Qual'è la natura dell'informatica?

Alcune note sul dibattito in corso

Selezione di estratti raccolti e commentati da Claudio Mirolo

Il problema di definire cosa sia l'*informatica*, o *computer science* per precisarne l'accezione utilizzando la terminologia anglosassone, può essere affrontato, in prima istanza, andando ad analizzare i modelli curriculari, in particolare quelli proposti dalle associazioni ACM e IEEE-CSE, che si sono occupate di questo problema a livello internazionale. L'attenzione è allora rivolta a come l'informatica viene insegnata in termini di contenuti e di approcci metodologici (struttura dei contenuti). Si presume, inoltre, che i modelli considerati riflettano sia le tematiche di ricerca, sia la pratica professionale.

Tuttavia, ai fini di una riflessione più approfondita sulla natura e sui fondamenti della disciplina, è interessante considerare anche il dibattito epistemologico e filosofico, recentemente sede di contributi più significativi che nel passato, che ci consente di diventare maggiormente consapevoli della ricchezza e dell'articolazione dell'ambito informatico e, conseguentemente, delle concezioni che se ne possono avere.

In queste pagine cerco di presentare una breve antologia, tuttaltro che esaustiva, di opinioni estratte da alcuni articoli e interventi sul tema, opinioni che si caratterizzano per i diversi punti di vista e per la varietà degli aspetti che tendono a mettere in luce. Il mio intento non è di suggerire l'adesione a una chiave di lettura piuttosto che ad un altra, ma principalmente di stimolare la riflessione e una presa di coscienza dell'urgenza di chiarirsi le idee in proposito, proprio ai fini dell'attività didattica, pur sapendo che al momento ogni scelta non potrà che avere una forte impronta soggettiva.

Una parte degli estratti che ho riportato rappresentano le opinioni di ricercatori dell'ambito informatico; solo in alcuni casi esprimono un approccio propriamente filosofico, e in questi casi i ruoli di informatico ed epistemologo tendono a confondersi perché la formazione degli autori li comprende entrambi.

April 6, 2013

0. Quali domande si pongono sulla natura dell'informatica

Il primo estratto, in particolare, presenta la parte introduttiva di un corso sull'argomento, e l'ho scelto per la chiarezza e completezza con cui pone le domande chiave sulla natura dell'informatica. Probabilmente le prime riflessioni filosofiche hanno interessato il campo dell'intelligenza artificiale (ultima sezione di domande dell'estratto), ma, a parte alcuni cenni, domande di questo tipo resteranno un po' al margine perché, a mio avviso, sono di pertinenza di un ambito un po' più vasto della computer science, prettamente multidisciplinare, e meriterebbero una discussione a se stante.

William J. Rapaport, "Philosophy of Computer Science: An Introductory Course", Teaching Philosophy 28(4): 319-341, 2005.

What is computer science?
[Although the "final" answer to this question may simply be the extensional "whatever computer scientists do", this is a reasonable issue to discuss, even if there is no intensional answer. The following subquestions indicate some of the interesting issues that this main question raises.]
(a) What is science? What is engineering?
(b) Is computer science a science? Or is it a branch of engineering?
(c) If it is a science, what is it a science of?
(d) Is it a science of computers (as some authors say)?
(e) What, then, is a computer?
(f) Or is computer science a science of computation (as other authors say)?
(g) What, then, is computation?
(h) What is an algorithm? Is an algorithm different from a procedure? Many authors say that an algorithm is (like) a recipe; is it, or are there important differences?
(i) What are Church's and Turing's "theses"?
(j) Some authors claim that there are forms of computation—often lumped together under the rubric "hypercomputation"—that, in some sense, go "beyond" Turing-machine (TM) computation: What is "hypercomputation"?

What is a computer program?
(a) What is the relation of a program to that which it models or simulates? What is simulation?
(b) Are programs (scientific) theories?
(c) What is an implementation?
(d) What is software? How does it relate to hardware?
(e) Can (or should) computer programs be copyrighted, or patented?
(f) Can computer programs be verified?

What is the philosophy of artificial intelligence?
(a) What is AI?
(b) What is the relation of computation to cognition?
(c) Can computers think?
(d) What are the Turing Test and the Chinese Room Argument?

Riporto qui di seguito ampi estratti da un contributo pubblicato in un numero monografico della rivista *Mind and Machines* dedicato alla filosofia dell'informatica (*philosophy of computer science*). Mi sembra che in questo articolo l'autore imposti molto chiaramente i problemi dell'inquadramento epistemologico della disciplina, distinguendo i principali approcci (*matematico*, *ingegneristico* e *scientifico*), relativamente ai presupposti (espliciti o impliciti) che li sottendono quando si considerino i *metodi* applicati per acquisire nuove conoscenze, la natura degli oggetti studiati (*ontologia*) e la natura della conoscenza su di essi (*epistemologia*).

Amnon H. Eden, "Three Paradigms of Computer Science", Minds and Machines, 17(2), 2007.

We examine the philosophical disputes among computer scientists concerning methodological, ontological, and epistemological questions: Is computer science a branch of mathematics, an engineering discipline, or a natural science? Should knowledge about the behaviour of programs proceed deductively or empirically? Are computer programs on a par with mathematical objects, with mere data, or with mental processes? We conclude that distinct positions taken in regard to these questions emanate from distinct sets of received beliefs or *paradigms* within the discipline:

– The *rationalist paradigm*, which was common among theoretical computer scientists, defines computer science as a branch of mathematics, treats programs on a par with mathematical objects, and seeks certain, a priori knowledge about their 'correctness' by means of deductive reasoning.
– The *technocratic paradigm*, promulgated mainly by software engineers and has come to dominate much of the discipline, defines computer science as an engineering discipline, treats programs as mere data, and seeks probable, a posteriori knowledge about their reliability empirically using testing suites.
– The *scientific paradigm*, prevalent in the branches of artificial intelligence, defines computer science as a natural (empirical) science, takes programs to be entities on a par with mental processes, and seeks a priori and a posteriori knowledge about them by combining formal deduction and scientific experimentation.

[...]

[...] we expand on the arguments of *complexity*, *non-linearity*, and *self-modifiability* for the unpredictability of programs and conclude that knowledge concerning certain properties of all but the most trivial programs can only be established by conducting scientific experiments.

[...]

The Methodological Dispute

[...] Mathematical methods of investigation guide the research in computability, automata theory, computational complexity, and the semantics of programming languages; design rules of thumb, extensive testing suites, and regimented development methods dominate the branches of software engineering, design, architecture, evolution, and testing; and the methods of natural sciences, which combine mathematical theories with scientific experiments, govern the research in artificial intelligence, machine learning, evolutionary programming, artificial neural networks, artificial life, robotics, and modern formal methods. [...]

The dispute concerning the definition of the discipline and its most appropriate methods of investigation can thus be paraphrased as follows:
Is computer science a branch of mathematics, on a par with logic, geometry, and algebra; is it an engineering discipline, on a par with chemical or aeronautical engineering; or is it indeed a natural, experimental (empirical) science, on a par with astronomy and geology? Should computer scientists rely primarily on deductive reasoning, on test suites and regimented software development process, or should they employ scientific practices which combine theoretical analysis with empirical investigation? How is the notion of a scientific experiment different from a test suite, if at all? What is the relation between theoretical computer science and computer science?

[...]

The Ontological Dispute

We take the notion of a computer program to be central to computer science. In this paper we focus our discussion in the ontological dispute concerning the nature of programs. [...]

