $|x|$ steps of this subcomposition that for one of these arguments x_i ($i \in \{1, ..., k\}$) the construction satisfied the requirement $A \neq \{j\}$, then R immediately halts and rejects x (note that for the first such x_i we actually have then $A(x_i) \neq \{j\}(x_i)$). Otherwise R continues the computation on input x as follows: R calls a program for the recursive function $i \mapsto t_i$ and uses it to compute the numbers $t_0, ..., t_j$. Afterwards R "decodes" each of the programs t_i ($i \in \{0, ..., j\}$), which are given as binary string (the binary representations of the number $t_i \in \mathbf{N}$). R uses for each instruction S of program t_i four registers in ARRAY$_i$ (that are not in MEMORY$_i$) to store the opcode and the up to three operands of S (similarly as in the proof of Theorem 3 in [CR]). In the same way R "decodes" the program j.

If this preprocessing phase of R on input x takes more than $100 \cdot |x|$ steps (in addition to the steps spend on "looking back") then R immediately halts and rejects x.

During the main part of its computation (while it simulates $t_0, ..., t_j, j$) R maintains in its odd-numbered registers (which do not belong to any of the arrays ARRAY$_k$), two counters that count the total number of steps that have been spent so far in the current phase p, as well as the number of steps of the currently considered program that have already been simulated during phase p. In order to allow enough time for the regular updating of both counters, as well as for the regular comparison of the values of both counters with the preset thresholds ($|x|, m_i$), it is convenient to add every c steps the number c to each counter (where $c \in \mathbf{N}$ is a sufficiently large constant in the program of R).

In addition, R stores in its odd-numbered registers the input x, j_x, the number of the currently simulated program, and for each of the simulated programs the address of the register that contains the opcode for the next instruction that has to be executed for that program. With this information R can resume the simulation of an earlier started program without any further "overhead steps" because each simulated program only acts on the registers in its "own" array ARRAY$_k$. Of course, R always has to spend several steps to simulate a single instruction of any of the stored programs $t_0, \ldots, t_j, j$. It has to apply a series of branching instructions in order to go from the stored (numerical) value of the opcode in ARRAY$_k$ to the actual instruction in its own program that corresponds to it, and it has to calculate the "real" addresses of the involved registers in MEMORY$_k$. However, it is obvious that for each simulated program k there exists a constant $c(k)$ (independent from $j = j_x$ and x) that bounds the number of steps that R has to spend to simulate a single instruction of program k.

We now verify that $(t_i)_{i \in \mathbf{N}}$ is characteristic for A.

CLAIM 1. *$A \in \mathrm{DTIME}(\{t_i\})$ for every $i \in \mathbf{N}$.*

Proof. Fix i and set $S_i := \{x \in \{0, 1\}^* \mid j_x < i\}$. Every $\tilde{x} \in S_i \cap A$ is placed into A in order to satisfy the requirement "$A \neq \{j\}$" for some $j < i$. We then have $A(\tilde{x}) \neq \{j\}(\tilde{x})$, and for sufficiently longer inputs x with $j_x = j$ the constructed RAM R finds out during the first part of its computation on input x (while "looking back" for $|x|$ steps) that the requirement "$A \neq \{j\}$" has already been satisfied. This implies that $x \notin A$. Thus for each $j < i$ only finitely many $\tilde{x}$ are placed into A in order to satisfy the requirement "$A \neq \{j\}$." Therefore $S_i \cap A$ is finite. Since $S_i \in \mathrm{DTIME}(n)$ it only remains to prove that $\bar{S}_i \cap A \in \mathrm{DTIME}(\{t_i\})$.

We show that R uses for every input $x \in \bar{S}_i$ at most $O(\{t_i\}(|x|))$ computation steps. By assumption we have $\{t_i\}(|x|) \geqslant |x|$, and therefore R uses only $O(\{t_i\}(|x|))$ steps in its preprocessing phase for input x.

We had fixed constants $K_i \in \mathbf{N}$ such that the computation of program t_i on input n consists of $\leqslant K_i \cdot \{t_i\}(n)$ steps (for all $n \in \mathbf{N}$). By construction the total number of steps that R spends for input x on the simulation of $j := j_x$ on x and of $t_{i+1}, \ldots, t_j$ on $|x|$ is bounded by $|x| + T_{i,x}$, where $T_{i,x}$ is the number of computation steps of program t_i on input $|x|$ that are simulated by R on input x. By construction we have $T_{i,x} \leqslant K_i \cdot \{t_i\}(|x|) + |x|$ (because the computation of R on x is halted at the latest at the end of the first phase p, where $\{t_i\}(|x|)$ is seen to converge, and no phase p consists of more than $|x|$ steps of R).

Furthermore, we know that there is a constant $c(t_i)$ that bounds the number of steps that R needs to simulate a single instruction of t_i. Thus R spends $\leqslant c(t_i) \cdot T_{i,x} = O(\{t_i\}(|x|))$ steps on the simulation of $t_{i+1}, \ldots, t_j$ on $|x|$ and j on x. Furthermore, the number of "overhead steps" of R for the updating of counters, the comparison of their values with preset thresholds, and the switching of programs can be bounded by a constant times the number of steps that R spends on the actual simulations. Thus it just remains to be shown by induction on $i - k$ that R on input $x \in \bar{S}_i$ spends for every $k < i$ altogether at most $O(\{t_i\}(|x|))$ steps on the simulation of t_k on $|x|$. However, this follows immediately from the construction, using the definition of the parameters m_k and the observation that the constants $c(t_k)$ do not depend on x. ∎

CLAIM 2. *Let $U(j, n)$ be the maximal number of steps that program j uses on an input of length n. Then*

$$A = \{j\} \Rightarrow U(j, n) = \Omega(\{t_j\}(n)).$$

Proof. Fix any $j \in \mathbf{N}$ such that $A = \{j\}$. Assume for a contradiction that it is not the case that $U(j, n) = \Omega(\{t_j\}(n))$, i.e., we have $\sup_n (\{t_j\}(n)/U(j, n)) = \infty$. We show that then the requirement "$A \neq \{j\}$" is satisfied at some argument x with $j_x = j$.

