Algebraic Computation Models

February 20, 2012

Motivation

Why use algebraic models of computation?

Motivation

Why use algebraic models of computation?

- Bit-based models of computation (like TM's) do not capture the essence of computations involving real or complex numbers (or other fields).

Motivation

Why use algebraic models of computation?

- Bit-based models of computation (like TM's) do not capture the essence of computations involving real or complex numbers (or other fields).
- Analyzing the complexity of algebraic computations using a bit-based model can be misleading (there is another side to the coin as we shall see below).

Motivation

Why use algebraic models of computation?

- Bit-based models of computation (like TM's) do not capture the essence of computations involving real or complex numbers (or other fields).
- Analyzing the complexity of algebraic computations using a bit-based model can be misleading (there is another side to the coin as we shall see below).
- Many areas of Computer Science use algebraic computations: computational algebra and geometry, numerical analysis, signal processing, robotics, et.c.

Motivation

Why use algebraic models of computation?

- Bit-based models of computation (like TM's) do not capture the essence of computations involving real or complex numbers (or other fields).
- Analyzing the complexity of algebraic computations using a bit-based model can be misleading (there is another side to the coin as we shall see below).
- Many areas of Computer Science use algebraic computations: computational algebra and geometry, numerical analysis, signal processing, robotics, et.c.
- Useful approximation to the asymptotic behavior of algebraic algorithms, as computers are allowed to use bigger precision day by day (progress in hardware).

Potential Pitfalls

Designers of algebraic models must be careful not to produce a *too powerful* model. We do not need something unrealistic.

Potential Pitfalls

Designers of algebraic models must be careful not to produce a *too powerful* model. We do not need something unrealistic.

Example. Shamir has shown that we can factor any integer N in $\texttt{poly}(\log N)$ time in any model that allows arithmetic (including `mod`) with arbitrary precision.

Potential Pitfalls

Designers of algebraic models must be careful not to produce a *too powerful* model. We do not need something unrealistic.

Example. Shamir has shown that we can factor any integer N in $\texttt{poly}(\log N)$ time in any model that allows arithmetic (including `mod`) with arbitrary precision. ■

Example. Avoid a model which allows hardwired real numbers in its programs, as a single real number can encode infinite amount of information (like an answer to *every* instance of `SAT`).

Potential Pitfalls

Designers of algebraic models must be careful not to produce a *too powerful* model. We do not need something unrealistic.

Example. Shamir has shown that we can factor any integer N in $\texttt{poly}(\log N)$ time in any model that allows arithmetic (including $\texttt{mod}$) with arbitrary precision. ■

Example. Avoid a model which allows hardwired real numbers in its programs, as a single real number can encode infinite amount of information (like an answer to *every* instance of $\texttt{SAT}$). ■

We can avoid such pitfalls by restricting the algorithm's ability to access individual bits.

Algebraic Straight-Line Programs

Something like the following:

Program. Computes $e \times (x_1 + e) + \pi \times x_2$ in $\mathbb{R}$.

Input: x_1, x_2
Output: y_4

$$y_1 = x_1 + e$$
$$y_2 = \pi \times x_2$$
$$y_3 = e \times y_1$$
$$y_4 = y_3 + y_2$$

In the above straight-line program, x_1 and x_2 are given as inputs and the output y_4 is computed from previous y_i's, which are the results of a binary operation in the field; π and e are built-in constants.

- Note: straight-line = no conditionals or loops.

Algebraic Straight-Line Programs (cnt'd)

Definition. An *algebraic straight-line program* of length T with *input variables* $x_1, x_2, \ldots, x_n \in \mathbb{F}$ and *built-in constants* $c_1, c_2, \ldots, c_n \in \mathbb{F}$ is a sequence of T statements of the form

$$y_i = z_{i_1} * z_{i_2},$$

where $*$ is an operation in $\mathbb{F}$, and each of z_{i_1}, z_{i_2} is either an input variable, a built-in constant or y_j for $j < i$.

Algebraic Straight-Line Programs (cnt'd)

Definition. An *algebraic straight-line program* of length T with *input variables* $x_1, x_2, \ldots, x_n \in \mathbb{F}$ and *built-in constants* $c_1, c_2, \ldots, c_n \in \mathbb{F}$ is a sequence of T statements of the form

$$y_i = z_{i_1} * z_{i_2},$$

where $*$ is an operation in $\mathbb{F}$, and each of z_{i_1}, z_{i_2} is either an input variable, a built-in constant or y_j for $j < i$.

