Nearly Linear Time

Yuri Gurevich[*]
Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109-2122

Saharon Shelah
Mathematics, Hebrew University, Jerusalem 91904, Israel
Mathematics, Rutgers University, New Brunswick, NJ 08903

Abstract

The notion of linear-time computability is very sensitive to machine model. In this connection, we introduce a class NLT of functions computable in *nearly linear* time $n(\log n)^{O(1)}$ on random access computers. NLT is very robust and does not depend on the particular choice of random access computers. Kolmogorov machines, Schönhage machines, random access Turing machines, etc., also compute exactly NLT functions in nearly linear time. It is not known whether usual multitape Turing machines are able to compute all NLT functions in nearly linear time. We do not believe they are and do not consider them necessarily appropriate for this relatively low complexity level. It turns out, however, that nondeterministic Turing machines accept exactly the languages in the nondeterministic version of NLT. We give also a machine-independent definition of NLT and a natural problem complete for NLT.

1 Introduction

What is Linear Time? In other words, what is the correct notion of linear time computability? The answer to this question is not clear at all. The notion of Linear Time seems to be badly dependent on computational model. It is possible that there is no universal notion of Linear Time and different versions of Linear Time are appropriate to different applications. On the other hand, Polynomial Time is very robust. Even Turing machines (walking painfully from A to B on their tapes to achieve what their luckier competitors can do in one step) are adequate for Polynomial Time. An analysis of arguments

[*]Partially supported by NSF grant DCR 85-03275 and by a grant from Binational US-Israel Science Foundation. A substantial portion of the work was done during a week in Fall 1985 when both authors visited Rutgers University; during the last stage of the work, the author was on a sabbatical visit to Stanford University and IBM Almaden Research Center.

in favor of the robustness of Polynomial Time turns up a much more modest but still very robust extension of the apparently ill-defined notion of Linear Time.

To prove that two machine models give the same notion of polynomial time computability, one often checks that any $T(n)$-time-bounded machine of one kind can be simulated by some machine of the other kind in time $T(n)h(n)$ where the overhead $h(n)$ is bounded by a polynomial of $T(n)$ or even n. (We restrict attention to computations with $T(n) \geq n$.) It is often the case that the overhead $h(n)$ is bounded by a polynomial of the logarithm of $T(n)$; let us call such simulations *efficient*. A noticeable exception is the simulation of random access machines by Turing machines.

Call a function $f(n)$ *nearly linear* if it is bounded by the product of n and some polynomial of $\log n$. (Functions bounded by polynomials of $\log n$ are often called *polylog* functions of n. Thus, a nearly linear function $f(n)$ is the product of n and a polylog function of n.) If two machine models efficiently simulate each other (i.e., every machine of one kind is efficiently simulated by a machine of the other kind) then they give the same notion of nearly linear time computability (as well as the same notion of nearly square time computability, etc.).

Turing machine models with 2, 3, etc. (linear) tapes form one cluster of machine models that efficiently simulate each other. Schnorr [12] introduced and studied the class QL (for Quasilinear Time) of functions computable on such Turing machines in nearly linear time and the class NQL of languages accepted by nondeterministic multitape Turing machines in nearly linear time. He showed in particular that SAT is complete for NQL with respect to QL reductions; see also [4] in this connection.

We identified an apparently different cluster of machine models that efficiently simulate each other. Choosing random access computers (RACs) of Angluin and Valiant [3] as our basic model in the cluster, we introduce a class NLT of functions computable on RACs in nearly linear time. (A language is NLT if its characteristic function is NLT.) Among other models in the cluster are random access Turing machines, Kolmogorov machines, storage modification machines of Schönhage, and Turing machines with the tape in the form of a tree. All models in the cluster give the same notion of nearly linear time computability. Thus NLT is very robust. In particular, the definition of NLT does not depend on whether RACs can multiply (or even add) in one step. Section 2 is devoted to the robustness of NLT. (The class NLT has been announced in [6].)

Whether NLT is the "robust closure" of Linear Time, we believe that it is a useful approximation to and an extension of Linear Time. QL, on the other hand, may not contain some functions computable in linear time on any — whatever modest — machine model $\mathcal{M}$ in the cluster of RACs. For, if (as one may expect) NLT properly contains QL, then padding inputs of any non-QL function in NLT gives a non-QL function computable in linear time on $\mathcal{M}$ machines.

