Algorithms and implementations

Yiannis N. Moschovakis
UCLA and University of Athens

Tarski Lecture 1, March 3, 2008

What is an algorithm?

- Basic aim: to "define" (or represent) algorithms in set theory, in the same way that we represent *real numbers* (Cantor, Dedekind) and *random variables* (Kolmogorov) by set-theoretic objects
- What set-theoretic objects represent algorithms?
- When do two two set-theoretic objects represent the same algorithm? (The algorithm identity problem)
- In what way are algorithms effective?
- . . . and do it so that the basic results about algorithms can be established rigorously (and naturally)
- . . . and there should be some applications!

Plan for the lectures

Lecture 1. *Algorithms and implementations*
Discuss the problem and some ideas for solving it

Lecture 2. *English as a programming language*
Applications to Philosophy of language (and linguistics?)
synonymy and faithful translation $\sim$ algorithm identity

Lecture 3. *The axiomatic derivation of absolute lower bounds*
Applications to complexity (joint work with Lou van den Dries)
Do not depend on pinning down algorithm identity

Lectures 2 and 3 are independent of each other and mostly independent of Lecture 1

I will oversimplify, but: All lies are white (John Steel)

Outline of Lecture 1

Slogan: *The theory of algorithms is the theory of recursive equations*

(1) Three examples
(2) Machines vs. recursive definitions
(3) Recursors
(4) Elementary algorithms
(5) Implementations

Notation:

$$\mathbb{N} = \{0, 1, 2, \ldots\}$$

$$a \geq b \geq 1, \quad a = bq + r, \quad 0 \leq r < b$$
$$\Longrightarrow \quad q = \mathrm{iq}(a, b), \quad r = \mathrm{rem}(a, b)$$

$\gcd(a, b) =$ the greatest common divisor of a and b

$a \perp\!\!\!\perp b \iff \mathrm{rem}(a, b) = 1$ (a and b are coprime)

The Euclidean algorithm ε

For $a, b \in \mathbb{N}, a \geq b \geq 1$,

$$\varepsilon : \boxed{\gcd(a, b) = \text{if } (\text{rem}(a, b) = 0) \text{ then } b \text{ else } \gcd(b, \text{rem}(a, b))}$$

$c_\varepsilon(a, b) =$ the number of divisions needed to compute $\gcd(a, b)$ using ε

Complexity of the Euclidean

If $a \geq b \geq 2$, *then* $c_\varepsilon(a, b) \leq 2 \log_2(a)$

Proofs of the correctness and the upper bound are by induction on $\max(a, b)$

What is the Euclidean algorithm?

$$\varepsilon : \boxed{\gcd(a,b) = \text{if } (\text{rem}(a,b) = 0) \text{ then } b \text{ else } \gcd(b, \text{rem}(a,b))}$$

- It is an algorithm on $\mathbb{N}$, from (relative to) the remainder function rem and it computes $\gcd : \mathbb{N}^2 \to \mathbb{N}$
- It is needed to make precise the optimality of the Euclidean:

Basic Conjecture

For every algorithm α which computes on $\mathbb{N}$ from rem *the greatest common divisor function, there is a constant $r > 0$ such that for infinitely many pairs $a \geq b \geq 1$,*

$$c_\alpha(a,b) \geq r \log_2(a)$$

Sorting (alphabetizing)

Given an ordering $\leq$ on a set A and any $u = \langle u_0, \ldots, u_{n-1} \rangle \in A^n$

$\text{sort}(u) =$ the unique, sorted (non-decreasing) rearrangement

$$v = \langle u_{\pi(0)}, u_{\pi(1)}, \ldots, u_{\pi(n-1)} \rangle$$

where $\pi : \{0, \ldots, n-1\} \rightarrowtail\!\!\!\!\rightarrow \{0, \ldots, n-1\}$ is a permutation

$$\begin{aligned}
\text{head}(\langle u_0, \ldots, u_{n-1} \rangle) &= u_0 \\
\text{tail}(\langle u_0, \ldots, u_{n-1} \rangle) &= \langle u_1, \ldots, u_{n-1} \rangle \\
\langle x \rangle * \langle u_0, \ldots, u_{n-1} \rangle &= \langle x, u_0, \ldots, u_{n-1} \rangle \quad \text{(prepend)} \\
|\langle u_0, \ldots, u_{n-1} \rangle| &= n \quad \text{(the length of } u\text{)} \\
h_1(u) &= \text{the first half of } u \quad \text{(the first half)} \\
h_2(u) &= \text{the second half of } u \quad \text{(the second half)}
\end{aligned}$$