The computation consists of executing these simple statements in order, finding values for $y_1, y_2, \ldots, y_T$. The *output* of the computation is the value of y_T.

Algebraic Straight-Line Programs (cnt'd)

Definition. An *algebraic straight-line program* of length T with *input variables* $x_1, x_2, \ldots, x_n \in \mathbb{F}$ and *built-in constants* $c_1, c_2, \ldots, c_n \in \mathbb{F}$ is a sequence of T statements of the form

$$y_i = z_{i_1} * z_{i_2},$$

where $*$ is an operation in $\mathbb{F}$, and each of z_{i_1}, z_{i_2} is either an input variable, a built-in constant or y_j for $j < i$.

The computation consists of executing these simple statements in order, finding values for $y_1, y_2, \ldots, y_T$. The *output* of the computation is the value of y_T.

Lemma. The output of a straight-line program of length T with variables $x_1, x_2, \ldots, x_n$ is a polynomial $p(x_1, x_2, \ldots, x_n)$ of degree at most 2^T.

Algebraic Straight-Line Programs (cnt'd)

When asking *complexity* questions for a problem computable in this model, we are interested in the length (as a function of n) of the program for an input $x_1, x_2, \ldots, x_n$.

[1] z is a primitive mth root of unity $\iff z^m = 1$ and $z^k \neq 1, \forall k < m$

Algebraic Straight-Line Programs (cnt'd)

When asking *complexity* questions for a problem computable in this model, we are interested in the length (as a function of n) of the program for an input $x_1, x_2, \ldots, x_n$.

Some problems computable in this model and their complexity:

- *Polynomial Multiplication*:
  - $O(n^2)$ using the school method,

[1] z is a primitive mth root of unity $\iff z^m = 1$ and $z^k \neq 1, \forall k < m$

Algebraic Straight-Line Programs (cnt'd)

When asking *complexity* questions for a problem computable in this model, we are interested in the length (as a function of n) of the program for an input $x_1, x_2, \ldots, x_n$.

Some problems computable in this model and their complexity:

- *Polynomial Multiplication*:
  - $O(n^2)$ using the school method,
  - $O(n \log n)$ using FFT, for fields that have a primitive mth root of unity[1], where m is the smallest power of 2 less than $2n$,

[1] z is a primitive mth root of unity $\iff z^m = 1$ and $z^k \neq 1, \forall k < m$

Algebraic Straight-Line Programs (cnt'd)

When asking *complexity* questions for a problem computable in this model, we are interested in the length (as a function of n) of the program for an input $x_1, x_2, \ldots, x_n$.

Some problems computable in this model and their complexity:

- *Polynomial Multiplication*:
  - $O(n^2)$ using the school method,
  - $O(n \log n)$ using FFT, for fields that have a primitive mth root of unity[1], where m is the smallest power of 2 less than $2n$,
  - $O(n \log n \log \log n)$ - using FFT, for any field.

[1] z is a primitive mth root of unity $\iff z^m = 1$ and $z^k \neq 1, \forall k < m$

Algebraic Straight-Line Programs (cnt'd)

When asking *complexity* questions for a problem computable in this model, we are interested in the length (as a function of n) of the program for an input $x_1, x_2, \ldots, x_n$.

Some problems computable in this model and their complexity:

- *Polynomial Multiplication*:
  - $O(n^2)$ using the school method,
  - $O(n \log n)$ using FFT, for fields that have a primitive mth root of unity[1], where m is the smallest power of 2 less than $2n$,
  - $O(n \log n \log \log n)$ - using FFT, for any field.
- FFT: $O(n \log n)$ [Cooley and Tukey].

---

[1] z is a primitive mth root of unity $\iff z^m = 1$ and $z^k \neq 1, \forall k < m$

Algebraic Straight-Line Programs (cnt'd)

When asking *complexity* questions for a problem computable in this model, we are interested in the length (as a function of n) of the program for an input $x_1, x_2, \ldots, x_n$.

Some problems computable in this model and their complexity:

- *Polynomial Multiplication*:
  - $O(n^2)$ using the school method,
  - $O(n \log n)$ using FFT, for fields that have a primitive mth root of unity[1], where m is the smallest power of 2 less than $2n$,
  - $O(n \log n \log \log n)$ - using FFT, for any field.
- FFT: $O(n \log n)$ [Cooley and Tukey].
- Matrix Multiplication: $O(n^3)$ with the naive method and this can be improved using techniques like Strassen's.