Efficient simulations of Section 2 survive if deterministic models are replaced by corresponding nondeterministic models (and computing functions is replaced by accepting languages). The nondeterministic version NNLT of class NLT is even more robust than NLT: We show in Section 3 that NNLT = NQL. The coincidence of NNLT and NQL sheds some light on the difficulty of the problem whether NLT properly includes QL. It implies that NQL contains every NLT language. Moreover, if NLT contains some non-QL function, then NQL contains some non-QL language L; without loss of generality, some NTM accepts L in linear time. This is a much wider gap between deterministic and nondeterministic versions of Turing time that is presently known [10].

In order to stress the machine independent character of NLT, we provide a calculus of total NLT operations on binary strings in Section 4.

Finally, what reductions are appropriate for NLT decision problems? A natural choice is to use QL reductions. Problems that are QL hard for NLT are imposible to solve on Turing machines in nearly linear time unless QL=NLT. In Section 5, we exhibit a decision problem QL complete for NLT.

2 The Robustness of NLT

2.1 Random Access Computers

A random access computer [3] is an abstract machine with a sequence of memory locations. The size and the number of locations depend on the input size, so that a RAC can be seen as a sequence of finite machines. Each location contains a binary string of length $l = c \times \log n$ where n is the size of input and c depends on the given machine only. There are exactly 2^l locations. RAC is controlled by a program (that does not depend on n) consisting of a finite sequence of instructions. The time-complexity of a computation is the number of instructions executed. RACs do all the usual (for random access machines) operations: store, load, etc. The exact instruction set is immaterial for our purposes. The instruction set used in [3] is fine. Those instructions do not use registers. For some purposes, it may be convenient to use registers.

Call a RAC *frugal* if the visited locations always form an initial segment. In particular, a frugal RAC with a nearly linear time bound uses only nearly linear many locations.

Lemma 1 *Every RAC can be efficiently simulated by a frugal RAC.*

Proof Whenever the simulated RAC R uses a new location i, the simulating RAC R' uses the first unused location $L(i)$ where it stors both i and the content of i. To be able to find $L(i)$ promptly (whenever R needs location i), R' uses the balanced tree search and insertion algorithm [3, 7], namely, the locations $L(i)$ with weights i form a balanced tree. $\square$

In accordance with [9], call a RAC is a *write-once memory machine* (or simply *non-erasing*) if any bit of (any location in) the memory may be changed from 0 to 1 but never from 1 to 0.

Lemma 2 *Every frugal RAC R can be efficiently simulated by an appropriate frugal non-erasing RAC R'.*

Proof When R writes into a location i for the 2^m-th time, R' creates (on a new bloc of locations) a binary tree T_m with 2^m leaves equidistant from the root. If and when R writes into i for the $(2^m + k)$-th time, where $k < 2^m$, R' writes (the same content) into the k-th leaf of T_m; R' makes sure that all ancestors of the first k leaves, except for the root, are marked. Marking makes it easy to find the last leaf that has been written into (and thus facilitates easy reads) as well as the first leaf that has not been written into yet. When all leaves of T_m are written into, the address of the next relevant block of locations is written into the root of T_m. □

2.2 Generalized Turing Machines

It is not difficult to check that RACs can be efficiently simulated by random access Turing machines. For the sake of definiteness, let us formulate a specific version of random access Turing machines.

Definition 1 *An RTM is a Turing machine with three linear tapes, called the* main tape, *the* address tape *and the* auxiliary tape, *such that the head of the main tape (the* main head*) is always in the cell whose number is the contents of the address tape. An instruction for an RTM has the form*

$$(p, \alpha_0, \alpha_1, \alpha_3) \rightarrow (q, \beta_0, \beta_1, \beta_2, \gamma_1, \gamma_2)$$

and means the following: If the control state is p and the symbols in the observed cells on the three tapes are binary digits $\alpha_0, \alpha_1, \alpha_2$ respectively, then print binary digits $\beta_0, \beta_1, \beta_2$ in the respective cells, move the address head to the left (resp. to the right) if $\gamma_1 = -1$ (resp. $\gamma = +1$), move the auxiliary-tape head with respect to γ_2, and go to control state q. An RTM is frugal *if at any time t, the length of the address tape is $1 + \log(1 + t)$ and the length of the auxiliary tape is $O(1 + \log(1 + t)$. An RTM is* non-erasing *if, for every instruction, $\beta_0 \geq \alpha_0$.*