The mergesort algorithm σ_m

$$\mathrm{sort}(u) = \text{if } (|u| \leq 1) \text{ then } u \text{ else } \mathrm{merge}(\mathrm{sort}(h_1(u)), \mathrm{sort}(h_2(u)))$$

$$\mathrm{merge}(v, w) = \begin{cases} w & \text{if } |v| = 0, \\ v & \text{else, if } |w| = 0, \\ \langle v_0 \rangle * \mathrm{merge}(\mathrm{tail}(v), w) & \text{else, if } v_0 \leq w_0, \\ \langle w_0 \rangle * \mathrm{merge}(v, \mathrm{tail}(w)) & \text{otherwise.} \end{cases}$$

(1) If v, w are sorted, then $\mathrm{merge}(v, w) = \mathrm{sort}(w * v)$

(2) The sorting and merging function satisfy these equations

(3) $\mathrm{merge}(v, w)$ can be computed using no more than $|v| + |w| \dot{-} 1$ comparisons

(4) $\mathrm{sort}(u)$ can be computed by σ_m using no more than $|u| \log_2(|u|)$ comparisons $(|u| > 1)$

What is the mergesort algorithm?

$$\mathrm{sort}(u) = \text{if } (|u| \leq 1) \text{ then } u \text{ else } \mathrm{merge}(\mathrm{sort}(h_1(u)), \mathrm{sort}(h_2(u)))$$

$$\mathrm{merge}(v, w) = \begin{cases} w & \text{if } |v| = 0, \\ v & \text{else, if } |w| = 0, \\ \langle v_0 \rangle * \mathrm{merge}(\mathrm{tail}(v), w) & \text{else, if } v_0 \leq w_0, \\ \langle w_0 \rangle * \mathrm{merge}(v, \mathrm{tail}(w)) & \text{otherwise.} \end{cases}$$

$$c_{\sigma_m}(u) = \text{the number of comparisons needed to compute } \mathrm{sort}(u)$$
$$\text{using } \sigma_m \leq |u| \log_2(|u|) \quad (|u| > 0)$$

- It is an algorithm from the ordering $\leq$ and the functions $\mathrm{head}(u), \mathrm{tail}(u), |u|, \ldots$
- It is needed to make precise the optimality of σ_m:
  For every sorting algorithm σ from $\leq, \mathrm{head}, \mathrm{tail}, \ldots$, there is an $r > 0$ and infinitely many sequences u such that $c_\sigma(u) \geq r|u| \log_2(|u|)$ (well known)

The Gentzen Cut Elimination algorithm

Every proof d of the Gentzen system for Predicate Logic can be transformed into a cut-free proof $\gamma(d)$ with the same conclusion

$$\begin{aligned}\gamma(d) = \text{if } & T_1(d) \text{ then } f_1(d)\\ & \text{else if } T_2(d) \text{ then } f_2(\gamma(\tau(d)))\\ & \text{else } f_3(\gamma(\sigma_1(d)), \gamma(\sigma_2(d)))\end{aligned}$$

- It is a recursive algorithm from natural syntactic primitives, very similar in logical structure to the mergesort
- **Main Fact**: $|\gamma(d)| \leq e(\rho(d), |d|)$, where $|d|$ is the length of the proof d, $\rho(d)$ is its cut-rank, and