We take into consideration all sorts of entities that computer scientists conventionally take to be 'computer programs', such as numerical analysis programs, database and World Wide Web applications, operating systems, compilers/interpreters, device drivers, computer viruses, genetic algorithms, network routers, and Internet search engines. We shall thus restrict most of our discussion to such conventional notions of computer programs, and generally assume that each is encoded for and executed by silicon-based von-Neumann computers. We therefore refrain from extending our discussion to the kind of programs that DNA computing and quantum computing are concerned with.

The ontological dispute in computer science may be recast in the terminology we shall introduce below as follows:
Are program-scripts mathematical expressions? Are programs mathematical objects? Alternatively, should program-scripts be taken to be just 'a bunch of data' and the existence of program-processes dismissed? Or should programscripts be taken to be on a par with DNA sequences (such as the genomic information representing a human), the interpretation of which is on a par with mental processes?

[...] We seek to distinguish between two fundamentally distinct senses of the term 'program' in conventional usage: The first is that of a static script, namely a wellformed sequence of symbols in a programming language, to which we shall refer as a *program-script*. The second sense is that of a process of computation generated by 'executing' a particular program-script, to which we shall refer as a *program-process*. Any mention of the term 'program' shall henceforth apply to both senses.

[...]

The Epistemological Dispute

[...] Most specifications however are not quite as simple [...]. For this reason, fully formulated specifications are not always feasible [...]. Indeed, although the correctness of a program can be a source of considerable damage, or even a matter of life and death, it may be very difficult—or, as Fetzer and Cohn claimed, altogether impossible—to establish formally. [...]

These questions are at the heart of the epistemological dispute:
Is warranted knowledge about programs a priori or a posteriori? In other words, does knowledge about programs emanate from empirical evidence or from pure reason? What does it mean for a program to be correct, and how can this property be effectively established? Must we consider correctness to be a welldefined property—should we insist on formal specifications under all circumstances and seek to prove it deductively—or should we adopt a probabilistic notion of correctness ('probably correct') and seek to establish it a posteriori by statistical means?

[...]

The Rationalist Paradigm

By the rationalist paradigm we refer to that paradigm of computer science which takes the discipline to be a branch of mathematics, the tenets of which have been common among scientists investigating various branches of theoretical computer science, such as computability and the semantics of programming languages. [...]

The Rationalist Methods

[...] The rationalist stance in the methodological dispute can thus be summarized as follows:

Computer science is a branch of mathematics, writing programs is a mathematical activity, and deductive reasoning is the only accepted method of the investigating programs. [...]

The Rationalist Ontology

[...] the rationalist position in the ontological dispute [...] can be recast and justified as follows:
Program-scripts are mathematical expressions. Mathematical expressions represent mathematical objects. A program *p* is that which is fully and precisely represented by [its script] s_p. Therefore *p* is a mathematical object. [...]

The Rationalist Epistemology

[...] The rationalist epistemological position can thus be recast as follows:
Programs can be fully and formally specified, and their 'correctness' is a well-defined problem. Certain, a-priori knowledge about programs emanates from pure reason, proceeding from self-evident axioms to the demonstration of theorems by means of formal deduction. A-posteriori knowledge is to be dismissed as anecdotal and unreliable.

[...]

The Technocratic Paradigm

The Technocratic Paradigm By the 'technocratic paradigm' we refer to that paradigm of computer science which defines the discipline as a branch of engineering, proponents of which dominate the various branches of software engineering, including software design, software architecture, software maintenance and evolution, and software testing. In line with the empiricist position in traditional philosophy, the technocratic paradigm holds that reliable, a posteriori knowledge about programs emanates only from experience, whereas certain, a priori 'knowledge' emanating from the deductive methods of theoretical computer science is either impractical or impossible in principle.

The Technocratic Methods

Wegner describes [...] the emergence of the technocratic paradigm [in the 1970s], echoing what we shall refer to as the *argument of complexity* [...].

[...] the technocratic doctrine contends that there is no room for theory nor for science in computer science. During the 1970 this position, promoted primarily by software engineers and programming practitioners, came to dominate the various branches of software engineering. Today, the principles of scientific experimentation are rarely employed in software engineering research. [...] Instead of conducting experiments, software engineers use testing suites, the purpose of which is to establish statistically the reliability of specific products of the process of manufacturing software. [...]

The position of the technocratic paradigm concerning the methodological dispute can thus be recast as follows:
Computer science is a branch of engineering which is concerned primarily with manufacturing reliable computing systems, a quality determined by methods of established engineering such as reliability testing and obtained by means of a regimented development and testing process. For all practical purposes, the methods of theoretical computer science are dismissed as 'naval gazing'. [...]

The Technocratic Epistemology

[...] The technocratic position concerning the epistemological dispute may be recast in terms of the argument of complexity as follows:
It is *impractical* to specify formally or to prove deductively the 'correctness' of a complete program. A priori, certain knowledge about the behaviour of actual programs is therefore unattainable. If at all meaningful, 'correctness' must be taken to mean tested and proven 'reliability', a posteriori knowledge about which is measured in probabilistic terms and established using extensive testing suites.

Fetzer (1993) and Avra Cohn (1989) offer what is essentially an ontological argument for an even stronger epistemological position [...]. According to this argument, a priori knowledge about the behaviour of machines is *impossible* in principle [...].

Peter Markie (2004) defines empiricism as that school of thought which holds that sense experience is the ultimate source of all our concepts and knowledge. Empiricism rejects pure reason as a source of knowledge, indeed any notion of a priori, certain knowledge, claiming that warranted beliefs are gained from experience. Thus, [this view is] in line with the empiricist philosophical position. [...]

The Technocratic Ontology

[...] Motivated by an underlying concern for ontological parsimony, and in particular the proliferation of universals in the platonist's putative sphere of abstract existence, the nominalist principle commonly referred to as Occam's Razor ("don't multiply entities beyond necessity") denies the existence of abstract entities. By this ontological principle, nothing exists outside of concrete particulars, including not entities that are 'that which is fully and precisely defined by the program script'. The existence of a program is therefore unnecessary.

The technocratic ontology can thus be summarized as follows:
'That which is fully and precisely represented by a script s_p' is a putative abstract (intangible, non-physical) entity whose existence is not supported by direct sensory evidence. The existence of such entities must be rejected. Therefore, 'programs' do not exist.

[...]

The Scientific Paradigm

The scientific paradigm contends that computer science is a branch of natural (empirical) sciences, on a par with “astronomy, economics, and geology” (Newell and Simon 1976), the tenets of which are prevalent in various branches of AI, evolutionary programming, artificial neural networks, artificial life (Bedau 2004), robotics (Nemzow 2006), and modern formal methods (Hall 1990). Since many programs are unpredictable, or even ‘chaotic’, the scientific paradigm holds that a priori knowledge emanating from deductive reasoning must be supplanted with a posteriori knowledge emanating from the empirical evidence by conducting scientific experiments. Since program-processes are temporal, non-physical, causal, metabolic, contingent upon a physical manifestation, and nonlinear entities, the scientific paradigm holds them to be on a par with mental processes.

The Scientific Methods

The scientific notion of *experiment* must be clearly distinguished from the technocratic notion of a *reliability test*. The purpose of a reliability test is to establish the extent to which a program meets the needs of its users, whereas a scientific experiment is designed to corroborate (or refute) a particular hypothesis. If a test suite fails, the subject of experiment (the program) must be revised (or discarded); if an experiment ‘fails’, the theory must be revised (or discarded), or else the integrity of the experiment is in doubt. [...]