Lemma 3 *Every frugal RAC R can be efficiently simulated by a frugal RTM M. Moreover, if R is non-erasing then so is M.*

Proof Suppose that R uses l-bit locations. Then M uses the i-th block of l cells of the main tape to store the contents of the i-th location of R. The auxiliary tape is used to perform arithmetical operations, to make sure that the address head does not fall off the address tape, etc. □

Definition 2 *A BTM is a Turing machine with jumping and bisecting abilities. It has one tape and one head. In addition to usual Turing instructions, a program for a BTM can use the following instructions.*

Goto(a) *Move the head to the cell with the first, i.e., leftmost occurrence of a; if a does not occur on the tape then do nothing.*

Bisect(a) *If symbol a appears on the tape to the left of the head and the distance $j - i$ between the position j of the head and the position i of the first occurrence of a is even then move the head to the middle cell $(i + j)/2$; otherwise do nothing.*

Lemma 4 *Frugal RTMs can be efficiently simulated by BTMs.*

Proof Define BTMs with several heads and call them GTMs. The Bisect command has the following form for GTMs: Bisect(h_1, h_2). It sends the active head to cell number $(i + j)/2$ where i, j are the positions of h_1, h_2 respectively (provided that i, j have the same parity; otherwise the command has no effect). A GTM can be simulated by some BTM with a constant overhead. The desired BTM uses additional tape symbols to mark the locations of the heads of the given GTM.

Now let M be the given frugal RTM. One head (the *right guard*) of the desired GTM M' occupies the position of the rightmost used cell of M. Another head (the *left guard*) stays in the leftmost cell of the tape. M' mimics the address tape of M on a special track of its only tape. Using the two guards and some auxiliary heads, M' is able to simulate one step of M in $O(\log t)$ steps where t is the current position of the right guard. □

2.3 Storage Modification Machines

There are two brands of storage modification machines in the literature: Kolmogorov (or Kolmogorov-Uspensky) machines [8] and Schönhage machines [13]. A "philosophical" discussion on storage modification machines versus Turing machines can be found in [5].

Lemma 5 *BTMs can be efficiently simulated by Kolmogorov machines.*

Lemma 6 *Kolmogorov machines can be efficiently simulated by Schönhage machines.*

We omit the easy proofs of these lemmas.

Lemma 7 *Schöngage machines can be efficiently simulated by RACs.*

Proof In [13], Schönhage proves that his machines can be real-time simulated by very restricted RAMs (RAM1 model in his terminology). The same proof shows that Schönhage machines can be efficiently simulated by very restricted frugal RACs. □

2.4 Robustness Theorem

The lemmas of this section imply the following theorem.

Theorem 1 *RACs, frugal non-erasing RACs, RTMs, frugal nonerasing RTMs, BTMs, Kolmogorov machines and Schönhage machines all efficiently simulate each other and therefore compute exactly NLT functions in nearly linear time.*

3 Nondeterministic Nearly Linear Time

Recall that NNLT (resp. NQL) is the class of languages accepted by nondeterministic RACs (resp. nondeterministic multitape Turing machines) in nearly linear time.

Theorem 2 *NQL = NNLT.*

Proof It is obvious that NNLT includes NQL. The proof that every NNLT language L is NQL builds on the fact that multitape Turing machines can sort in nearly linear time [12].

Fix an RTM M that accepts L in nearly linear time $T(n)$. If C is a computation of M of some length t_0, define the trace of C as the sequence $\langle(t, q_t, a_t, I_t, b_t, J_t, c_t) : t \leq t_0\rangle$ where $q_t, a_t, I_t, b_t, J_t, c_t$ are respectively the state, the configuration of the address tape, the position of the head of the address tape, the configuration of the auxiliary tape, the position of the head of the auxiliary tape, and the observed character on the main tape in the moment t.

Given a string x of length n on a special input tape, the desired nondeterministic multitape Turing machine N guesses the trace $\langle(t, q_t, a_t, I_t, b_t, J_t, c_t) : t \leq T(n)\rangle$ of a presumably accepting computation C of M. It guesses the 6-tuples one by one and checks that the first tuple is correct, and every $t+1$-st tuple is consistent with the t-th one, and $q_T(n)$ is accepting. However, N does not check whether characters c_t are correct; this may require going too far into the history of the computation.