---

[1] z is a primitive mth root of unity $\iff z^m = 1$ and $z^k \neq 1, \forall k < m$

Algebraic Straight-Line Programs (cnt'd)

When asking *complexity* questions for a problem computable in this model, we are interested in the length (as a function of n) of the program for an input $x_1, x_2, \ldots, x_n$.

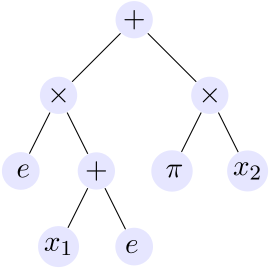
Some problems computable in this model and their complexity:

- *Polynomial Multiplication*:
  - $O(n^2)$ using the school method,
  - $O(n \log n)$ using FFT, for fields that have a primitive mth root of unity[1], where m is the smallest power of 2 less than $2n$,
  - $O(n \log n \log \log n)$ - using FFT, for any field.
- FFT: $O(n \log n)$ [Cooley and Tukey].
- Matrix Multiplication: $O(n^3)$ with the naive method and this can be improved using techniques like Strassen's.
- Determinant: $O(n^3)$ using Gaussian elimination.

[1] z is a primitive mth root of unity $\iff z^m = 1$ and $z^k \neq 1, \forall k < m$

Algebraic Circuits

Again, an example (x_1, x_2 inputs, e, π constants):



The above circuit computes $e \times (x_1 + e) + \pi \times x_2$ in $\mathbb{R}$. Note the similarity with the straight-line program that we saw earlier.

Algebraic Circuits (cnt'd)

Definition. An *algebraic circuit* consists of an acyclic graph. The leaves are called *input nodes*, are labeled $x_1, x_2, \ldots, x_n$ and take values in a field $\mathbb{F}$. We also allow special input nodes labeled with arbitrary constants $c_1, c_2, \ldots, c_k \in \mathbb{F}$. Each internal node, called a *gate*, is labeled with one of the operations $+, \times$. We consider only circuits with a single output node and with the in-degree of each gate being 2.

Algebraic Circuits (cnt'd)

Definition. An *algebraic circuit* consists of an acyclic graph. The leaves are called *input nodes*, are labeled $x_1, x_2, \ldots, x_n$ and take values in a field $\mathbb{F}$. We also allow special input nodes labeled with arbitrary constants $c_1, c_2, \ldots, c_k \in \mathbb{F}$. Each internal node, called a *gate*, is labeled with one of the operations $+, \times$. We consider only circuits with a single output node and with the in-degree of each gate being 2.

The *size* of a circuit is the number of gates in it. The *depth* of the circuit is the length of the longest path from input to output in it.

Algebraic Circuits (cnt'd)

Definition. An *algebraic circuit* consists of an acyclic graph. The leaves are called *input nodes*, are labeled $x_1, x_2, \ldots, x_n$ and take values in a field $\mathbb{F}$. We also allow special input nodes labeled with arbitrary constants $c_1, c_2, \ldots, c_k \in \mathbb{F}$. Each internal node, called a *gate*, is labeled with one of the operations $+, \times$. We consider only circuits with a single output node and with the in-degree of each gate being 2.

The *size* of a circuit is the number of gates in it. The *depth* of the circuit is the length of the longest path from input to output in it.

Lemma. Let $f : \mathbb{F}^n \to \mathbb{F}$ be some function. If f has an algebraic straight-line program of size S, then it has an algebraic circuit of size $3S$. If it is computable by an algebraic circuit of size S then it is computable by an algebraic straight line program of length S.

Algebraic Circuits (cnt'd)

Definition. Let $\mathbb{F}$ be a field. We say that a family of polynomials $\{p_n\}_{n \in \mathbb{N}}$, where p_n takes n variables over $\mathbb{F}$, has *polynomially-bounded degree* if there is a constant c s.t. for every n the degree of p_n is at most cn^c.

Algebraic Circuits (cnt'd)

Definition. Let $\mathbb{F}$ be a field. We say that a family of polynomials $\{p_n\}_{n\in\mathbb{N}}$, where p_n takes n variables over $\mathbb{F}$, has *polynomially-bounded degree* if there is a constant c s.t. for every n the degree of p_n is at most cn^c.