For this reason, experiments with programs go beyond establishing the usability of a particular manufactured artefact, even beyond the ‘extent and limitations of mechanistic explanation’. [...]

If computer science is indeed a branch of natural sciences then its methods must also include deductive and analytical methods of investigation.

From this Wegner (1976) concludes that theoretical computer science stands to computer science as theoretical physics stands to physical sciences: deductive analysis therefore plays the same role in computer science as it plays in other branches of natural sciences. [...]

To summarize, the scientific position concerning the methodological question can [be stated] as follows:
Computer science is a natural science on a par with astronomy, geology, and economics, any distinction between their respective subject matters is no greater than the limitations of scientific theories. Seeking to explain, model, understand, and predict the behaviour of computer programs, the methods of computer science include both deduction and empirical validation. Theoretical computer science therefore stands to computer science as theoretical physics stands to physics.

The Scientific Epistemology

[...] from the compelling arguments of *complexity*, *self-modifiability*, and *non-linearity* for the unpredictability of programs, the behaviour of some programs is inevitably a source of a surprise, and a priori knowledge about them is severely limited. [...]

The tenets of the scientific epistemology can therefore be summarized as follows:
While it may be possible *in principle* to deduce some of the properties of the program and all the consequences of executing it, *in practice* it is very often impossible. Therefore, while some knowledge about programs can be established *a priori*, much of what we know about programs must necessary be limited to some probabilistic, *a posteriori* notion of knowledge.

The Scientific Ontology

[...] The scientific ontology and the arguments in its favour can thus be summarized as follows:
Program-scripts are on a par with DNA sequences, in particular with the genetic representation of human organs such as the brain, the product of whose execution—program-processes—are on a par with mental processes: temporal, non-physical, causal, metabolic, contingent upon a physical manifestation, and non-linear entities.

Cercando di riassumere le caratterizzazioni in mdo schematico:

	Paradigma razionalista	**Paradigma tecnocratico**	**Paradigma scientifico**
Metodo	l’informatica fa parte della matematica ragionamento deduttivo	l’informatica fa parte dell’ingegneria test di affidabilità	l’informatica fa parte delle scienze teoria (formulazione di ipotesi) → deduzione → validazione empirica
Ontologia	testo del programma = espressione matematica programma = oggetto matematico	testo del programma = aggregato di meri dati il programma è un’entità immateriale, perciò è inutile presupporne l’esistenza	testo del programma assimilabile a una rete di neuroni programma assimilabile a un processo mentale
Epistemologia	specifiche formali dei programmi complete la correttezza è ben definita conoscenza a-priori	specifiche formali dei programmi impraticabili (o impossibili) ha senso parlare solo di affidabilità conoscenza a-posteriori (misure di affidabilità)	specifiche formali dei programmi incomplete proprietà certe / proprietà stimate conoscenza a-priori (parziale) e a-posteriori (probabilistica)

1. Un punto di vista centrato sull'espressività dei programmi: "epistemologia procedurale"

Se rifletto sul mio approccio all'insegnamento dell'informatica, con un'ottica prevalentemente orientata alla programmazione, e sugli obiettivi che mi pongo, probabilmente il punto di vista autorevole di Gerald Sussman coglie alcuni aspetti sostanziali che mi sembra di condividere, ma che non saprei dire fino a che punto possano essere ritenuti rappresentativi dell'ambito informatico in tutta la sua complessità.

Gerald Jay Sussman, "The Legacy of Computer Science", in Computer Science: Reflections on the Field, Reflections from the Field, The National Academies Press, 2004.

> But, as I have pointed out (H. Abelson, G.J. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd Edition, MIT Press, Cambridge, Mass., 1996):
>
> Computer Science is not a science, and its ultimate significance has little to do with computers. The computer revolution is a revolution in the way we think and in the way we express what we think. The essence of this change is the emergence of what might best be called *procedural epistemology*—the study of the structure of knowledge from an imperative point of view, as opposed to the more declarative point of view taken by classical mathematical subjects. Traditional mathematics provides a framework for dealing precisely with notions of "what is." Computation provides a framework for dealing precisely with notions of "how to."
>
> Computation provides us with new tools to express ourselves. This has already had an impact on the way we teach other engineering subjects. For example, one often hears a student or teacher complain that the student knows the "theory" of the material but cannot effectively solve problems. We should not be surprised: the student has no formal way to learn technique. We expect the student to learn to solve problems by an inefficient process: the student watches the teacher solve a few problems, hoping to abstract the general procedures from the teacher's behavior with particular examples. The student is never given any instructions on how to abstract from examples, nor is the student given any language for expressing what has been learned. It is hard to learn what one cannot express.
>
> [...]
>
> Traditionally, we try to communicate these skills by carefully solving selected problems on a blackboard, explaining our reasoning and organization. We hope that the students can learn by emulation, from our examples. However, the process of induction of a general plan from specific examples does not work very well, so it takes many examples and much hard work on the part of the faculty and students to transfer the skills.
>
> However, if I can assume that my students are literate in a computer programming language, then I can use programs to communicate ideas about *how to* solve problems: I can write programs that describe the general technique of solving a class of problems and give that program to the students to read. Such a program is precise and unambiguous —it can be executed by a dumb computer! In a nicely designed computer language a well-written program can be read by students, who will then have a precise description of the general method to guide their understanding. With a readable program and a few well-chosen examples it is much easier to learn the skills. Such intellectual skills are very hard to transfer without the medium of computer programming. Indeed, "a computer language is not just a way of getting a computer to perform operations but rather it is a novel formal medium for expressing ideas about methodology. Thus programs must be written for people to read, and only incidentally for machines to execute" (Abelson et al., *Structure and Interpretation of Computer Programs*, 1996).
>
> [...] In our class computational algorithms are used to express the methods used in the analysis of dynamical phenomena. Expressing the methods in a computer language forces them to be unambiguous and computationally effective. Students are expected to read our programs and to extend them and to write new ones. The task of formulating a method as a computer-executable program and debugging that program is a powerful exercise in the learning process. Also, once formalized procedurally, a mathematical idea becomes a tool that can be used directly to compute results.
>
> We may think that teaching engineering and science is quite removed from daily culture, but this is wrong. Back in 1980 (a long time ago!) I was walking around an exhibit of primitive personal computers at a trade show. I passed a station where a small girl (perhaps 9 years old) was typing furiously at a computer. While I watched, she reached over to a man standing nearby and pulled on his sleeve and said: "Daddy! Daddy! This computer is very smart. Its BASIC knows about recursive definitions!" I am sure that her father had no idea what she was talking about. But notice: the idea of a recursive definition was only a mathematician's dream in the 1930s. It was advanced computer science in the 1950s and 1960s. By 1980 a little girl had a deep enough operational understanding of the idea to construct an effective test and to appreciate its significance.