In order to check the correctness of characters c_{t+1}, N sorts the tuples first by the third and then by the first components. Then it reads the sorted list from the left to right. It uses a head on the input tape to check the correctnes of the first tuple in the block of tuples with the same content a of the address tape. Now suppose that (t, q, a, I, b, J, c), $(t', q', a, I', b', J', c')$ are in the same block and N knows already that c is correct. The first of the two tuples has enough information for N to decide whether and what M writes on the main tape at moment t. If M does not write at moment t then $c' = c$, else c' is exactly the character that M writes on the main tape at moment t. □

4 A Calculus for NLT

In this section, the term *operation* is used to denote total functions from binary strings to binary strings. The lower case letters u, v, w, x, y, z (with or without subscripts) denote binary strings.

Definition 3 Initial replacement *operations are as follows.*

$R0_{u,y}(x)$ *If u is an initial segment of x then replace it with y; else do nothing.*

$R1_{u,y}(x)$ *If u appears in x (as a substring) then replace the first, i.e., leftmost occurrence of u in x with y; else do nothing.*

$R2_{u,v,y,z}(x)$ *If u and v appear in x and their first occurrences do not overlap then replace the first occurrences of u and v with y and z respectively. It is assumed that all four parameter strings have the same length.*

$R3_{u,v,w,y,z}(x)$ *If x has a form $\ldots ux'vx''w\ldots$ where the shown occurrences of u and w are leftmost and $|x'| = |x''|$ then replace the shown occurrences of v and w with y and z respectively; else do nothing. It is assumed that all five parameter strings have the same length.*

Definition 4 *If f is an operation, then $f^0(x) = x$, $f^{i+1}(x) = f(f^i(x))$ and $f^*(x) = f^{|x|}(x)$. The operation f^* is called the* iteration *of f.*

Definition 5 *An* iterated replacement *is the iteration of the composition of initial replacement operation.*

Lemma 8 *Every iterated replacement is NLT.*

Proof Sketch Suppose we want to compute the iteration of a composition g of initial replacement operations. The initial replacement operation in g have parameter strings (u or u, v or u, v, w) that need to be found and ma;y be replaced; let U be the collection of all such parameter strings. The idea of the desired algorithm is to maintain a tree such that the current string x is the string of leaves and every node p of the tree keeps the list of U-strings that occur in the portion of x below p (trees tend to grow downward in computer science). It is easy to see that every replacement and the resulting update of the tree can be performed in polylog time. □

Definition 6 Initial extension and contraction *operations are as follows.*

$E_{u,v}(x)$ *Simultaneously replace every 0 with u and every 1 with v. Here u and v are different strings of the same length.*

$C_{u,v}(x)$ *If $x = E_{u,v}(y)$ for some y then y, else x.*

$A_u(x)$ *Add a tail of ln many copies of u to x. Here n is the length of x, and l is the length of the binary notation for n.*

$D_u(x)$ *Delete the (maximal) tail of u's in x.*

Let K be the closure of initial operations and iterated replacements under composition.

Theorem 3 *K is exactly the set of NLT operations.*

Proof It is easy to see that every initial operation is NLT and that NLT is closed under composition. It remain to apply Lemma 8 to show that every K operation is NLT.

To show that every NLT operation f is in K, fix a BTM (see Section 2) that computes f in nearly linear time and consider the computation of M on an input x of length n. Let A_1 be the set of states of M, A_2 be the set of tape symbols of M and $A = A_2 \cup (A_1 \times A_2)$. Assign different binary strings of a fixed length l to elements of A; if an element a is assigned a string $b_1 \dots b_l$, code a with a string $c(a) = 11b_1 0 b_2 0 \dots b_l 0$ of length $2+2l$. If w is a string $a_1 \dots a_k$ over A, code w with $c(w) = c(a_1) \dots c(a_k)$. (The artificial form of letter codes serves the following purpose: There are no "illegal" sneaking occurrences of letter codes in any $c(w)$.)