Definition. The class $\mathbf{AlgP_{/poly}}$ contains all polynomially bounded degree families of polynomials that are computable by algebraic circuits of polynomial size and polynomial degree.

Algebraic Circuits (cnt'd)

Definition. Let $\mathbb{F}$ be a field. We say that a family of polynomials $\{p_n\}_{n\in\mathbb{N}}$, where p_n takes n variables over $\mathbb{F}$, has *polynomially-bounded degree* if there is a constant c s.t. for every n the degree of p_n is at most cn^c.

Definition. The class $\mathbf{AlgP_{/poly}}$ contains all polynomially bounded degree families of polynomials that are computable by algebraic circuits of polynomial size and polynomial degree.

Definition. The class $\mathbf{AlgNP_{/poly}}$ is the class of polynomially bounded degree families $\{p_n\}$ that are definable as

$$p_n(x_1, x_2, \ldots, x_n) = \sum_{e\in\{0,1\}^{m-n}} g_m(x_1, x_2, \ldots, x_n, e_{n+1}, \ldots, e_m),$$

where $g_m \in \mathbf{AlgP_{/poly}}$ and m is polynomial in n.

Algebraic Circuits (cnt'd)

Definition. A function $f(x_1, x_2, \ldots, x_n)$ is a *projection* of a function $g(y_1, y_2, \ldots, y_n)$ if there is a mapping σ from $\{y_1, y_2, \ldots, y_n\}$ to $\{0, 1, x_1, x_2, \ldots, x_n\}$ s.t.

$$f(x_1, x_2, \ldots, x_n) = g(\sigma(y_1), \sigma(y_2), \ldots, \sigma(y_n)).$$

We say f is *projection-reducible* to g if f is a projection of g.

Algebraic Circuits (cnt'd)

Definition. A function $f(x_1, x_2, \ldots, x_n)$ is a *projection* of a function $g(y_1, y_2, \ldots, y_n)$ if there is a mapping σ from $\{y_1, y_2, \ldots, y_n\}$ to $\{0, 1, x_1, x_2, \ldots, x_n\}$ s.t.

$$f(x_1, x_2, \ldots, x_n) = g(\sigma(y_1), \sigma(y_2), \ldots, \sigma(y_n)).$$

We say f is *projection-reducible* to g if f is a projection of g.

Example. Let $f(x_1, x_2) = x_1 + x_2$; f is projection-reducible to $g(y_1, y_2, y_3) = y_1^2 y_3 + y_2$ since $f(x_1, x_2) = g(1, x_1, x_2)$. ■

Algebraic Circuits (cnt'd)

Definition. A function $f(x_1, x_2, \ldots, x_n)$ is a *projection* of a function $g(y_1, y_2, \ldots, y_n)$ if there is a mapping σ from $\{y_1, y_2, \ldots, y_n\}$ to $\{0, 1, x_1, x_2, \ldots, x_n\}$ s.t.

$$f(x_1, x_2, \ldots, x_n) = g(\sigma(y_1), \sigma(y_2), \ldots, \sigma(y_n)).$$

We say f is *projection-reducible* to g if f is a projection of g.

Example. Let $f(x_1, x_2) = x_1 + x_2$; f is projection-reducible to $g(y_1, y_2, y_3) = y_1^2 y_3 + y_2$ since $f(x_1, x_2) = g(1, x_1, x_2)$. ■

Completeness results based on the above:

- `determinant` is $\mathbf{AlgP_{/poly}}$**-complete**.
- `permanent` is $\mathbf{AlgNP_{/poly}}$**-complete**.

Algebraic Circuits (cnt'd)

Definition. A function $f(x_1, x_2, \ldots, x_n)$ is a *projection* of a function $g(y_1, y_2, \ldots, y_n)$ if there is a mapping σ from $\{y_1, y_2, \ldots, y_n\}$ to $\{0, 1, x_1, x_2, \ldots, x_n\}$ s.t.

$$f(x_1, x_2, \ldots, x_n) = g(\sigma(y_1), \sigma(y_2), \ldots, \sigma(y_n)).$$

We say f is *projection-reducible* to g if f is a projection of g.

Example. Let $f(x_1, x_2) = x_1 + x_2$; f is projection-reducible to $g(y_1, y_2, y_3) = y_1^2 y_3 + y_2$ since $f(x_1, x_2) = g(1, x_1, x_2)$. ∎

Completeness results based on the above:

- `determinant` is $\mathbf{AlgP_{/poly}}$-**complete**.
- `permanent` is $\mathbf{AlgNP_{/poly}}$-**complete**.