Attenzione, però: la conoscenza procedurale, in se, è anche una delle più primitive forme di conoscenza. "Ricette" per perseguire determinati obiettivi possono essere apprese dagli individui di molte specie di animali superiori. Ciò, in particolare, se l'unico operatore in gioco è la composizione sequenziale delle varie azioni. Le conoscenze "proposizionali" della matematica, e delle scienze, sono da questo punto di vista ben più astratte e difficili da capire—quindi, da apprendere. Quello che caratterizza l'informatica, secondo questa accezione, non è dunque una mera collezione di ricette, più specificamente di algoritmi, per risolvere questo o quel problema, ma la capacità di *capire* che ricette complesse effettivamente *risolvono ogni istanza* del problema in esame. Cioè la ricetta (come?) assieme alla sua logica (perché)—ed eventualmente anche assieme alla caratterizzazione dei termini entro cui può essere applicata (con quali risorse?).

Provo ad esemplificare il contenuto di questa mia osservazione riportando, come aneddoto, parte di una discussione che alcuni anni or sono è seguita alla questione posta da una specializzanda (abilitanda nella calsse A042), se fosse cioè più difficile apprendere il concetto matematico di *funzione* o quello informatico di *algoritmo*. I nostri punti di vista divergevano al riguardo, ma questo probabilmente perché la nostra interpretazione della sostanza di un algoritmo era diversa. Fino a prova contraria, resto infatti convinto che l'algoritmo sia un concetto ben più complesso, che implica quello di funzione (nel senso matematico) come prerequisito: astrarre la relazione funzionale fra i dati di un problema e la relativa soluzione è un passo che viene prima di qualunque tentativo di immaginarne una strategia risolutiva generale. Frequentemente, a mio avviso, quando uno studente sostiene di "avere capito un algoritmo", la sua affermazione dovrebbe essere interpretata nel senso che lo studente ritiene di "essere in grado di simularne il comportamento su alcune specifiche istanze del problema" (cosa che, per inciso, il computer sa fare addirittura meglio) e tuttalpiù "intuisce" una certa relazione con la strategia che lui stesso avrebbe adottato per risolvere quel problema. Non è detto, però, che sia anche in grado di formalizzare autonomamente questa sua intuizione. In realtà, noi non dobbiamo dare per scontato che lo studente capisca l'algoritmo nel senso che interessa l'informatica: ciò su cui si concentra la sua attenzione, in genere, è il modello di calcolo sottostante. D'altro canto, per parlare di un algoritmo dobbiamo presumere che il modello di calcolo sottostante sia già acquisito, cioè che almeno su questo l'interpretazione degli interlocutori sia concorde. (Ovviamente, si pone anche il problema di creare le condizioni in cui lo studente possa acquisire familiarità con un modello computazionale, ma questo è un aspetto diverso.)

Al di là della mancanza di accordo sulla definizione (o sulla definibilità in un senso indipendente da specifici modelli computazionali), le trattazioni degli algoritmi che si trovano nei testi introduttivi spaziano fra i due estremi rappresentati dalla familiarizzazione con il modello computazionale da un lato e dalla piena "astrazione algoritmica" dall'altro, con uno sbilanciamento, però, verso l'accezione più banale, in quanto le soluzioni algoritmiche sono raramente argomentate in modo convincente.

Da questo punto di vista, è interessante il seguente intervento di Lars Janlert, che giudica insoddisfacente una concezione di programma che si limiti a descrivere un processo risolutivo (*come?*), ma non includa la sua giustificazione razionale (*perché?*). Si tratta in sostanza dello stesso problema sollevato sopra: spiegare perché un programma risolve un certo problema vuol dire (almeno) argomentare la correttezza dell'algoritmo realizzato da quel programma.

Lars-Erik Janlert, "The program is the solution—what is the problem?", European Conference on Computing and Philosophy, 2006.

> [...] in addition to being causative and descriptive, we should also require of a program that it be rationally justified: that the specific structure of the program can be explained by the (rational) ways in which it contributes to achieving the goal of the intended process. Non-traditional ways of producing programs, "metamethods" such as evolutionary programming, do not intrinsically involve the production of a rationale. Moreover, they are not restricted to be easily (or at all) rationalized after the fact. If the requirement of rationality is accepted and if the worries that metaprogramming sometimes results in products difficult to rationalize are taken seriously, the question arises whether we can comfortably accept that the requirement of rationality is only satisfied at the metalevel? If the use of metaprogramming spreads, and we are *not* satisfied with incomprehensible "programs," then the business of programming may take a turn from problem-solving in the sense of invention and engineering towards problem-solving in the sense of explanation and research.
> I propose the following three necessary criteria for something being a program, formulated in terms of its relation to the intended process:
>
> 1) *causative* – the program causes the process;
> 2) *descriptive* – the program describes the process;
> 3) *justified* – the program has a rational explanation in terms of the goal of the process; i.e. the structure of the program can be explained by the (rational) ways in which it contributes to achieving the goal.
>
> I expect points 1 and 2 to correspond well to generally held views, whereas point 3 may have some news value. The requirement of justification mirrors the reasonable assumption that a subjective method is *not* just a recipe: it also needs a rationale; it must be a reasoned method (which is why manually simulating the execution of a working "program" cannot in itself be a subjective method).

2. Anche se l'informatica non si occupa della realtà, la scientificità sta nello spirito di ricerca conoscitiva

Nell'intervento che segue, anziché rivolgere la propria attenzione all'oggetto di studio o al metodo della disciplina, Richard Bornat sposta il problema della "scientificità" dell'informatica su un piano diverso, quello della motivazione ideale del ricercatore, ispirato da puro desiderio di conoscenza, non dall'utilità pratica.

Richard Bornat, "Is 'Computer Science' science?", European Conference on Computing and Philosophy, 2006.

> CS could be science, or it could be mathematics, or it could be stamp collecting. One difficulty is that, unless you are a Platonist – which, since programming is constructivist logic, few programmers are at heart – there is no reality to be scientific about, unlike the realities that confront physics, chemistry, biology and sometimes even psychology. Simon [...] made an unconvincing attempt to define a Platonist science of computing. It won't wash: programming is all made up, quite unreal, so we can't be physically scientific about programs or even their executions. Mathematics is made up and unreal too, so far as non-Platonists are concerned. So maybe CS is mathematics after all (there isn't really a case to be made for stamp collecting: hence the failure of software science).
>
> But mathematics is utterly abstracted from reality, the study of the structure of arguments that have not yet been made. CS, despite the fact that programs are completely meaningless, is nailed to reality. Programs drive trains, fly planes, work your radio, live inside your TV remote control, mess you up at every turn. There have been unreal programs – physicists are writing them now, in anticipation of a quantum computer that will work in ways that only they truly believe in – but they are rare and so far unimportant.
>
> It's clear that all of CS has more than a touch of engineering about it. When we devise programming languages, for example, we make them as convenient to use as we can for the construction of programs, at the same time as making them as spare and clean as we can for the purposes of mechanical translation. [...]
>
> CS isn't reality-science and it isn't mathematics and it's at least half engineering. Is there a way that it can be a science? Perhaps we are looking in the wrong direction, at the subject rather than its practitioners. Hoare has proposed [6] that we are scientists because of the way we behave. Engineers compromise, fudge, make things fit. I do that, and I enjoy doing it. But when I'm not programming, I'm a seeker after truth. Scientists, Hoare observed, search ruthlessly for truths without regard for the consequences. That's what physicists, chemists and biologists do, hoping to weigh electrons, purify proteins or get inside the heads of baboons. None of these activities is useful at present, so scientists are a little like mathematicians, but they are all obviously about something real, so scientists aren't mathematicians after all. For myself, I'm sure that I'm sometimes a scientist. In practice I'm always both behind (engineer) and ahead (scientist) of the game.
>
> [...]
>
> Conclusion
>
> Hoare behaves like a scientist most of the time. I do so some of the time. CS is what we do, so CS is science. Science is possible even in the unreal landscape of computing.