Obviously we have for all sufficiently long x with $j_x = j$ that the computation of R on x is not halted prematurely because the preprocessing phase takes too many steps, or because a phase p in the main part of the computation requires more than $|x|$ steps. Furthermore, for each $i \leqslant j$ there exists by construction a constant c_i such that, for all x with $j_x = j$, R simulates for each step in the computation of j on x at most c_i steps in the computation of t_i on input $|x|$. Therefore, our assumption $\sup_n (\{t_j\}(n)/U(j, n)) = \infty$, together with the fact that $\{t_0\}(n) \geqslant \cdots \geqslant \{t_j\}(n)$ for all $n \in \mathbf{N}$, implies that there is some x with $j_x = j$ such that R on input x does not halt prematurely because some $\{t_i\}(|x|)$ with $i \leqslant j$ is seen to converge before $\{j\}(x)$ is seen to converge. For such input x the RAM R succeeds in satisfying the requirement "$A \neq \{j\}$" by setting $A(x) \neq \{j\}(x)$. This contradiction completes the proof of Claim 2. ▌

Claim 1 and Claim 2 together imply the claim of the theorem. ▌

As an immediate consequence we obtain from the preceding two theorems the following result (recall that $S \subseteq \{0, 1\}^*$ is called *sparse* if there is a polynomial p such that $\forall n(|\{x \in S|\ |x| = n\}| \leqslant p(n))$; see [Ma] for a contrasting result about sparse sets):

THEOREM 3. *Every complexity type contains a sparse set.*

Proof. Let $\mathscr{C}$ be an arbitrary complexity type. By Theorem 1 there exists some sequence $(t_i)_{i \in \mathbf{N}}$ which is characteristic for $\mathscr{C}$. In order to obtain a sparse set S such that $(t_i)_{i \in \mathbf{N}}$ is characteristic for S we use a variation of the proof of Theorem 2. In this variation of the construction one never places x into the constructed set unless $|j_x| \leqslant \log |x|$. ▌

Remarks.

(1) It is an open problem whether every complexity type contains a tally set.

(2) Further results about the relationship between extensional properties of a set and its complexity type can be found in [MS2].

It is obvious that the partial order $\leqslant_C$ on sets (which was defined in Section 1) induces a partial order $\leqslant_C$ on complexity types. In this paper we are not concerned with the structure of this partial order $\leqslant_C$; however, we want to mention the following immediate consequence of Theorems 1 and 2.

THEOREM 4. *The complexity types of computable sets with the partial order $\leqslant_C$ form a lattice. Furthermore, if the characteristic sequence $(t_i')_{i \in \mathbf{N}}$ is characteristic for the complexity type $\mathscr{C}'$ and the characteristic sequence $(t_i'')_{i \in \mathbf{N}}$ is characteristic for the complexity type $\mathscr{C}''$, then the sequence $(t_i^{\min})_{i \in \mathbf{N}}$ with $\{t_i^{\min}(n)\} = \min(\{t_i'\}(n), \{t_i''\}(n))$ is characteristic for the infimum $\mathscr{C}' \vee \mathscr{C}''$ of $\mathscr{C}'$, $\mathscr{C}''$, and the sequence*

$(t_i^{\max})_{i\in\mathbf{N}}$ *with* $\{t_i^{\max}\}(n)=\max(\{t_i'\}(n), \{t_i''\}(n))$ *is characteristic for the supremum* $\mathscr{C}' \vee \mathscr{C}''$ *of* $\mathscr{C}', \mathscr{C}''$.

Proof. The key fact for the proof is the following elementary observation: Suppose that two sets T and S are given to be recursive subsets of $\{0, 1\}^*$. Let $(t_i)_{i\in\mathbf{N}}$ and $(s_i)_{i\in\mathbf{N}}$ be their characteristic sequences. Then

$$T \geqslant_C S \Leftrightarrow \forall i\, \exists j\, (\{t_i\} = \Omega(\{s_j\})).$$

In order to apply this observation and prove the claim of the theorem one first has to verify that one can find programs $t_i^{\min}$, $t_i^{\max}$ for $\min(\{t_i'\}, \{t_i''\})$, respectively $\max(\{t_i'\}, \{t_i''\})$, so that $(t_i^{\min})_{i\in\mathbf{N}}$ and $(t_i^{\max})_{i\in\mathbf{N}}$ are characteristic sequences. The only nontrivial point is the requirement to define the recursive function $i \mapsto t_i^{\min}$ in such a way that for all $i\in\mathbf{N}$, $\{t^{\min}\}(n)=\min(\{t_i'\}(n), \{t_i''\}(n))$ **and** $t_i^{\min}$ is a witness for the time constructibility of this function. In order to achieve this, it is essential that program $t_i^{\min}$ "knows" time constructibility factors c_i', c_i'' such that $\{t_i'\}(n)$ converges in $\leqslant c_i' \cdot \{t_i'\}(n)$ steps and $\{t_i''\}(n)$ converges in $\leqslant c_i'' \cdot \{t_i''\}(n)$ steps. Since program $t_i^{\min}$ has to compute $\min(\{t_i'\}(n), \{t_i''\}(n))$ in a time constructible fashion, it needs c_i', c_i'' in order to know when it is "safe" to abandon the simulation of the longer one of the two computations $\{t_i'\}(n)$, $\{t_i''\}(n)$ (after the shorter one has converged). But this is no problem, since the proof of Theorem 1 shows (as in the corollary to Theorem 1) that one can assume without loss of generality that recursive sequences $(c_i')_{i\in\mathbf{N}}$, $(c_i'')_{i\in\mathbf{N}}$ of time constructibility factors are given together with the characteristic sequences $(t_i')_{i\in\mathbf{N}}$ and $(t_i'')_{i\in\mathbf{N}}$.

Since $(t_i^{\min})_{i\in\mathbf{N}}$ is a characteristic sequence, there exists by Theorem 2 a set $T^{\min} \subseteq \{0, 1\}^*$ such that $(t_i^{\min})_{i\in\mathbf{N}}$ is characteristic for $T^{\min}$. Let $\mathscr{C}^{\min}$ be the complexity type of $T^{\min}$. It is obvious that $\mathscr{C}^{\min} \leqslant_C \mathscr{C}'$ and $\mathscr{C}^{\min} \leqslant_C \mathscr{C}''$. In order to show that $\mathscr{C}^{\min}$ is the infimum of $\mathscr{C}'$ and $\mathscr{C}''$, one has to verify for an arbitrary complexity type $\mathscr{C}$ with $\mathscr{C} \leqslant_C \mathscr{C}'$ and $\mathscr{C} \leqslant_C \mathscr{C}''$ that $\mathscr{C}^{\min} \geqslant_C \mathscr{C}$. Let the sequence $(t_i)_{i\in\mathbf{N}}$ be characteristic for $\mathscr{C}$ ($(t_i)_{i\in\mathbf{N}}$ exists by Theorem 1). The key fact at the beginning of this proof implies that $\forall i\, \exists j\, (\{t_i'\} = \Omega(\{t_j\}))$ and $\forall i\, \exists j\, (\{t_i''\} = \Omega(\{t_j\}))$. Hence $\forall i\, \exists j\, (\{t_i^{\min}\} = \Omega(\{t_j\}))$, which implies that $\mathscr{C}^{\min} \geqslant_C \mathscr{C}$.