$$e(0, k) = k, \quad e(n+1, k) = 2^{e(n,k)}$$

The infinitary Gentzen algorithm

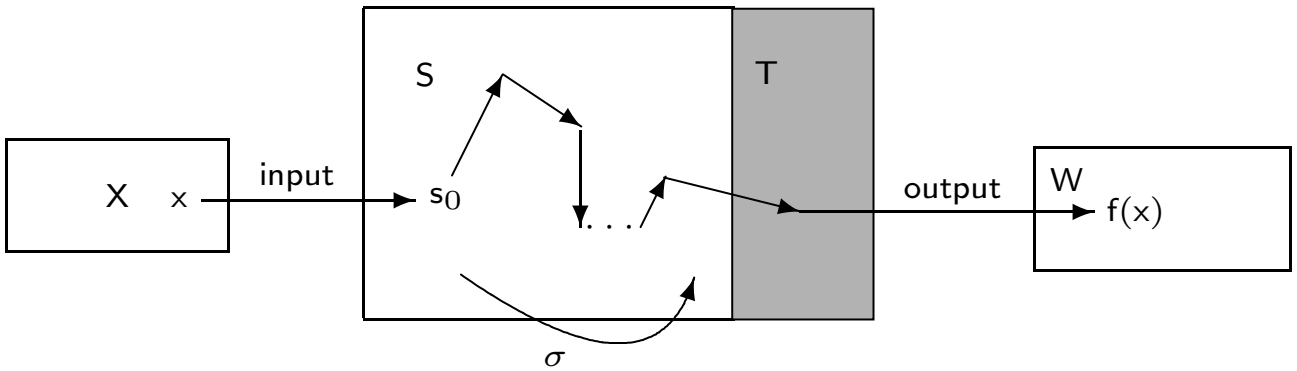
If we add the ω-rule to the Gentzen system for Peano arithmetic, then cuts can again be eliminated by an extension of the finitary Gentzen algorithm

$$\begin{aligned}\gamma^*(d) = \text{ if } & T_1(d) \text{ then } f_1(d)\\ & \text{else if } T_2(d) \text{ then } f_2(\gamma^*(\tau(d)))\\ & \text{else if } T_3(d) \text{ then } f_3(\gamma^*(\sigma_1(d)), \gamma^*(\sigma_2(d)))\\ & \text{else } f_4(\lambda(n)\gamma^*(\rho(n,d))),\end{aligned}$$

where f_4 is a functional embodying the ω-rule

- Again $|\gamma^*(d)| \leq e(\rho(d), |d|)$, where cut-ranks and lengths of infinite proofs are ordinals, $e(\alpha, \beta)$ is defined by ordinal recursion, and so every provable sentence has a cut-free proof of length less than
  $\varepsilon_0 =$ the least ordinal > 0 and closed under $\alpha \mapsto \omega^\alpha$

Abstract machines (computation models)



A machine $\mathfrak{m} : X \rightsquigarrow Y$ is a tuple $(S, \text{input}, \sigma, T, \text{output})$ such that

1. S is a non-empty set (of states)
2. $\text{input} : X \to S$ is the input function
3. $\sigma : S \to S$ is the transition function
4. T is the set of terminal states, $T \subseteq S$
5. $\text{output} : T \to Y$ is the output function

$$\overline{\mathfrak{m}}(x) = \text{output}(\sigma^n(\text{input}(x)))$$

where $n =$ least such that $\sigma^n(\text{input}(x)) \in T$

Infinitary algorithms are not machines

- It is useful to think of the infinitary Gentzen "effective procedure" as an algorithm
- There are applications of infinitary algorithms (in Lecture 2)
- Machines are special algorithms which implement finitary algorithms
- The relation between an (implementable) algorithm and its implementations is interesting

Which machine is the Euclidean?

$$\varepsilon : \boxed{\gcd(a,b) = \text{if } (\text{rem}(a,b) = 0) \text{ then } b \text{ else } \gcd(b, \text{rem}(a,b))}$$

- Must specify a set of states, an input function, a transition function, etc.
- This can be done, in many ways, generally called implementations of the Euclidean
- The choice of a "natural" (abstract) implementation is irrelevant for the correctness and the log upper bound of the Euclidean, which are derived directly from the recursive equation above and apply to all implementations

- Claim: ε is completely specified by the equation above

Which machine is the mergesort algorithm?