Benché interessante come spunto di riflessione, una simile concezione non sembra destinata ad avere significativi riscontri operativi da un punto di vista epistemologico, essendo troppo legata alla soggettività delle motivazioni, e indebolita dalla probabile confluenza di motivazioni di natura diversa nel lavoro dello scienziato. Tuttavia, l'interpretazione di Bornat del lavoro dell'informatico certamente diverge dall'accezione comune, e in questa sede sollecita almeno una domanda importante: come potremmo fare emergere, di fronte ai nostri allievi, l'interesse per la "conoscenza pura" che motiva almeno una parte delle attività dell'ambito informatico?

La rilevanza di questa domanda appare evidente quando si guarda alle matricole universitarie iscritte ai corsi di laurea in informatica dell'Università di Udine. Da un lato molti studenti partono con un'accezione prettamente strumentale della disciplina, e di conseguenza rischiano una delusione delle aspettative quando devono confrontarsi con argomenti caratterizzati scientificamente. D'altro lato, le valutazioni conseguite all'esame di maturità sono mediamente inferiori a quelle dei loro colleghi immatricolati in ingengeria, medicina, economia, lingue straniere o lettere. (Prendendo come riferimento la coorte 2005-06, per esempio, tale valutazione media è 75.8/100, che può essere confrontata con 82.5 di ingegneria, 79.7 di economia, o 77.3 di lettere.) Una possibile interpretazione di queste osservazioni è che, al termine della formazione superiore, ai corsi di laurea in informatica gli studenti non associano dei contenuti scientifici o l'interesse a mettere in gioco le proprie capacità di risoluzione di problemi, come invece ci si aspetta per le vocazioni in matematica o fisica, ma piuttosto si immaginano come utenti di tecnologie. Quindi, ai fini di consentire agli studenti una scelta più consapevole, quando pensano di intraprendere studi accademici in informatica, è necessario che la scuola secondaria sia in grado di delineare un quadro coerente di ciò di cui l'informatica si occupa.

3. L'informatica è una disciplina scientifica

Come esempio di articolata discussione a supporto di una collocazione scientifica dell'informatica, riporto alcuni ulteriori estratti dall'articolo di A.H. Eden. Le argomentazioni fanno riferimento alla caratterizzazione del paradigma scientifico in base agli aspetti metodologici, epistemologici e ontologici cui si è accennato nella sezione introduttiva.

Amnon H. Eden, "Three Paradigms of Computer Science", Minds and Machines, 2007 (citato sopra).

> The *argument of complexity* receives further corroboration from the technological progress during the past three decades since it was first articulated. Since 1979, the average size of programs and operating systems grew in at least four orders of magnitude. More importantly, the complexity of compilers, operating systems, microprocessors, and input is today compounded by component-based software engineering technologies [...]. These technologies gave rise to gigantic programs such as Internet search engines and electronic commerce applications which consist of hundreds of software components (e.g., dynamically linked libraries, server-side and client-side threads), whose construction is often 'outsourced' or otherwise delegated to a range of independent commercial bodies or individual volunteers and which execute on any one of a wide range of microprocessors (i.e., in a 'heterogeneous environment'). The notion of 'input' with regard to these programs has also been much further complicated as signals and data arrive to these programs from innumerable other interacting programs [...] and which communicate via vast and very complex communication networks. Any form of deductive reasoning about such programs requires the representation of petabytes of instructions and data in every one of the components of the program and of every computer, operating system, and network router that is involved (directly or indirectly) in their execution. Since these often change during the lifespan of a program-process, the very notion of a program-script is therefore not well-defined, specifications are not well-defined, and deductive reasoning about their de facto representations is an idealization that is as unrealistic and ineffective as, say, deductive reasoning about the individual atomic particles of airplanes and power stations.
>
> [...]
>
> Additional arguments supporting the relevance of scientific experimentation concern the limits of analytical methods. [Here] we shall examine the *argument of non-linearity* and the *argument of self-modifiability* and conclude that, indeed, knowledge about even some of the simplest programs can only be gained via experiments.
>
> Computer programs can also be used as tools in discovering and empirically establishing the laws of nature. In particular, program simulations can be used to examine the veracity of models of non-linear phenomena [...] in other natural sciences. For example, in cognitive psychology, an artificial intelligence program can be taken to be a tool for empirical examinations of models of memory and learning; in bioinformatics, genetic algorithms are used to test the extent to which models of the reproduction of DNA molecules are corroborated by the laws of Darwinian natural selection; and in astronomy, the predictions of models for the creation of the universe can be tested by means of computer simulations. If computer science is concerned with the 'phenomena surrounding computers'—such as the behaviour of computer simulations—then its subject matter is distinct from any given class of natural phenomena at most in the extent to which scientific theories deviate from reality. In other words, our programs are only 'incorrect' to the extent to which the scientific theories they implement deviate from the phenomena they seek to explain. In Popper's (1963) terms, the difference between programs and the (naturalistic view of) reality is at most limited by the verisimilitude (or truthfulness) of our most advanced scientific theory. The progress of science is manifest in the increase in this verisimilitude. Since any distinction between the subject matter of computer science and natural sciences is taken to be at most the product of the (diminishing) inaccuracy of scientific theories, the methods of computer science are the methods of natural sciences.
>
> But the methods of the scientific paradigm are not limited to empirical validation, as mandated by the technocratic paradigm. Notwithstanding the technocratic arguments to the unpredictability of programs [...], the deductive methods of theoretical computer science have been effective in modelling, theorizing, reasoning about, constructing, and even in predicting—albeit only to a limited extent—innumerable actual programs in countless many practical domains. For example, context-free languages has been successfully used to build compilers (Aho et al. 1986); computable notions of formal specifications (Turner 2005) offer deductive methods of reasoning on program-scripts without requiring the complete representation of petabytes of program and data; and classical logic can be used to distinguish effectively between abstraction classes in software design statements (Eden et al. 2006). [...]
>
> Analytical investigation is used to formulate hypotheses concerning the properties of specific programs, and if this proves to be a highly complex task [...] it nonetheless an indispensable step in any scientific line of enquiry.
>
> [...]
>
> The *argument of complexity* demonstrates that deductive reasoning is impractical for large programs. The following arguments however demonstrate that the outcome of executing even very small programs cannot be determined analytically.
>
> The *argument of self-modifiability* for the unpredictability of programs concerns the fact that certain program-processes modify the very set of their instructions (the program-script) during the process of computation. [...]
>
> The *argument of non-linearity* for the unpredictability of programs relies on the fact that the vast majority of program-processes belong to the deterministically chaotic class of phenomena. [...]
>
> In conclusion from the compelling arguments of complexity, self-modifiability, and non-linearity for the

unpredictability of programs, the behaviour of some programs is inevitably a source of a surprise, and a priori knowledge about them is severely limited. Therefore, while it may be possible in principle to deduce some of the properties of the program and all the consequences of executing it, in practice it is very often impossible.

[...]

We postulate that an adequate ontological explanation for program-processes must offer an account for the following unique set of their apparent properties:

1. Temporal: The existence of program-processes extends in time in the interval between being created and being destroyed;
2. Non-physical: Program-processes are non-physical, intangible entities;
3. Causal: Program-processes can interact with and move physical devices;
4. Metabolic: Program-process ‘consume’ energy;
5. Contingent upon a physical manifestation: The existence of program-processes depends on the existence of that physical computer which is said to be ‘executing’ it;
6. Nonlinear: The outcome of a program-process, in the general case, cannot be analytically determined.