One verifies analogously that the complexity type $\mathscr{C}^{\max}$ which is defined by $(t_i^{\max})_{i\in\mathbf{N}}$ is the supremum of $\mathscr{C}', \mathscr{C}''$ with regard to $\leqslant_C$. ∎

3. Time Complexity versus Polynomial Time Reducibility

In this section we study the relationship between the complexity type of a set and its polynomial time Turing degree. We first introduce the customary recursion theoretic vocabulary for the discussion of priority constructions. One important part of the following constructions is the construction technique of Theorem 2, which allows us to control the complexity type of a set that is constructed to meet various other requirements. Therefore we will review briefly the construction of Theorem 2 in recursion theoretic terms.

A *stage* is just an integer s viewed in the context of a definition by recursion. A *strategy* is just an algorithm to determine the action taken during stage n, recursively based on various parameters of the construction. Typically, a strategy is used to show that the sets being constructed have some specific property; we call this a *requirement*. We organize our attempts to satisfy the requirements in some order which we call the *priority ordering*. If requirement Q comes before requirement R, we say that Q has *higher priority* and R has *lower priority*.

In the construction of Theorem 2 we built A to satisfy two families of requirements. For each i, A had to be computable on a RAM, running in time $O(\{t_i\})$. Second, any RAM that computed A had to operate with time $\Omega(\{t_i\})$, for some i. We assigned priority by interleaving the two types of requirements in order of their indices.

For an element of the first family of requirements, we used a strategy which imposed a sequence of time controls on the construction. The behavior of the ith strategy was to terminate all action of lower priority in determining $A(x)$ once the construction had executed sufficiently many phases to exceed $\{t_i\}(|x|)$ many steps. Suppose that strategies of higher priority only act finitely often to cause A to accept strings. Then, using the finite data describing the higher priority activity we can correctly compute A at x in time $O(\{t_i\})$ by first checking the data for an answer at x. If the value of A at x is not included in the data then we run the construction until the ith time control strategy calls a halt to lower priority activity. We then read off the answer.

For each of the second family of requirements, we used a diagonalization strategy. Namely, if $\{i\}$ ever converges at an argument x before a strategy of higher priority terminates our action then we define A at x to make $A(x) \neq \{i\}(x)$. By our association of at most one diagonalization strategy to each string, the two possibilities were for $\{i\}$ to disagree with A or to have running time $\Omega(\{t_i\})$. The latter being the case when diagonalization is impossible, since the time control associated with higher priority strategies terminates the diagonalization attempt at every string.

We collectively refer to the time control and diagonalization strategies as the *$\mathscr{C}$-strategies.*

The proof that the sequence $(t_j)_{i \in \mathbb{N}}$ is characteristic for A has two essential features. The first is that each of the constituent strategies is injured finitely often. The jth time control strategy s_j is injured when the value of A at x is determined differently from the one assumed by s_j. In the proof of Theorem 2, this occurs when the value of A is determined by a diagonalization strategy with index less than j. The second feature, which occurs on a higher level, is that no single move in the construction prohibits the subsequent application and complete implementation of further time control and diagonalization strategies.

With this in mind, we can look for other families of strategies, which are compatible with the $\mathscr{C}$-strategies, to produce interesting examples within a given complexity type. Our next result compares complexity with relative computability.

A Turing reduction is given by a RAM M, augmented with the ability to query an oracle as to whether it accepts the query string. We evaluate M relative to A by answering all with the value of A on the query string. M specifies a polynomial time Turing reduction if there is a polynomial g such that for every oracle A and every string x, if the evaluation of M with input x halts in less than $g(|x|)$ steps. Say that B is polynomial time Turing reducible to A, if there is a polynomial time Turing reduction that, when evaluated on a string x relative to A returns value $B(x)$. We use the term *Turing* reduction to avoid confusion with the related notion of many–one reduction. Note, the choice of RAMs in the definition of polynomial time Turing reduction is not important; for example, the same class is obtained using any of a variety of machine models.

THEOREM 5. *A complexity type $\mathscr{C}$ contains sets A and B that are incomparable with regard to polynomial time reductions if and only if $\mathscr{C} \not\subseteq P$.*

Proof. Let $\mathscr{C}$ be fixed and let $(t_i)_{i \in \mathbb{N}}$ be characteristic for $\mathscr{C}$. We build A and B and use the $\mathscr{C}$-strategies to ensure that A and B belong to $\mathscr{C}$. In addition, we ensure for each polynomial time Turing reduction $\{e\}$,

$$\begin{aligned} &\{e\}(A) = B \Rightarrow \mathscr{C} \subseteq P \\ &\{e\}(B) = A \Rightarrow \mathscr{C} \subseteq P. \end{aligned} \tag{3.1}$$

We will describe the strategies for the new requirement, the *inequality strategies*. In fact, since they are symmetric, we only describe the strategy to ensure the first of the two implications in (3.1). These strategies are combined with the earlier ones using the same combinatorial pattern as before.

We describe the eth new strategy. We first arrange, by a variation of looking back, that no strategy of lower priority is implemented until we have established $\{e\}(A, x) \neq B(x)$ for some specific string x. Let l_0 be the greatest length of a string which is accepted into A or B by the effect of a strategy of higher priority. Let A_0 and B_0 be A and B restricted to strings of length less than or equal to l_0. Again, by looking back, we may assume that A_0, B_0, and l_0 are known. For each length l, we use a string x_l of length l to attempt to establish the inequality. (The choice of x_l is made to ensure that x_l will not even potentially be used by some strategy of higher priority.) We simulate the computation relative to the oracle that is equal to A_0 on strings of length less than l_0 and empty elsewhere. If we are able to complete the simulation without being canceled by a time control strategy of higher priority, then we define $B(x_l)$ to disagree with the answer returned by the simulation.

There are two possible outcomes for this strategy. We could succeed in establishing the inequality between $\{e\}(A)$ and B. In this case, the inequality strategy is compatible with the $\mathscr{C}$-strategies. It only requires that finitely may strings are in B and it places no permanent impediment to the implementation of all of the $\mathscr{C}$-strategies. These are the two features we already isolated as determining compatibility.

On the other hand, the inequality between $\{e\}(A)$ and B might never be established. This can only occur if for every l, the inequality strategy is terminated before completion of its simulation of $\{e\}(A, x_l)$ by some time control strategy of higher priority. But then the time control imposed by that higher priority strategy must be polynomial, since it is bounded by a constant times the running time of $\{e\}$. In this case, $\mathscr{C}$ is contained in P.