Interesting fact: we need to show $\mathbf{AlgP_{/poly}} \neq \mathbf{AlgNP_{/poly}}$ before we can show $\mathbf{P} \neq \mathbf{NP}$.

Blum-Shub-Smale

- The first uniform algebraic model that we see.
- Generalization of TM's.

Blum-Shub-Smale

- The first uniform algebraic model that we see.
- Generalization of TM's.
    - Input string in $\mathbb{F}^n$.

Blum-Shub-Smale

- The first uniform algebraic model that we see.
- Generalization of TM's.
  - Input string in $\mathbb{F}^n$.
  - Each cell can hold an element of $\mathbb{F}$.

Blum-Shub-Smale

- The first uniform algebraic model that we see.
- Generalization of TM's.
  - Input string in $\mathbb{F}^n$.
  - Each cell can hold an element of $\mathbb{F}$.
- Three categories of states:
  - *Shift*: move head ± 1 cell.

Blum-Shub-Smale

- The first uniform algebraic model that we see.
- Generalization of TM's.
  - Input string in $\mathbb{F}^n$.
  - Each cell can hold an element of $\mathbb{F}$.
- Three categories of states:
  - *Shift*: move head ± 1 cell.
  - *Branch*: `If` current cell $= a$, `then` goto q_1 `else` goto q_2.

Blum-Shub-Smale

- The first uniform algebraic model that we see.
- Generalization of TM's.
  - Input string in $\mathbb{F}^n$.
  - Each cell can hold an element of $\mathbb{F}$.
- Three categories of states:
  - *Shift*: move head ± 1 cell.
  - *Branch*: `If` current cell $= a$, `then` goto q_1 `else` goto q_2.
  - *Compute*: replace the content a of the current cell with $f(a)$, where f is a hard-wired function (polynomial or rational depending on whether $\mathbb{F}$ is a ring or a field).

Blum-Shub-Smale

- The first uniform algebraic model that we see.
- Generalization of TM's.
  - Input string in $\mathbb{F}^n$.
  - Each cell can hold an element of $\mathbb{F}$.
- Three categories of states:
  - *Shift*: move head ± 1 cell.
  - *Branch*: `If` current cell $= a$, `then` goto q_1 `else` goto q_2.
  - *Compute*: replace the content a of the current cell with $f(a)$, where f is a hard-wired function (polynomial or rational depending on whether $\mathbb{F}$ is a ring or a field).
- Note: need a register for branching.

Blum-Shub-Smale

- The first uniform algebraic model that we see.
- Generalization of TM's.
  - Input string in $\mathbb{F}^n$.
  - Each cell can hold an element of $\mathbb{F}$.
- Three categories of states:
  - *Shift*: move head ± 1 cell.
  - *Branch*: `If` current cell $= a$, `then` goto q_1 `else` goto q_2.
  - *Compute*: replace the content a of the current cell with $f(a)$, where f is a hard-wired function (polynomial or rational depending on whether $\mathbb{F}$ is a ring or a field).
- Note: need a register for branching.
- Very powerful model (e.g. computes x^{2^n} in n steps).

Blum-Shub-Smale

- The first uniform algebraic model that we see.
- Generalization of TM's.
  - Input string in $\mathbb{F}^n$.
  - Each cell can hold an element of $\mathbb{F}$.
- Three categories of states:
  - *Shift*: move head ± 1 cell.
  - *Branch*: `If` current cell $= a$, `then` goto q_1 `else` goto q_2.
  - *Compute*: replace the content a of the current cell with $f(a)$, where f is a hard-wired function (polynomial or rational depending on whether $\mathbb{F}$ is a ring or a field).
- Note: need a register for branching.
- Very powerful model (e.g. computes x^{2^n} in n steps).
- Could be even more powerful.

Blum-Shub-Smale

- The first uniform algebraic model that we see.
- Generalization of TM's.
  - Input string in $\mathbb{F}^n$.
  - Each cell can hold an element of $\mathbb{F}$.
- Three categories of states:
  - *Shift*: move head ± 1 cell.
  - *Branch*: `If` current cell $= a$, `then` goto q_1 `else` goto q_2.
  - *Compute*: replace the content a of the current cell with $f(a)$, where f is a hard-wired function (polynomial or rational depending on whether $\mathbb{F}$ is a ring or a field).
- Note: need a register for branching.
- Very powerful model (e.g. computes x^{2^n} in n steps).
- Could be even more powerful.
  - Allowing $\geq$ comparisons when branching would give it the ability to decide every language in $\mathbf{P_{/poly}}$. (even undecidable!)