If p is the initial state of M, a codes the pair $(p,0)$ and b codes $(p,1)$, then

$$f_0 = R0_{c(1),b} \circ R0_{c(0),a} \circ E_{c(0),c(1)}$$

is the code for the the initial configuration of M where the blanks are ignored. Let B the code of the blank tape symbol of M and $m = m(n)$ be a nearly linear upper bound on both the run-time of M and the space used by M. There is a composition $(A_B)^k$ of several copies of A_B such that the length of the string

$$f_1(x) = (A_B)^k(f_0(x))$$

exceeds lm.

For expositional purposes, it is convenient to assume that M has a finite tape of length m. Then instantaneous descriptions (IDs) of M have the form $a_1 \dots a_m$ where exactly one of the symbols a_i belongs to $A_1 \times A_2$ and the others belong to A_2. The code $c(w)$ of an ID w will be called the *binary instantaneous description* (BID) of M. In particular, $f_1(x)$ is the initial BID of M.

For every instruction I of M there is a composition g_I of initial replacement operations such that for every BID y of M,

$$g_I(y) = [\text{If I is applicable at } y \text{ then the next BID, else } y\ ].$$

In the case of a Turing instruction, the desired g_I is a composition of R1 operations. In the case of Goto, g_I is the composition of R2 operations. And in the case of Bisect, g_I is a composition of R3 operations. Let g be the composition of all g_I. Then for every BID y of M, $g(y)$ is the next BID of M (if y is a halting BID then $g(y) = y$).

It is easy to see that $f_2(x) = g^*(f_1(x))$ is the final BID of M. Let f_3 be the composition of f_2 with a composition of R1 operation that "erases" the state, so that $f_3(x)$ is $c(f(x))$ with a tail of B's. Then

$$f = C_{c(0),c(1)} \circ D_B \circ f_3.\square$$

5 A Complete Problem

We start with a generalization of boolean formulas. The two boolean constants "true" and "false" are identified with 1 and 0 respectively. A *simple* (boolean) variable is an expression of the form p_u, where u is a binary string. A *complex* (boolean) variable is an expression of the form p_w where w is a string of boolean constants and simple boolean variable. (A simple variable is also a complex variable.) An *equation* is an expression of the form $p_w = F$ where p_w is a complex variable and F is a boolean combination of complex variables. A *generalized boolean formula* is a sequence of equations.

An *environment* is a function that evaluates (i.e. assigns boolean constants to) some simple variables. An environment e *identifies* a complex variable p_w if every simple variable in w belongs to the domain of e. For example, if $w = 01p_0p_1$ and $e(p_0) = 0$ and $e(p_1) = 1$ then e identifies p_w as the simple variable p_{0101}. An environment e *evaluates* a complex variable p_w if it identifies p_w and evaluates the resulting simple variable. Suppose that e identifies p_w as some simple variable q and suppose that e evaluates every variable in a boolean combination F and therefore evaluates F to some boolean value b; then the equation $p_w = F$ *alters* e to a new environment e' such that $\mathrm{dom}(e') = \mathrm{dom}(e) \cup \{q\}$, and $e'(p) = e(p)$ for every p in $\mathrm{dom}(e) - \{q\}$, and $e'(q) = b$ (the original environment e could be defined or undefined at q.)

A generalized boolean formula $E_0, \ldots, E_m$ is true if there are environments $e_0, \ldots, e_m$ such that

- e_0 is the empty environment,
- E_i alters e_i to e_{i+1}, for every $i < m$, and
- e_m evaluates E_m to "true".

Theorem 4 *The problem of evaluating generalized boolean formulas is complete for NLT under QL reductions.*

Proof It is clear that the problem is NLT. We will prove that the problem is hard for NLT. Given a language in NLT, fix a frugal non-erasing RTM M (see Section 2) that accepts the language. The desired generalized boolean formula describes the computation of M on some input of length n. Let t range over the steps in the computation and a over possible contents of the address tape. The difficulty is how to describe the main tape of M at all times; we cannot have a separate boolean variable for every pair (t, a). To get around a similar difficulty in his proof of NQL completeness of SAT, Schnorr used the fact that Turing machines can be efficiently simulated by oblivious Turing machines [11]. However no obliviousness result is known for random access Turing machine.

Our generalized boolean formula will use nearly linear many simple boolean variables which split into several groups. For example, q_t is a group of about $T \cdot \log m$ simple boolean variables where T is a bound on the length of the computation and m is the number of control states; the intended meaning of q_t is the control state at moment t. The intended meaning of other groups of simple variables is as follows:

a_t The contents of the address tape at moment t.