Thus, either the inequality strategies are compatible with the $\mathscr{C}$-strategies and we may use the framework developed earlier to build A and B in $\mathscr{C}$ of incomparable polynomial time Turing degrees or $\mathscr{C}$ is contained in P. ∎

The inequality strategies are, in some ways, simpler than the $\mathscr{C}$-strategies. In the style of Ladner's early constructions [La], they act to establish an inequality during the first opportunity to do so. If no opportunity arises, we conclude that $\mathscr{C}$ is contained in P. On the other hand, these strategies have a feature not appearing in the earlier argument. If an inequality strategy never finds a string at which it can establish the desired inequality, then it completely halts the implementation of lower priority strategies. This behavior is acceptable in the context of our construction, since if it occurs then we may conclude that the theorem is true for a fairly trivial reason. Of course, these types of strategies appear in basic looking back constructions. Here, we interleaved them with other finite injury strategies.

Minimal pairs. In Theorem 5, we gave a construction showing that in every complexity type there is a pair of sets which are polynomial time Turing incomparable. Thus, in the sense of Theorem 5, complexity type is never directly tied to relative computability. In this section, we ask whether there is any correlation between informational content as expressed by polynomial time Turing degree and complexity type. We give a partial negative answer in Theorem 6.

THEOREM 6. *There is a complexity type $\mathscr{C}$ which contains sets A and B that form a minimal pair with regard to polynomial time reductions* (*i.e.*, A, $B \notin P$, *but for all* X, $X \leqslant_p A$ *and* $X \leqslant_p B$ *implies that* $X \in P$).

Proof. We build A and B by recursion. A *condition* is a specification of an oracle on all strings whose lengths are bounded by a fixed integer; i.e., a finite approximation to an oracle. For U a condition, let *domain*(U) denote the set of strings on which U is defined. During stage n, we specify conditions A_n and B_n on A and B with domains at least including $\{0, 1\}^{\leqslant n}$ so that $A_{n-1} \subseteq A_n$ and $B_{n-1} \subseteq B_n$.

Readers familiar with recursion theoretic forcing and priority methods (see [Le]) will recognize our building a pair of Cohen generic subsets of $\{0, 1\}^*$. Their mutual genericity with respect to polynomial time Turing reductions implies that they form a minimal pair of P-degrees. We use the priority method to arrange that A and B meet enough dense sets for genericity to apply.

The fact that A and B are recursive follows from the observation that the recursion step in the generation of A_n and B_n is computable. In fact, there is a RAM that implements this recursion. When we speak of a step in the construction we are referring to a step in the execution of this RAM.

In the construction, we take steps to ensure that A and B satisfy the claims of the theorem by ruling out each possible counterexample. Thus, we individually satisfy the following individual requirements:

G_d. If $\{d\} = A$ then there is a RAM that runs at least as fast as $\{d\}$ and computes B. Similarly, for A and B with roles reversed.

$H_{e,f}$. If $\{e\}$ and $\{f\}$ are polynomial time oracle RAMs such that $\{e\}(A) = \{f\}(B)$, then their common value X is in P. In fact, in satisfying $H_{e,f}$ we will exhibit the polynomial time algorithm to compute X.

We ensure each of these requirements by use of an associated strategy. We will shortly sketch how the strategies operate. First, we give some indication of their context, since it is somewhat different than that in the earlier constructions. As before, we assign a priority ranking to the requirements. We invoke the first n strategies during stage n and thereby arrange that every strategy is in use during all but finitely many stages. We determine the action taken in the main recursion during stage n by means of a nested (minor) recursion of length n in which we calculate the effects of strategies. During stage n, we extend A and B so as to agree with the common value chosen by the maximum possible initial segment of strategies. Provided that for each strategy Σ,

(1) for all but finitely many stages, the conditions on A and B chosen by Σ are the same as those chosen by all higher priority strategies and

(2) for any sets A and B which are produced by a construction whose operation during all but finitely many stages is determined by Σ, A and B satisfy the requirement associated with Σ,

then A and B constructed as above will satisfy all of the requirements. We will first sketch the operation of the strategies and then the way be which they are combined in the minor recursion.

We turn now to our specific strategies. We view a strategy Σ as a *procedure*. It is called with arguments A_{n-1}, B_{n-1}, A_n^{default}, B_n^{default} and *default-time*. These arguments have the following types: the first four are conditions and the final one is an integer. Their intended roles in the construction are to have A_{n-1} and B_{n-1} as the conditions on A and B determined in the previous stage; the next two, A_n^{default} and B_n^{default}, are the default conditions on A and B which will be used at the end of stage n if Σ does not disallow their use; *default-time* is the number of steps needed to run the construction up to the point of calling Σ, which is enough to compute A_n^{default} and B_n^{default}. The strategy returns two conditions A_n and B_n which extend A_{n-1} and B_{n-1}. In the construction, they indicate the conditions that Σ intends be used as the stage n computation of A and B.

G_d. The strategy g_d to ensure the satisfaction of the requirement G_d acts as follows:

(1) First, check for a string y in domain(A_{n-1}) such that $\{d\}(y)$ converges in less than n steps and gives a value that is different from $A_{n-1}(y)$. If there is such a y, return A_n^{default} and B_n^{default}. *(In step* (1), *we look back to see whether G_d is already satisfied by an inequality between $\{d\}$ and A.)*

(2) Otherwise, check for a string y in domain(A_n^{default}) − domain(A_{n-1}) such that $\{d\}$ converges at y in less than *default-time* many steps. If there is such a y then for A_n, return the extension A^* of A_{n-1} that is identically equal to 0 at every string in domain(A_n^{default}) − domain(A_{n-1}) other than y. At y, A^* is defined to disagree with $\{d\}(y)$. For B_n, return the extension B^* of B_{n-1} that is identically equal to 0 on every string in domain (B_n^{default}) − domain(B_{n-1}). *(We look for an argument where it is faster to compute $\{d\}$ than it is to run the default computation. If we find one then we define A_n to establish $A \neq \{d\}$.)*

(3) If neither of these cases applies, then return A_n^{default} and B_n^{default}. *(If no action is required, return the default values.)*

If there is a y such that the evaluation of $\{d\}(y)$ takes less time than the evaluation of the default value for $B(y)$, then g_d ensures that $\{d\} \neq A$. Otherwise, g_d ensures that B is always given by the default calculation and so can be computed in less time than the evaluation of $\{d\}$. Thus, g_d ensures that if $\{d\} = A$ then there is an algorithm to compute B that runs in less time.

In addition, the values for A and B returned by g_d only deviate from the input default values finitely often. Once g_d's outputs are identical with the ultimate values of A_n and B_n, if g_d ever returns a value other than its input default value then, by (1), it automatically returns the default during every stage large enough to verify $\{d\} \neq A$.