Let us examine briefly the weaknesses of the rationalist and of the technocratic ontological explanations with relation to the apparent properties of programprocesses. Rationalism asserts that programs are mathematical objects. But mathematical objects, such as Turing machines, recursive functions, triangles, and numbers cannot be meaningfully said to metabolize nor have a causal effect on the physical reality in any immediate sense. It would be particular difficult to justify also a claim that mathematical objects have a specific lifespan or that they are contingent upon any specific physical manifestation (except possibly as mental artefacts). [...]

Alternatively, the technocratic paradigm reduces program-scripts to mere “bunches of data”. It is hostile towards assertions of existence of any abstract, ontologically independent manifestations of whatever the data is taken to represent. But program-processes do have causal effect on physical reality: They control robotic arms, artificial limbs, [...] the navigation of automated and semi-automated vehicles, the sale and purchase of stocks in stock exchanges, and to some degree almost every single home appliance. Programs also treat depression [...], count votes in national elections, and spread copies of themselves over the Internet. Program-processes came to have a tangible effect on concrete, physical reality [...].

4. Per inquadrare le scienze moderne, come l'informatica, non bastano più i paradigmi tradizionali

Il lavoro di Gordana Dodig-Crnkovic da cui sono tratti i passi seguenti affronta in modo ampio e competente una parte dei temi evocati da queste mie note. La prospettiva mi sembra pragmatica. Lo sforzo di classificare l'informatica secondo i paradigmi che sono stati utilizzati per le scienze tradizionali, quali la matematica e la fisica, potrebbe risultare sterile, in quanto è cambiato il contesto di sviluppo delle scienze moderne. Tuttavia, all'interno dell'informatica sono riconoscibili, a giudizio dell'autrice, aspetti scientifici ampiamente sufficienti a caratterizzare l'informatica (anche) come disciplina scientifica.

Gordana Dodig-Crnkovic, "Computer Science in a Theory of Science Discourse", Master Thesis in Computer Science, Department of Computer Science, Mälardalen University, 2002.

> Computer Science is a very new field and among the youngest sciences. As a consequence, it has typical modern characteristics: it is interdisciplinary and very close related to technology. It suffers from a lack of its own scientific traditions at the same time as it undergoes a tremendously dynamic development. The need to reflect upon scientific methods in Computer Science is urgent. In classical sciences (as for example physics) the theory of science has a great tradition and it provides essential tools for understanding the structure, logic and modes of growth of knowledge. Theory of science helps define standards for what can be considered as science. Computer technology changes so quickly that knowledge very rapidly becomes out of date. It is vital to concentrate on the science behind the technology in order to assure continuity. The results of theory of science are in one way or another directly useful for the development of science itself. In that sense the ambition of theory of science is not only understanding of common patterns and features within scientific fields, but also assisting progress of science. The aim of this essay is to situate Computer Science among other sciences in a theory of science perspective, as well as to define the scientific discipline of Computer Science as it appears today, in the year of 2002.
>
> [...]
>
> This essay aims to analyze a new science that does not even have a commonly accepted name, in either English or Swedish. The Swedish terms "datalogi" and "datavetenskap" are used with different meaning at different universities and in different contexts exhibiting great diversity of definitions and educational contents. Swedish National Agency for Higher Education, Högskoleverket, uses the term ”datavetenskap” to correspond to English term Computer Science. According to definitions of IEEE-ACM Computing Curricula 2001, the umbrella term for Computer Science, Computer Engineering, Software Engineering and Information Systems is ”Computing”. Some other European languages (German, French) focus more on IT (Information Technology) that is a generic term encompassing even ”Computing”. In Sweden and even in number of other countries IT has a specific and more limited meaning, and it is very often at different universities organizationally separated from the rest of Computing. The above is only an illustration indicating how difficult it is to characterize scientific aspects of this new field that more resemble a dynamic process than a stable equilibrium phenomenon. Nevertheless it is necessary and instructive to try to conceptualize even such a dynamical process and to relate to classical sciences, including mathematics and logics (abstract/formal sciences) as well as natural sciences (and sometimes in certain methodological questions, especially within Software Engineering, even Social Sciences) that are more empirically focused.
>
> [...]
>
> *The discipline of computing* thus encompasses Computer Science, Computer Engineering, Software Engineering and Information Systems. Of course it is impossible to give a simple and unobjectionable definition of Computing and Computer Science. Let me mention some of the existing ones:
>
> 1. The discipline of Computing is the systematic study of algorithmic processes that describe and transform information: their theory, analysis, design, efficiency, implementation, and application.
> 2. Computer Science is the study of phenomena related to computers, Newell, Perlis and Simon, 1967.
> 3. Computer Science is the study of information structures, Wegner, 1968, Curriculum 68.
> 4. Computer Science is the study and management of complexity, Dijkstra, 1969.
> 5. Computer Science is the mechanization of abstraction, Aho and Ullman 1992.
> 6. Computer Science is a field of study that is concerned with theoretical and applied disciplines in the development and use of computers for information storage and processing, Mathematics, Logic, science, and many other areas.
>
> The second definition reflects an empirical tradition since it asserts that Computer Science is concerned with the study of a class of phenomena. The first and third definitions reflect a mathematical tradition since algorithms and information structures are two abstractions from the phenomena of Computer Science.
>
> The third definition was used by Wegner as the unifying abstraction in his book on Programming Languages, Information Structures and Machine Organization. This view of Computer Science has its historical roots in information theory. It strongly influenced the development of Curriculum 68; a document which has been very prominent in the development of undergraduate Computer Science curricula afterwards. It is implicit in the German and French use of the respective terms *"Informatik"* and *"Informatique"* to denote the discipline of Computer Science.
>
> [...]
>
> The fourth definition reflects the great complexity of engineering problems encountered in managing the construction of complex software-hardware systems.

> It is argued [...] that Computer Science was dominated by empirical research paradigms in the 1950s, by mathematical research paradigms in the 1960s and by engineering oriented paradigms beginning with the 1970s.
>
> The diversity of research paradigms within Computer Science may be responsible for the divergences of opinion concerning the nature of Computer Science research.
>
> The fundamental question underlying all computing is: What can be (efficiently) automated? The discipline was born in 1940s through the joining of Mathematical Logic, algorithm theory and the electronic computer.
>
> Logic is important for computing not only because it forms the basis of every programming language, or because of its investigating into the limits of automatic calculation, but also because of its insight that strings of symbols (also encoded as numbers) can be interpreted both as data and as programs.
>
> [...]
>
> In spite of all characteristics that differ the young field of Computer Science from several thousand years old sciences such as Mathematics and Logic, we can draw a conclusion that Computer Science contains a critical mass of scientific features to qualify as a science.
>
> From the principal point of view it is important to point out that all modern sciences are very strongly connected to technology. This is very much the case for Biology, Chemistry and Physics, and even more the case for Computer Science. That is a natural consequence of the fact that the research leading to the development of modern computers very often is conducted within industry.
>
> The engineering parts in the Computer Science have more or less close connection to the hardware (physical) aspects of computer. Software engineering is concerned with the problems of design, construction and maintenance of the, often huge, software systems that are typical of industry.
>
> Theoretical Computer Science, on the other hand, is scientific in the same sense as theoretical parts of any other science. It is based on a solid ground of Logic and Mathematics.
>
> The important difference is that the computer (the physical object that is directly related to the theory) is not a focus of investigation (not even in the sense of being the cause of certain algorithm proceeding in certain way) but it is rather *theory materialized, a tool always capable of changing in order to accommodate even more powerful theoretical concepts*.