$$\mathrm{sort}(u) = \text{if } (|u| \leq 1) \text{ then } u \text{ else } \mathrm{merge}(\mathrm{sort}(h_1(u)), \mathrm{sort}(h_2(u)))$$

$$\mathrm{merge}(v, w) = \begin{cases} w & \text{if } |v| = 0, \\ v & \text{else, if } |w| = 0, \\ \langle v_0 \rangle * \mathrm{merge}(\mathrm{tail}(v), w) & \text{else, if } v_0 \leq w_0, \\ \langle w_0 \rangle * \mathrm{merge}(v, \mathrm{tail}(w)) & \text{otherwise.} \end{cases}$$

- Many (essentially) different implementations
  sequential (with specified orders of evaluation), parallel, ...
- The correctness and $n \log_2(n)$ upper bound are derived directly from a (specific reading) of these recursive equations
- They should apply to all implementations of the mergesort
- Claim: σ_m is completely specified by the system above
- *Task: Define σ_m, define implementations, prove* •

Slogans and questions

- Algorithms compute functions from specific primitives
- They are specified by systems of recursive equations
- An algorithm is (faithfully modeled by) the semantic content of a system of recursive equations
- Machines are algorithms, but not all algorithms are machines
- Some algorithms have machine implementations
- An algorithm codes all its implementation-independent properties
- What is the relation between an algorithm and its implementations?
  ... or between two implementations of the same algorithm?

Main slogan

The theory of algorithms is the theory of recursive equations

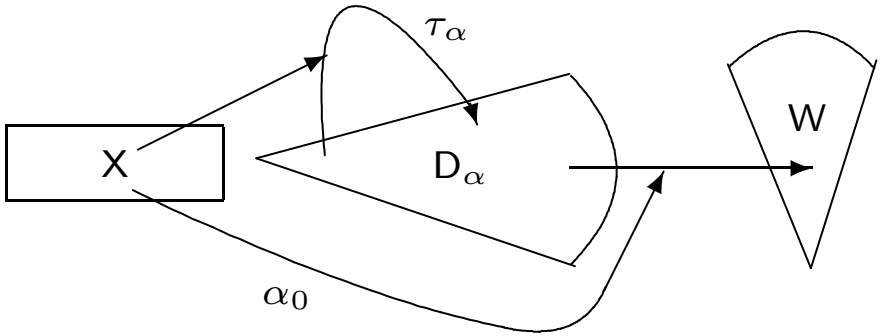
(Skip non-deterministic algorithms and fairness)

Monotone recursive equations

- A complete poset is a partial ordered set $D = (\text{Field}(D), \leq_D)$ in which every directed set has a least upper bound
- Standard example:
  $(X \rightharpoonup Y)$ = the set of all partial functions $f : X \rightharpoonup Y$
- A function $f : D \to E$ is monotone if $x \leq_D y \implies f(x) \leq_E f(y)$
  ($f : X \rightharpoonup Y$ is a monotone function on X to $Y \cup \{\perp\}$)
- *For every monotone $f : D \to D$ on a complete D, the equation* $\boxed{x = f(x)}$ *has a least solution*

- Complete posets (domains) are the basic objects studied in Scott's Denotational Semantics for programming languages
- Much of this work can be viewed as a refinement of Denotational Semantics (which interprets programs by algorithms)

Recursors



A recursor $\alpha : X \rightsquigarrow W$ is a tuple $\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_k)$ such that

1. X is a poset, W is a complete poset
2. $D_1, \ldots, D_k$ are complete posets, $D_\alpha = D_1 \times \cdots \times D_k$, the solution space of α
3. $\alpha_i : X \times D_\alpha \to D_i$ is monotone $(i = 1, \ldots, k)$
4. $\tau_\alpha(x, \vec{d}) = (\alpha_1(x, \vec{d}), \ldots, \alpha_k(x, \vec{d}))$ is the transition function, $\tau_\alpha : X \times D_\alpha \to D_\alpha$
5. $\alpha_0 : X \times D_1 \times \cdots \times D_k \to W$ is monotone, the output map

$\overline{\alpha}(x) = \alpha_0(x, \overline{d}_1, \ldots, \overline{d}_k)$ for the least solution of $\boxed{\vec{d} = \tau_\alpha(x, \vec{d})}$