$H_{e,f}$. Let $\{e\}$ and $\{f\}$ be polynomial time oracle RAMs. Let p be a polynomial that bounds their running times. The strategy $h_{e,f}$ to ensure the satisfaction of $H_{e,f}$ acts as follows:

(1) First, check for a string y such that $\{e\}(A_{n-1}, y) \neq \{f\}(B_{n-1}, y)$ is verified by a computation of length less than n. If there is such a y, return A_n^{default} and B_n^{default}. *(Look back to see whether the requirement is already satisfied by the inequality $\{e\}(A) \neq \{f\}(B)$.)*

(2) Otherwise, check for a y of length less than or equal to *default-time* such that there are two extensions A' and A'' of A_{n-1} such that $\{e\}(A', y)$ and $\{e\}(A'', y)$ are defined by computations with queries only to the domains of A' and A'' and have different values. If there is such a y, then return the value B^* for B_n that extends B_{n-1} and is identically equal to 0 on every string in $\{0, 1\}^{\leq p(\textit{default-time})}$ − domain(B_{n-1}). Return as value for A_n whichever of A' and A'' establishes $\{e\}(A_n, y) \neq \{f\}(B_n, y)$, for the (returned) value $B_n = B^*$. *(If the condition A_{n-1} does not already decide the value of $\{e\}(A, y)$ for all y's of length less than or equal to default-time then use the split in $\{e\}$ to make $\{e\}(A) \neq \{f\}(B)$. We attempt to meet a set of conditions associated with mutual genericity.)*

(3) If neither of these cases apply, then return A_n^{default} and B_n^{default}. (*As in* g_d, *if no action is required then return the default values.*)

Assume that $h_{e,f}$ is respected during all but finitely many stages. If the inequality between $\{e\}(A)$ and $\{f\}(B)$ is established in (2), then the requirement is trivially satisfied.

Assume that $\{e\}(A)$ and $\{f\}(B)$ are equal and let X be their common value. Let y be a string such that $h_{e,f}$ is respected during the stage when $A(y)$ is defined. We compute $X(y)$ as follows: First, compute the largest m so that the evaluation of A_{m-1}^{default} and B_{m-1}^{default} involves less than $|y|$ steps. Then, evaluate $\{e\}(A^*, y)$, where A^* is equal to A_{m-1}^{default} on domain$(A_{m-1}^{\text{default}})$ and is identically equal to 0 elsewhere.

Since its operations are explicitly evaluated in polynomial time, it is clear that this procedure can be implemented on a RAM in polynomial time. To see that it correctly computes X, note that since $h_{e,f}$ is respected during almost every stage and does not establish the inequality between $\{e\}(A)$ and $\{f\}(B)$, the default values for A and B are the ones actually used in the construction. Further, the m computed by y is the stage when $h_{e,f}$ examines all conditions extending A_{m-1} $(=A_{m-1}^{\text{default}})$ to find a pair of conditions that gives a pair of incompatible values for $\{e\}$ at y. By assumption, $h_{e,f}$ could not split the values of $\{e\}$, so every extension of A_{m-1} gives the same answer to $\{e\}(-, y)$ as A gives. In particular, $\{e\}(A^*, y) = \{e\}(A, y) = X(y)$, as desired.

Note, that this strategy also only returns conditions different from the input default values finitely often by the same argument that applied to g_d.

The Construction. This method of combining strategies also appears in [SS1]. We let A_0 and B_0 be the trivial conditions with empty domain. During stage n, we set up the minor recursion to invoke the first n strategies in the priority ordering. First, we execute the nth strategy with arguments A_{n-1}, B_{n-1}; A_n^{default} and B_n^{default}, given by the trivial extensions of A_{n-1} and B_{n-1} which are identically 0 on every string of length less than or equal to n not in the domains of A_{n-1} and B_{n-1}; and *default-time*, equal to the number of steps needed to compute these quantities. By recursion, in decreasing order of priority, we execute the next strategy with arguments A_{n-1}, B_{n-1}; A_n^{default} and B_n^{default}, obtained from the previous strategy's returned values; and *default-time*, equal to the number of steps needed to compute the construction through the point of executing the previous strategy. The output of the highest priority strategy (namely, the last strategy executed) gives the values for A_n and B_n.

Suppose that Σ is one of the above strategies. Then, Σ is in operation for all but finitely many stages. Hence, it determines the default value given to the strategies of higher priority for all but finitely many stages. Further, each strategy of higher priority than Σ only returns conditions different from the its input default values at most finitely often. So, the values returned by Σ will be the ones used by the construction during all but finitely many stages. By the above remarks, this is enough to conclude that the requirement associated with Σ is satisfied. ▌

Remark. It is an open problem whether *every* complexity type $\mathscr{C} \nsubseteq P$ contains a minimal pair.

4. On the Fine Structure of P

Linear time degrees. In contrast to many results in structural complexity theory that are only relevant for sets outside of P, the investigation of complexity types also leads to some challenging questions about the fine structure of P itself. One may argue that the exploration of the possibilities and limitations of construction techniques for sets in P may potentially be useful in order to distinguish P from larger complexity classes (e.g., PSPACE).

Those time bounds f that are commonly used in the analysis of algorithms for problems in P have the property that

$$\sup\{f(m) \mid m \leqslant c \cdot n\} = O(f(n))$$

for every constant $c \in N$, and that f agrees almost everywhere with some concave function g (i.e., $\forall k > n(g(n) \leqslant (n/k)\, g(k))$). These two properties together entail the useful fact that DTIME(f) is closed under linear time Turing reductions (we assume that the query tape is erased after each query). Note also that the first property alone guarantees already that DTIME(f) is closed under linear time many–one reductions.

In view of the preceding fact it is of interest to analyze for sets in P a slightly different notion of complexity type, where the underlying set T of time bounds is replaced by the class T_L of those f in T that satisfy the two additional properties above. This version has the advantage that each linear time Turing degree is contained in a single complexity type (in other words: each complexity type is closed under the equivalence relation $=_{\text{lin}}$). Therefore we assume in this subsection that T has been replaced by T_L.

Linear time reductions have provided the only successful means to show that certain concrete sets have exactly the same time complexity (e.g., Dewdney [D] proved that BIPARTITE MATCHING $=_{\text{lin}}$ VERTEX CONNECTIVITY ("are there $\geqslant k$ disjoint uv-paths in G, for u, v, k given")). The following result implies that this method is not general.