In sintesi: le scienze "tradizionali" appaiono meglio fondate epistemologicamente non solo perché la riflessione sui fondamenti è cominciata molto prima, ma anche perché sono nate in un contesto storico in cui la complessità dell'universo scientifico-tecnologico era inferiore a quella attuale e, in particolare, le attività scientifiche erano più facilmente separabili dalle attività tecnologiche. Tuttavia, l'aumento della complessità e la contaminazione interdisciplinare sono una caratteristica dello sviluppo di tutte le scienze moderne.

Un concetto per certi versi analogo, sia pure espresso in modo più radicale, viene evidenziato da Jean-Claude Simard, epistemologo e storico della scienza.

Jean-Claude Simard, "Science, technologie et société", Le Saut Quantique, `www.apsq.org/sautquantique`, 2002. (L'approssimativa traduzione dal francese è mia.)

> [...] il Progetto manhattan ha completamente rivoluzionato i rapporti fra la scienza e la tecnica. Se si eccettua qualche spirito libero come Hawking, una cosa come il pensiero puro ha ancora qualche peso oggi? O l'antica ragione teorica è diventata essenzialmente strumentale? In ogni caso, scienza e tecnica sono al giorno d'oggi difficilmente separabili e si parla a ragione di *technoscience*. [...] Attualmente, la tecnica o precede [...] la scienza o si sviluppa [...] indipendentemente. Molti ricercatori credono perfino, più che mai, che sia autonoma, che evolva secondo una sua propria razionalità, e di conseguenza al di fuori del controllo umano.

Ad integrazione delle "definizioni" di computer science riportate da Gordana Dodig-Crnkovic, è utile aggiungere quelle raccolte da William Rapaport, in particolare perché includono anche i punti di vista di due autorevoli e ben conosciuti pionieri dell'informatica: Donald Knuth e Juris Hartmanis.

W.J. Rapaport, "Philosophy of Computer Science: An Introductory Course", 2005 (citato sopra):

> We surveyed the following answers that have been given to the question "What is computer science?":
>
> - It is a *science of computers and surrounding phenomena* (such as algorithms, etc.) (Newell et al. 1967).
> - It is the *study* (N.B.: not "science") *of algorithms and surrounding phenomena* (such as the computers they run on, etc.) (Knuth 1974).
> - It is the empirical study ("*artificial* science") of the phenomena surrounding computers (Newell & Simon 1976; cf. Simon 1996).
> - It *is* a *natural* science, not of computers or algorithms, but of *procedures* (Shapiro 2001).
> - It is *not* a science, but a branch of *engineering* (Brooks 1996).
> - It is the body of knowledge dealing with information-transforming *processes* (Denning 1985).
> - It is the study of *information* itself (Hartmanis & Lin 1992).

> Note that several of these (especially the first two) might be "extensionally equivalent" but approach the question from very different perspectives: Some emphasize the computer (hardware); others emphasize algorithms, processes, procedures, etc. (software), or even something more abstract (e.g., information). An orthogonal dimension focuses on whether computer science is a science or perhaps something else (a "study", a "body of knowledge", an engineering discipline, etc.). And, of course, the name itself varies (computer science, computing science, informatics, etc.), often for political, not philosophical, reasons.

Inoltre, le interpetazioni più diffuse della computer science vengono messe in discussione da alcuni lavori più recenti di Paul Wegner e altri, che propongono un nuovo paradigma computazionale basato sulle *interazioni*:

> The notion that these characteristics are inherently ouside the traditional algorithmic conceptualization of computation is the basis for this new paradigm for computing, built around the unifying concept of *interaction* (Wegner, 2006).

5. La natura dell'astrazione è diversa nell'informatica e nella matematica

Di tutti gli autori a cui ho fatto riferimento in queste note, Timothy Colburn si pone le domande di natura più marcatamente filosofica: oltre all'analisi epistemologica e logica, nel suo lavoro mette in rilievo i problemi ontologici ed etici.

Timothy Colburn, "What is Philosophy of Computer Science?", European Conference on Computing and Philosophy, 2006.

> Today, study of the models of computation falls within the area computer scientists call the theoretical foundations of computer science, or FCS. The sub-areas of FCS are wide-ranging and encompass myriad topics ranging from automata and formal languages, to algorithms and data structures, to logic and complexity theory. If FCS includes computability and logic, and if these topics spawned Church's thesis, which I call an example of philosophy of computer science, or PCS, are PCS and FCS then the same? I would say not [...]. The point of this is that FCS, despite the "Foundations" descriptor, is simply "pure" computer science, without the pressing constraints of applications. To the extent that the activity of pure science is not to be identified with the philosophy of that science, FCS is not PCS.
>
> [...] I first came across Turing's classic "Computing Machinery and Intelligence." In it, Turing, who can be regarded as an early philosopher of computer science, was, arguably, posing a theory in the philosophy of mind, namely the functional theory of mind championed by many modern cognitive scientists.
>
> [...] I met Jim Fetzer, a philosopher of science who became intensely interested in verification of program correctness, or the attempt by computer scientists to prove that a program meets its specification. This seemingly innocuous aspect of computer science methodology, under Fetzer's analytical lens, became, almost overnight, a contentious battlefield when Fetzer made, in a flagship journal of computer science, the (to my mind) tame observation that a formal program verification can, at best, prove the correctness only of an abstract algorithm embodied by a program, and not the running program itself. I regarded this observation, though not, I thought, a terribly shocking one, as a paradigm example of philosophy of computer science, because it subjected a foundational aspect of a given science, namely program verification, to a characteristically philosophical treatment, namely the analysis of concepts of that science and the critical evaluation of its beliefs.
>
> In observing the firestorm and debate that ensued following Fetzer's article, it occurred to me that the controversy seemed to result from the seeming incompatibility of two points of view regarding computer science: one looking at computer science as an experimental science, situated in an uncertain world where people, their behavior, and the vaguely described problems they have to solve hold sway; and one looking at computer science as solution engineering, where problems are understood and described completely in advance of their solution, which amounts to the application of perhaps formal procedures in the accomplishing of that solution, in much the same way as mathematical reasoning, and perhaps discovery, proceeds. I attempted to reconcile these points of view by stepping back and seeing them both in the purview of computer science "in the large."
>
> [...]
>
> My own view is that philosophy, at its core, is concerned with the concept of knowledge (epistemology), reasoning (logic), existence (ontology), and value (ethics), and that a "philosophy of X" addresses these concepts within the domain and conceptual framework of X. [...]
>
> What, then, might be a central question for PCS? [...] The controversy surrounding Fetzer's paper 16 years ago seemed to point to friction between those who embrace a mathematical paradigm for computer science and those who see computer science as more of an empirical discipline. [...] I have come to regard computer science as a discipline non-subservient to mathematics but dependent upon it to no more or less an extent than any other natural science is in the construction of needed mathematical models. Gary Shute and I have thought about the kinds of abstraction that occur in computer science and mathematics and found them strikingly different. Since mathematics and its methods have been well ensconced for centuries, then perhaps the abstraction methods that are used in software development are philosophically novel? And if not, have these abstraction methods been employed before in other areas of inquiry? These, at least for Shute and I, are current central questions in PCS.
>
> [...]
>
> Finally, what is the relationship between abstraction and ontology in software development? What is the ontological status of an object in a running object-oriented program? What kinds of objects are involved in a computational process? Do they depend on the kinds of objects described in a source program? Nearly 60 years ago, right around the dawn of the age of digital electronic computation, the philosopher W. V. O. Quine made a radical claim for the time. In trying to deal with conundrums involving existence, and in adjudicating among various philosophies of mathematics, he said "To be is to be the value of a bound variable." His point was that one's language has much to do with one's ontology. He was quick to say that such a formula cannot say what exists, but only what a given theory says exists. As object-oriented programmers, our language says that things as various as shopping carts, chat rooms, and network sockets exist, in some sense, in our computational processes. Have we, in 60 years, come any closer to saying what there is in a computational process? Do our powers of abstraction as programmers have any effect on what there is? I believe that PCS should have something to say about this.