Blum-Shub-Smale

- The first uniform algebraic model that we see.
- Generalization of TM's.
  - Input string in $\mathbb{F}^n$.
  - Each cell can hold an element of $\mathbb{F}$.
- Three categories of states:
  - *Shift*: move head ± 1 cell.
  - *Branch*: `If` current cell $= a$, `then` goto q_1 `else` goto q_2.
  - *Compute*: replace the content a of the current cell with $f(a)$, where f is a hard-wired function (polynomial or rational depending on whether $\mathbb{F}$ is a ring or a field).
- Note: need a register for branching.
- Very powerful model (e.g. computes x^{2^n} in n steps).
- Could be even more powerful.
  - Allowing $\geq$ comparisons when branching would give it the ability to decide every language in $\mathbf{P_{/poly}}$. (even undecidable!)
  - Allowing rounding as a basic operation would give it the ability of integer factorization in poly-time.

Blum-Shub-Smale (cnt'd)

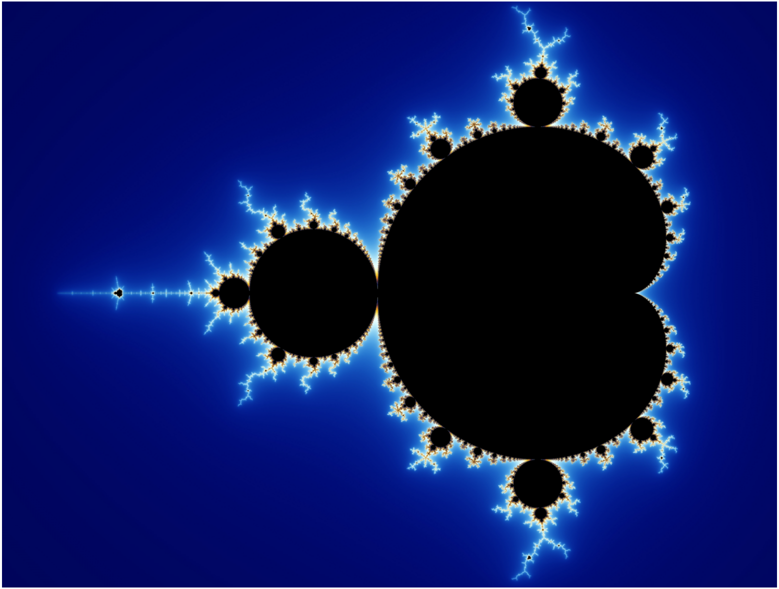
This power raises decidability questions. Can we compute anything with it?

Blum-Shub-Smale (cnt'd)

This power raises decidability questions. Can we compute anything with it?—NO

Definition. For complex c, z define $p_c(z) = z^2 + c$. The *Mandelbrot set* is defined as

$$\mathcal{M} = \{c \in \mathbb{C} \mid \text{the sequence } p_c(0), p_c(p_c(0)), \ldots \text{ is bounded } \}.$$



[http://upload.wikimedia.org/wikipedia/commons/2/21/Mandel_zoom_00_mandelbrot_set.jpg]

Blum-Shub-Smale (cnt'd)

Theorem. $\mathcal{M}$ is undecidable by a macine over $\mathbb{C}$.

Blum-Shub-Smale (cnt'd)

Theorem. $\mathcal{M}$ is undecidable by a macine over $\mathbb{C}$.

Philosophical questions: Roger Penrose vs. Artificial Intelligence.

Bibliography

- S. Arora and B. Barak, Computational complexity: a modern approach, Cambridge University Press, 2009.
- A. Shamir, Factoring numbers in O(log n) arithmetic steps, Inf. Process. Lett., 1979.
- L. G. Valiant, Completeness classes in algebra. In STOC, ACM, 1979.
- L. G. Valiant, The complexity of computing the permanent, Theoretical Computer Science, 1979.
- L. Blum, M. Shub, and S. Smale, On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines, American Mathematical Society, 1989.
- R. Penrose, The Emperor's New Mind, Oxford University Press, 1989.