Theorem 7. *Every complexity type* $\mathscr{C} \nsubseteq$ DTIME(n) *of polynomial time computable sets contains infinitely many different linear time degrees, and the linear time degrees in* $\mathscr{C}$ *are dense. Furthermore,* $\mathscr{C}$ *contains incomparable linear time degrees, but no smallest linear time degree.*

Proof. Assume that $\mathscr{C} \nsubseteq$ DTIME(n). The construction of sets A, B in $\mathscr{C}$ that are incomparable with regard to linear time reductions proceeds as in the proof of Theorem 5.

In order to show that $\mathscr{C}$ contains no smallest linear time degree we fix some arbitrary set $A \in \mathscr{C}$. We construct a set $B \in \mathscr{C}$ with $A \not\leqslant_{\text{lin}} B$ by deleting from A all elements that lie in certain "intervals" $I_{n,m} := \{x \in \{0, 1\}^* \mid n \leqslant |x| \leqslant m\}$. Since $A \notin \text{DTIME}(n)$ one can falsify each possible linear time reduction from A to B by choosing the length $m - n$ of the removed interval $I_{n,m}$ sufficiently large (for given n define m via "looking back"). In order to guarantee that in addition $B \in \mathscr{C}$, we combine these strategies in a finite injury priority argument with the "$\mathscr{C}$-strategies" from the proof of Theorem 2 (see the discussion at the beginning of Section 3).

In order to show that the linear time degrees in $\mathscr{C}$ are dense we assume that sets A, $B \in \mathscr{C}$ are given with $B <_{\text{lin}} A$. Let $\tilde{B} := \{0^\frown x \mid x \in B\}$ and $\tilde{A} := \{1^\frown x \mid x \in A\}$. Then we have $A =_{\text{lin}} \tilde{A} \cup \tilde{B}$ (thus $\tilde{A} \cup \tilde{B} \in \mathscr{C}$ and $B <_{\text{lin}} \tilde{A} \cup \tilde{B}$). One constructs in the usual manner (with "looking back" as in [La]) a linear time computable set L such that for $D := (\tilde{A} \cap L) \cup \tilde{B}$ one has $D \not\leqslant_{\text{lin}} B$ and $\tilde{A} \cup \tilde{B} \not\leqslant_{\text{lin}} D$. Furthermore, it is obvious from the definition of D that $B \leqslant_{\text{lin}} D \leqslant_{\text{lin}} \tilde{A} \cup \tilde{B}$. Thus $D \in \mathscr{C}$ and $B <_{\text{lin}} D <_{\text{lin}} \tilde{A} \cup \tilde{B} =_{\text{lin}} A$.

Finally, we observe that the existence of sets A, B in $\mathscr{C}$ with $A \not\leqslant_{\text{lin}} B$ (see the beginning of the proof) implies that there is a set $G \in \mathscr{C}$ with $B <_{\text{lin}} G$: set

$$G := \{0^\frown x \mid x \in A\} \cup \{1^\frown x \mid x \in B\}.$$

Thus an iterative application of the preceding density result implies that $\mathscr{C}$ contains infinitely many linear time degrees. ▌

Complete sets. Often linear time functionals appear in proofs where one shows that some particularly natural set is complete in some complexity class. One shows that K is complete within a complexity class by showing that for any A in the class there is a linear time function Φ such that

$$(\forall \sigma)(\sigma \in A \Leftrightarrow \Phi(\sigma) \in K).$$

In other words, every A in the complexity class is *many-one* reducible to K by means of a linear time function. This condition is much stronger than the one asserting that K is complete with regard to Turing reductions.

DEFINITION. If $\mathscr{C}$ is a complexity type and $K \in \mathscr{C}$ then K *is complete in* $\mathscr{C}$ if for every A in $\mathscr{C}$ there is a linear time function Φ such that

$$(\forall \sigma)(\sigma \in A \Leftrightarrow \Phi(\sigma) \in K).$$

In [CR], Cook and Reckhow show that any principal complexity type has a complete set. In the following theorem, we analyze the non-principal case.

DEFINITION. A non-principal complexity type $\mathscr{C}$ is *dominated by linear composition* if there is a function F such that $\mathscr{C} \subseteq \text{DTIME}(F)$ and a constant c such that for every f in T, if $\mathscr{C} \subseteq \text{DTIME}(f)$ then

$$\lim_{n \to \infty} f(c \cdot n)/F(n) = \infty.$$

THEOREM 8. *A non-principal complexity type $\mathscr{C}$ has a complete set K if and only if $\mathscr{C}$ is dominated by linear composition.*

Proof. We begin by showing that the first condition implies the second. Suppose that K is in $\mathscr{C}$ but that $\mathscr{C}$ is not dominated by linear composition. We build a set A so that A is in $\mathscr{C}$ but for any linear time function Φ, there is a string σ such that $A(\sigma) \neq K(\Phi(\sigma))$.

We use the assumptions on $\mathscr{C}$ to make the following calculation. Suppose that F is in T, $\mathscr{C} \subseteq \mathrm{DTIME}(F)$ and that c is a constant. By the fact that $\mathscr{C}$ is not dominated by linear composition, there is a g in T such that $\mathscr{C} \subseteq \mathrm{DTIME}(g)$ and a constant m_g with

$$\liminf_{n \to \infty} g(c \cdot n)/F(n) \not\geq m_g.$$

Since $\mathscr{C}$ is not principal, there is an f such that $\mathscr{C} \subseteq \mathrm{DTIME}(f)$ and $\lim_{n \to \infty} f(n)/g(n) = 0$. For this f,

$$(\forall m)[\liminf_{n \to \infty} f(c \cdot n)/F(n) \not\geq m].$$

We will use the family of time control strategies to ensure that A is in $\mathscr{C}$. The additional inequality requirements state that A should not be reduced to K by any linear time function. We begin by describing the strategy to satisfy a single inequality requirement. With an eye to the future, we will design our strategy to work in the environment produced by finitely many time control strategies. Further, we will show that this strategy is compatible with the further implementation of strategies for all the remaining time control requirements.

Let F be given so that K is in $\mathrm{DTIME}(F)$. F will be viewed as the defining parameter in a time controlled environment, henceforth called the F-environment. Let M be a RAM which computes K in time $O(F)$. By working in the F-environment, we ensure that A can be computed by a RAM that runs in time $O(F)$. Let Φ be a linear time function on strings. We also ensure that Φ does not reduce A to K.

The F-environment is determined as follows: There is a simulation constant m_{sim} such that our strategy determining whether σ is an element of A will be terminated once we have executed $m_{\mathrm{sim}} \cdot F(|\sigma|)$ many steps.