We write $\alpha(x) = \alpha_0(x, \vec{d})$ where $\{\vec{d} = \tau_\alpha(x, \vec{d})\}$

Recursor isomorphism

Two recursors

$$\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_k), \quad \alpha' = (\alpha'_0, \alpha'_1, \ldots, \alpha'_m) : X \rightsquigarrow W$$

are isomorphic ($\alpha \simeq \alpha'$) if

(1) $k = m$ (same number of parts)

(2) There is a permutation $\pi : \{1, \ldots, k\}$ and poset isomorphisms $\rho_i : D_i \to D'_{\pi(i)}$ $(i = 1, \ldots, k)$ such that ...
(the order of the equations in the system $\vec{d} = \tau_\alpha(x, \vec{d})$ does not matter)

Isomorphic recursors $\alpha, \alpha' : X \rightsquigarrow W$ compute the same function $\overline{\alpha} = \overline{\alpha}' : X \to W$

Machines or recursors?

With each machine $\mathfrak{m} = (S, \text{input}, \sigma, T, \text{output}) : X \rightsquigarrow Y$ we associate the tail recursor

$$\mathfrak{r}_{\mathfrak{m}}(x) = p(\text{input}(x)) \text{ where}$$
$$\{p = \lambda(s)[\text{if } (s \in T) \text{ then output}(s) \text{ else } p(\sigma(s))]\}$$

- $\mathfrak{m}$ and $\mathfrak{r}_{\mathfrak{m}}$ compute the same partial function $\bar{\mathfrak{r}}_{\mathfrak{m}} = \overline{\mathfrak{m}} : X \rightharpoonup Y$
- **Theorem** (with V. Paschalis) The map $\mathfrak{m} \mapsto \mathfrak{r}_{\mathfrak{m}}$ respects isomorphisms, $\mathfrak{m} \simeq \mathfrak{m}' \iff \mathfrak{r}_{\mathfrak{m}} \simeq \mathfrak{r}_{\mathfrak{m}'}$
- The question is one of choice of terminology (because the mergesort system is also needed)
- Yuri Gurevich has argued that algorithms are machines (and of a very specific kind)
- Jean-Yves Girard has also given similar arguments

Elementary (first order) algorithms

Algorithms which compute partial functions from given partial functions

(Partial, pointed) algebra $\mathbf{M} = (M, 0, 1, \Phi^{\mathbf{M}})$

where $0, 1 \in M$, Φ is a set of function symbols (the vocabulary) and $\Phi^{\mathbf{M}} = \{\phi^{\mathbf{M}}\}_{\phi \in \Phi}$, with $\phi^{\mathbf{M}} : M^{n_\phi} \rightharpoonup M$ for each $\phi \in \Phi$

$\mathbf{N}_\varepsilon = (\mathbb{N}, 0, 1, \mathrm{rem})$, the Euclidean algebra
$\mathbf{N}_u = (\mathbb{N}, 0, 1, S, \mathrm{Pd})$, the *unary numbers*
$\mathbf{N}_b = (\mathbb{N}, 0, 1, \mathrm{Parity}, \mathrm{iq}_2, (x \mapsto 2x), (x \mapsto 2x + 1))$, the *binary numbers*
$\mathbf{A}^* = (A^*, 0, 1, \leq, \mathrm{head}, \mathrm{tail}, \ldots)$, the mergesort algebra, with $0, 1 \in A^*$

Standard model-theoretic notions must be mildly adapted, for example for (partial) subalgebras:

$$\mathbf{U} \subseteq_p \mathbf{M} \iff \{0, 1\} \subseteq U \subseteq M \text{ and for all } \phi, \phi^{\mathbf{U}} \subseteq \phi^{\mathbf{M}}$$

Recursive (McCarthy) programs of $\mathbf{M} = (M, 0, 1, \Phi^{\mathbf{M}})$

Explicit Φ-terms (with partial function variables and conditionals)

$$A :\equiv 0 \mid 1 \mid \mathrm{v}_i \mid \phi(A_1, \ldots, A_n) \mid \mathrm{p}_i^n(A_1, \ldots, A_n) \mid \text{if } (A = 0) \text{ then } B \text{ else } C$$