$a_{t,i}$ The i-th digit in a_t.

I_t The position of the head of address tape at moment t.

b_t The contents of the auxiliary tape at moment t.

$b_{t,j}$ The j-th digit of b_t

J_t The position of the head of auxiliary tape at moment t.

c_a The contents of the cell number a of the main tape. (Notice the lack of the time parameter.)

accept A special boolean variable to signal the acceptance.

For each t, the desired system of equations contains a subsystem S_t of polylog(n) many equations each of length polylog(n). If $t_1 < t_2$ then S_{t_1} equations precede S_{t_2} equations. S_0 equations reflect the initial conditions of M; they

- set q_0 to the initial state of M,
- initialize all $a_{0,i}$ and $b_{0,j}$ to zero,
- set variables c_a with respect to the initial configuration.

The S_{t+1} equations reflect the updates performed at time t. They set each of

$$q_{t+1}, a_{t+1,I_t}, I_{t+1}, b_{t+1,J_t}, J_{t+1}$$

equal to an expression of the form

$$f(q_t, a_{t,I_t}, b_{t,J_t}, c_{a_t})$$

for appropriate boolean functions f. And c_{a_t} is set to:

$$c_{a_t} \vee g(q_t, a_{t,I_t}, b_{t,J_t})$$

for an appropriate function g.

The final equation in our system (in addition to all equations in all S_t) sets the variable *accept* to 1 if the final state of M is accepting. $\Box$ ¡ **Remark 1** During the presentation of this paper in IBM Almaden Center on Bay Area Theory Day, November 11, 1988, Moshe Vardi noted the special character of the generalized boolean formulas, constructed in the proof of Theorem 4: if an environment e is altered to an environment e' and $e'(q) \neq e(q)$ then $e'(q) = 1$. In other words, environments are revised only upward; this gives a flavor of least fixed points to our generalized boolean formulas. Moshe asked about a natural NLT problem explicitly in terms of least fixed points. The thought is attractive; the details need to be worked out.

Remark 2 The reductions of Theorem 4 are not only QL but also logspace. There is however a problem with logspace QL reductions: there is no reason to believe that they are closed under composition.

References

[1] G. M. Adelson-Velsky. Soviet Math. 3 (1962), 1259-1263.

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

[3] Dana Angluin and Les Valiant. *Fast Probabilistic Algorithm for Hamiltonian Circuits and Matchings.* J. of Computer and System Sciences 18 (1979), 155–193.

[4] Stephen A. Cook. *Short Propositional Formulas Represent Nondeterministic Computations.* IPL 26 (1987/88), 269–270.

[5] Yuri Gurevich. *Kolmogorov Machines and Related Issues: The Column on Logic in Computer Science.* Bulletin of European Assoc. for Theor. Comp. Science 35, June 1988, 71–82.

[6] Yuri Gurevich and Saharon Shelah. *Functions Computable in Nearly Linear Time.* AMS Abstracts 7:4 (1986), p. 236.

[7] Donald E. Knuth. *The Art of Computer Programming: Volume 3 / Sorting and Searching.* Addison-Wesley, Reading, Mass. 1973.

[8] A. N. Kolmogorov and V. A. Uspensky. *On the Definition of an Algorithm.* Uspekhi Mat. Nauk 13:4 (1958), 3–28 (Russian) or AMS Translations, ser. 2, vol. 21 (1963), 217–245.

[9] Sandy Irani, Moni Naor, Ronitt Rubinfeld. *On the Time and Space Complexity of Computation Using Write-Once Memory.* Manuscript, Computer Science Division, UC Berkeley, Nov. 1988.

[10] W. J. Paul, N. Pippenger, E. Szemeredi and W. T. Trotter, *On determinism versus non-determinism and related problems.* Proc. 24th IEEE Symposium on Foundation of Computer Science, November 1983, Tucson, Arizona, 429–438.

[11] N. Pippinger and M. J. Fischer. *Relations among Complexity Measures.* J. ACM 26:2 (1979), 361–381.

[12] Claus P. Schnorr. *Satisfiability is Quasilinear Complete in NQL.* Journal of ACM 25:1 (1978), 136–145.

[13] A. Schönhage. *Storage Modification Machines.* SIAM J. Computing 9:3, August 1980, 490–508.