Since Φ can be evaluated in linear time, let p be a constant such that for all σ, the evaluation of $\Phi(\sigma)$ takes only $p \cdot |\sigma|$ many steps. As calculated above, there is an f in T such that $\mathscr{C} \subseteq \mathrm{DTIME}(f)$ and for all m, $\liminf_{n \to \infty} f(p \cdot n)/F(n) \not\geq m$. Potentially, we use strategies for all RAMs M_i to attempt to satisfy the requirement. One of the strategies associated with a RAM that computes K with running time satisfying the above property of f will ensure that Φ does not reduce A to K.

The strategy associated with the RAM M_i is a straightforward diagonalization strategy. During stage s, we operate the strategy as follows: To begin with, by looking back, we ensure that this strategy has a purely finite effect if Φ does not reduce A to K. That is, we terminate the strategy once we observe a string σ^* such that $A(\sigma^*)$ is not equal to $K(\Phi(\sigma^*))$, where we compute K using M. If this inequality is not observed then we fix a string σ of length s. First, we simulate the computation of $\Phi(\sigma)$. Second, we simulate the execution of M_i on input $\Phi(\sigma)$. Finally, we define $A(\sigma)$ to be unequal to $M_i(\Phi(\sigma))$.

If we reach the conclusion of the above three steps, then we ensure that Φ does not reduce A to the set computed by M_i. Now consider the number of steps needed to execute this strategy on σ. Let f_i be the running time for M_i. The calculation of $\Phi(\sigma)$ involves $O(|\sigma|)$ many steps. Consequently, its simulation involves $O(|\sigma|)$ many steps as well. Let p be the constant associated with this simulation. Thus the first step of our strategy is over in $p \cdot |\sigma|$ many steps. The second step of the strategy is to simulate the execution of M_i on an input τ, equal to $\Phi(\sigma)$, of length less than or equal to $p \cdot |\sigma|$. Since M_i runs in time f_i, this simulation takes $O(f_i(p \cdot |\sigma|))$ many steps. Let p_1 be given so that the second step of our strategy involves at most $p_1 \cdot f_i(p \cdot |\sigma|)$ many steps.

Finally, the last step of the strategy is to specify the value of $A(\sigma)$. This involves only a constant number, p_2, of steps. Thus, the entire strategy can be executed in $p \cdot |\sigma| + p_1 \cdot f_i(p \cdot |\sigma|) + p_2$ many steps.

The inequality strategy will be terminated prematurely if and only if its execution exceeds $m_{\text{sim}} \cdot F(|\sigma|)$ many steps. We have already calculated that there is a RAM M_i that computes K such that for every m, $\liminf_{n \to \infty} f_i(c \cdot n)/F(n) \ngtr m$. Suppose that M_{i_0} is such an M_i. There are infinitely many n such that $p_1 \cdot f_i(c \cdot n)/F(n) < m_{\text{sim}}/2$. Since $\mathscr{C}$ is not linear $\lim_{n \to \infty} (p \cdot n + p_2)/F(n)$ is equal to 0. Thus there is an n such that

$$p \cdot n + p_1 \cdot f_i(p \cdot n) + p_2 < m_{\text{sim}} \cdot F(n).$$

If the requirement is not already seen to be satisfied by looking back during stage n, the M_{i_0}-strategy will not be terminated prematurely and will ensure that Φ fails to reduce A to K on its associated string of length n.

In organizing the global construction, we will have finitely many diagonal strategies active during any particular stage, all attempting to establish $A \neq K(\Phi)$. No two of these will operate with the same string σ. All of them are compatible with the time control strategy associated with F. Finally, since one of these strategies must ensure that A is not equal to $K(\Phi)$, the collective action of all of the strategies associated with the inequality requirement will be finite. These are the properties which we have isolated as implying compatibility with the global requirement that A be in the same complexity class as K.

We now consider the global construction of A. There is only one other ingredient to be described. In the above analysis, we worked relative to a particular RAM M which computed K and discussed the inequality strategies relative to the single time control strategy derived from F. In general, we will have to incorporate time control strategies derived from other RAMs M^* that potentially compute K in less time. These strategies operate exactly as the time control strategies of the earlier section. Each one imposes a time control on the collection of all lower priority strategies so that the actions of the lower priority strategies will be terminated in $O(f^*)$, where f^* is the running time of M^*.

We organize the global construction as follows. First, we fix a RAM M_K which computes K and a characteristic sequence $\mathbf{f}$ for $\mathscr{C}$. During any stage s we will have finitely many active time control strategies $C_1, ..., C_k$ associated with $f_1, ..., f_k$. Second, for some linear Φ, we will have finitely many strategies R_j associated with the inequality requirement associated with Φ that were active during the previous stage:

(1) By looking back, we check whether the inequality requirement for Φ_k has already been satisfied on some short string. If so then we set $A(\sigma) = 0$ for each σ of length s, say that Φ_k is satisfied, cancel all of the R_j associated with Φ_k and go to (2).

If not, we assign each strategy R_j a unique string of length s and execute the strategies R_j within the constraint imposed by the sequence of active C_i. If the strategy for σ is terminated by time control or σ is not associated with any strategy, we set $A(\sigma) = 0$.

As in the proof of Theorem 2, we activate strategies at a slow enough rate (logarithmic) so that the number of strategies is small enough to allow the time control strategies to function correctly. If possible within this constraint, we add the strategy R_{j_0} to our list of active strategies, where R_{j_0} is associated with the least linear time function that has not been satisfied and j_0 is the least index of a strategy associated with that function such that R_{j_0} had not been active earlier. We go to the next stage without executing (2).

(2) Under the same proviso as in (1), we introduce a strategy C_l for the least l such that C_l is not satisfied and the strategy C_l has not been active during any earlier stage. We go to the next stage.

Suppose that A is built according to the above blueprint. By the earlier analysis, the global effect of the family of strategies associated with a single linear time functional Φ will be finite. This ensures that every C_i will eventually be introduced. Hence, A will be no more complex than $\mathscr{C}$. Similarly, for each Φ we will reach a stage after which we sequentially introduce and execute the strategies associated with Φ until one of them establishes the inequality that ensures that A is not equal to $K(\Phi)$. Thus, $A + K$ is an element of $\mathscr{C}$ and is a counter example to the completeness of K. This completes the first half of the proof.

For the second half of the proof, we must show that if $\mathscr{C}$ is non-principal and dominated by linear composition, then there is a complete element in $\mathscr{C}$. This follows almost immediately from the Cook–Reckow theorem, that every principal complexity type has a complete element.