Recursive program (only $\vec{x}_i, p_1, \ldots, p_K$ occur in each part A_i):

$$A \quad : \quad \begin{cases} \mathrm{p}_A(\vec{x}_0) = A_0 \\ \mathrm{p}_1(\vec{x}_1) = A_1 \\ \qquad \vdots \\ \mathrm{p}_K(\vec{x}_K) = A_K \end{cases} \qquad (A_0 : \text{ the head}, (A_1, \ldots, A_K) : \text{ the body})$$

- What is the semantic content of the system A?

The recursor of a program in $\mathbf{M}$

$$A \quad : \quad \begin{cases} \mathrm{p}_A(\vec{x}_0) = A_0 \\ \mathrm{p}_1(\vec{x}_1) = A_1 \\ \vdots \\ \mathrm{p}_K(\vec{x}_K) = A_K \end{cases}$$

$$\mathfrak{r}(A, \mathbf{M})(\vec{x}) = \mathrm{den}(A_0, \mathbf{M})(\vec{x}, \vec{p}) \text{ where}$$

$$\left\{ p_1 = \lambda(\vec{x}_1)\mathrm{den}(A_1, \mathbf{M})(\vec{x}_1, \vec{p}), \ldots, p_K = \lambda(\vec{x}_K)\mathrm{den}(A_K, \mathbf{M})(\vec{x}_K, \vec{p}) \right\}$$

$\mathfrak{r}(A, \mathbf{M})$ is not exactly the algorithm expressed by A in $\mathbf{M}$.

For example, if $A \quad : \quad p_A(\vec{x}) = A_0(\vec{x})$ has empty body, then

$$\mathfrak{r}(A, \mathbf{M})(\vec{x}) = \mathrm{den}(A_0, \mathbf{M})(\vec{x}) \text{ where } \{ \quad \}$$

is just the function defined on $\mathbf{M}$ by A_0
(which may involve much explicit computation)

The problem of defining implementations

van Emde Boas:

> *Intuitively, a simulation of [one class of computation models] M by [another] M' is some construction which shows that everything a machine $M_i \in M$ can do on inputs x can be performed by some machine $M'_i \in M'$ on the same inputs as well;*

We will define a reducibility relation $\alpha \leq_r \beta$ and call a machine $\mathfrak{m}$ an implementation of α if $\alpha \leq_r \mathfrak{r}_\mathfrak{m}$

(where $\mathfrak{r}_\mathfrak{m}$ is the recursor representation of the machine $\mathfrak{m}$)

Recursor reducibility

Suppose $\alpha, \beta : X \rightsquigarrow W$, (e.g., $\beta = \mathfrak{r}_{\mathfrak{m}}$ where $\mathfrak{m} : X \rightsquigarrow W$):
A *reduction* of α to β is any monotone mapping

$$\pi : X \times D_\alpha \to D_\beta$$

such that the following three conditions hold, for every $x \in X$ and every $d \in D_\alpha$:

(R1) $\tau_\beta(x, \pi(x, d)) \leq \pi(x, \tau_\alpha(x, d))$.
(R2) $\beta_0(x, \pi(x, d)) \leq \alpha_0(x, d)$.
(R3) $\overline{\alpha}(x) = \overline{\beta}(x)$.

- $\alpha \leq_r \beta$ if a reduction exists
- $\mathfrak{m}$ implements α if $\alpha \leq_r \mathfrak{r}_{\mathfrak{m}}$

Implementations of elementary algorithms

Theorem (with Paschalis)

For any recursive program A in an algebra $\mathbf{M}$*, the standard implementation of A is an implementation of* $\mathfrak{r}(A, \mathbf{M})$

. . . *Uniformly* enough, so that (with the full definitions), the standard implementation of A implements the elementary algorithm expressed by A in $\mathbf{M}$

. . . And this is true of all familiar implementations of recursive programs

. . . so that the basic (complexity and resource use) upper and lower bounds established from the program A hold of all implementations of A

And for the applications to complexity theory, we work directly with the recursive equations of A