Since $\mathscr{C}$ is dominated by linear composition, let F be an element of T such that $\mathscr{C} \subseteq \mathrm{DTIME}(F)$ and let c be a constant such that for all f in T, if $\mathscr{C} \subseteq \mathrm{DTIME}(f)$ then

$$\lim_{n \to \infty} f(c \cdot n)/F(n) = \infty.$$

Let K be a complete element of DTIME(F). Let K_c be defined by $\sigma \in K_c$ if and only if there is a τ in K such that σ is the sequence $0^{c \cdot |\tau|} * 1 * \tau$. Here $*$ denotes concatenation and 0^k is the sequence of zeros of length k. By the above equation, K_c is in DTIME(f) for every f with $\mathscr{C} \subseteq$ DTIME(f). Clearly, K can be reduced to K_c in linear time. Consequently, since $\mathscr{C} \subseteq$ DTIME(F) every set in $\mathscr{C}$ can be reduced to K_c. If we take the joint of K with some element of $\mathscr{C}$, we obtain a set in $\mathscr{C}$ which is complete in $\mathscr{C}$. ▌

5. Open Problems

There are various results in structural complexity theory which state that there exist, in some complexity class K, sets with a certain property Q. Each such result gives rise to the more precise question, which complexity types in K contain sets with property Q. As examples we mention the open questions whether every complexity type contains a tally set (i.e., a subset of $\{0\}^*$) and whether every complexity type $\mathscr{C} \nsubseteq P$ contains a minimal pair of polynomial time Turing degrees. The answers to a number of open problems of this type appear to be of interest on their own. Furthermore, their solutions may help to enlarge our reservoir of construction techniques for computable sets (in particular, also for sets of „low" complexity).

Another problem area is the characterization of the structure of the degrees of computability of the sets in a complexity type $\mathscr{C}$. For example, one would like to know whether the sets in $\mathscr{C}$ realize infinitely many different types in the first-order language of the partial order of the degrees in $\mathscr{C}$, and whether this theory is decidable. Other open problems arise if one compares the degree structures of different complexity types. For example, we do not know whether the structure of polynomial time Turing degrees of sets in a complexity type $\mathscr{C} \nsubseteq P$ is the same for each such complexity type, and we do not know whether Theorem 8 specifies the only difference between the structure of linear time degrees within a non-zero complexity type $\mathscr{C} \subseteq P$.

The global structure of the complexity types ordered by $\leqslant_C$ should have some interesting features. In particular, it is not clear whether the complexity types of the familiar complexity classes occupy a distinguished position in this structure. By padding, one can produce a non-trivial homomorphism but this simple approach would seem to require that the padding function be a polynomial. It would be remarkable if the ideal of complexity types contained in P were preserved by all homomorphisms or even all automorphisms of the complexity types at large, ordered by $\leqslant_C$.

Finally, we would like to point out that all other resource-bounds for computations (e.g., nondeterministic time, or deterministic space) also give rise to the consideration of corresponding equivalence classes (or "complexity types") of those sets that are equivalent with regard to these complexity measures. Many questions that relate complexity types for deterministic computations with complexity types for nondeterministic computations or space bounded computations are obviously very hard. However, some of these may turn out to be easier to answer than the related "global" questions about inclusions among the corresponding complexity classes. As an example we would like to mention the problem whether there are sets A, B such that $A \neq_C B$ (i.e., A and B have different deterministic time complexity), but for all space constructible space bounds f we have $A \in \text{DSPACE}(f) \Leftrightarrow B \in \text{DSPACE}(f)$.

References

[AHU] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "The Design and Analysis of Computer Algorithms," Addison-Wesley, Reading, MA, 1974.

[B] M. Blum, A machine-independent theory of the complexity of recursive functions, *J. Assoc. Comput. Mach.* **14** (1967), 322–336.

[CR] S. A. Cook and R. A. Reckhow, Time-bounded random access machines, *J. Comput. System Sci.* 7 (1973), 354–375.

[D] A. K. Dewdney, Linear time transformations between combinatorial problems, *Internat. J. Comput. Math.* **11** (1982), 91–110.

[GHS] J. G. Geske, D. T. Huynh, and A. L. Selman, A hierarchy theorem for almost everywhere complex sets with applications to polynomial complexity degrees, *in* "Proceedings, 4th Annual Symposium on Theoretical Aspects of Computer Science," Lecture Note in Computer Science, Vol. 247, pp. 125–135, Springer-Verlag, New York/Berlin, 1987.

[HH] J. Hartmanis and J. E. Hopcroft, An overview of the theory of computational complexity, *J. Assoc. Comput. Mach.* **18** (1971), 444–475.

[La] R. Ladner, On the structure of polynomial-time reducibility, *J. Assoc. Comput. Mach.* **22** (1975), 155–171.

[LLR] L. Landweber, R. Lipton, and E. Robertson, On the structure of sets in NP and other classes, *Theoret. Comput. Sci.* **15** (1981), 181–200.

[Le] M. Lerman, "Degrees of Unsolvability," Springer-Verlag, New York/Berlin, 1983.

[L] L. A. Levin, On storage capacity for algorithms, *Soviet Math. Dokl.* **14** (1973), 1464–1466.

[Ly] N. Lynch, Helping: Several formalizations, *J. Symbolic Logic* **40** (1975), 555–566.

[MS1] W. Maass and T. A. Slaman, The complexity types of computable sets (extended abstract), *in* "Proc. of the 4th Annual Conf. on Structure in Complexity Theory 1989," pp. 231–239, I.E.E.E. Computer Society Press, Washington, 1989.

[MS2] W. Maass and T. A. Slaman, Extensional properties of sets of time bounded complexity (extended abstract), *in* "Proceedings, 7th Int. Conf. on Fundamentals of Computation Theory" (J. Csirik, J. Demetrovics, and F. Gésceg, Eds.), Lecture Notes in Computer Science, Vol. 380, pp. 318–326, Springer-Verlag, Berlin, 1989.

[MY] M. Machtey and P. Young, "An Introduction to the General Theory of Algorithms," North-Holland, Amsterdam, 1978.

[Ma] S. Mahaney, Sparse complete sets for NP: Solution of a conjecture of Berman and Hartmanis, *J. Comput. System Sci.* **25** (1982), 130–143.

[MF] A. R. Meyer and P. C. Fischer, Computational speed-up by effective operators, *J. Symbolic Logic* **37** (1972), 55–68.

[MW] A. R. Meyer and K. Winklmann, The fundamental theorem of complexity theory, *Math. Centre Tracts* **108** (1979), 97–112.

[P] W. J. Paul, "Komplexitaetstheorie," Teubner, Stuttgart, 1978.

[SS] C. P. Schnorr and G. Stumpf, A characterization of complexity sequences, *Z. Math. Logik Grundlag. Math.* **21** (1975), 47–56.

[SS1] J. Shinoda and T. A. Slaman, On the theory of the PTIME degrees of the recursive set, *in* "Proceedings, of Structure in Complexity Theory, 1988," pp. 252–257.