Probabilmente è necessaria una preparazione filosofica di base per cogliere pienamente la portata dell'analisi di Colburn, inquadrandola nel dibattito filosofico che ha attraversato un paio di millenni. Almeno a livello intuitivo, comunque, se consideriamo la struttura delle tecnologie informatiche, che nascono dalla commistione di componenti fisiche (hardware) ed entità immateriali (software), e se ne teniamo presente l'impatto sulla realtà, possiamo anche intuire come si riproponga in una forma nuova il dualismo conosciuto dai filosofi che nel corso dei secoli hanno affrontato la dicotomia mente-corpo. Alcuni problemi di natura filosofica, inoltre, sono destinati ad avere importanti conseguenze sul piano pratico. È il caso dell'ontologia del software, che nella sostanza sta alla base delle recenti controversie in sede di Commissione e Parlamento Europeo, intese a definirne l'interpretazione dal punto di vista giuridico: il software *è* una macchina, quindi brevettabile, oppure *è* un'opera di ingegno, a cui si applicano i diritti d'autore? Qual'è la vera natura del software, separato da ogni specifica macchina da cui potrebbe essere eseguito?

Un estratto da un altro intervento di Colburn nell'ambito della stessa conferenza distingue invece l'astrazione in matematica dall'astrazione in informatica.

Timothy Colburn & Gary Shute, "Abstraction in Computer Science", European Conference on Computing and Philosophy, 2006.

> All legitimate sciences build and study mathematical models of their subject matter. These models are usually intended to facilitate the acquisition of knowledge about an underlying concrete reality. Their role is one of supporting scientific reasoning in an attempt to discover explanations of observed phenomena. Of course, empirical sciences also employ nonmathematical models through arrangements of experimental apparatus by which hypotheses concerning the nature of physical, or perhaps social, reality are tested in an effort to uncover explanatory or descriptive laws of nature.
>
> Traditional empirical sciences are thus concerned with both concrete models in the form of experimental apparatus, and abstract models in the form of mathematics. Computer science, insofar as it is concerned with software, is distinguished from the empirical sciences in that the *only* models it builds and studies are abstractions. Since mathematics is also distinguished from the empirical sciences in that the types of models it builds and studies are abstractions, one may be tempted to infer that there is a close relationship between computer science and mathematics, even to infer that computer science is a subspecies of mathematics. We show that this is only a surface similarity, and that the fundamental nature of abstraction in computer science is quite different from that in mathematics.
>
> [...]
>
> We argue that the primary abstract subject matter of mathematics is *inference patterns*, while the primary abstract subject matter of computer science is *interaction patterns*. This is a crucial difference, and shapes the kind of abstraction used in the two disciplines.
>
> [...] the central activity of computer science is the production of software, and this activity is characterized primarily not by the creation and exploitation of inference patterns, but by the modeling of interaction patterns. The kind of interaction involved depends upon the level of abstraction used to describe programs. At a basic level, software prescribes the interacting of a certain part of computer memory, namely the program itself, and another part of memory, called the program data, through explicit instructions carried out by a processor. At a different level, software embodies algorithms that prescribe interactions among subroutines, which are cooperating pieces of programs. At a still different level, every software system, whether it is an operating system, a networked system distributed over multiple machines, or a single user program, is a carefully orchestrated interaction of entities variously called tasks, threads, or processes. Today's extremely complex software is possible only through abstraction levels that leave machine-oriented concepts behind. Still, these levels are used to describe interaction patterns, whether they be between software objects or between a user and a system.
>
> [...] The formalism of mathematics is relatively *monolithic*, based on set theory and predicate calculus. [...] Computer science, as we have seen, is about interaction patterns. The myriad kinds of such patterns, and the injection into them of the human element, leads to a formalism of computer science that is *pluralistic* and *multilayered*, involving multiple programming languages, alternative software design notations, and diverse system interaction patterns embodied by language compilers, operating systems, and networks.
>
> [...]
>
> Any science succeeds in part by constructing formal mathematical models of their subject matter that eliminate inessential details. Inasmuch as such details constitute information, we might call such elimination *information neglect*. It proceeds by dropping information from consideration as though it does not exist. Computer science, however, cannot afford to treat its information this way. It must use abstraction to manage the complexity involved in modeling ever increasing kinds of interaction. Computer science is therefore distinguished from mathematics in the use of a kind of abstraction that computer scientists call *information hiding*. The complexity of behaviour of modern computing devices makes the task of programming them impossible without abstraction tools that hide, but do not neglect, details that are essential in a lower-level processing context but inessential in a software design and programming context.

Due punti vanno comunque tenuti presenti, a mio giudizio, nell'interpretare questi passi. Primo punto: non è esattamente vero che l'astrazione in informatica non "trascura", ma semplicemente "nasconde" i dettagli. I modelli dell'informatica, come quelli di ogni altra scienza, per quanto complessi, eliminano del tutto alcuni dettagli. Altrimenti non sarebbero dei modelli, ma sarebbero la realtà stessa. Quello che succede, in informatica, è che coesistono livelli di astrazioni diversi, e non solo per finezza di risoluzione, ma anche per il punto di vista che rappresentano (questo è ben presente nella programmazione orientata agli oggetti). Secondo punto: se confrontiamo la matematica e l'informatica in questi termini, significa che scegliamo la prospettiva delle scienze "empiriche", quindi guardiamo alla matematica come matematica applicata, i cui modelli trascurano determinati aspetti della realtà modellata a vantaggio di altri. Secondo l'accezione della matematica come scienza "pura", infatti, questo problema non si pone, in quanto le strutture studiate sono precisamente gli oggetti di studio, non i modelli di qualche altra entità.

Alcuni passi sono argomentati più chiaramente in un articolo successivo, di cui qui riprendo alcuni estratti ad integrazione di quanto già riportato sopra.

Timothy Colburn & Gary Shute, "Abstraction in Computer Science", Minds and Machines, 17(2), 2007.

> Computer science is rich with references to various entities characterized as *abstract* and various activities characterized as *abstraction*. Programming, for example, involves the definition of *abstract data types*. Programming languages facilitate varying levels of *data abstraction* and *procedural abstraction*. Language architects specify *abstract machines*. One computer science textbook even characterizes its subject matter as the science of 'concrete abstractions' (Hailperin et al. 1999). A philosophy of computer science should be able to characterize abstraction as it occurs in computer science, and also to relate it to abstraction as it occurs in its companion, mathematics.
>
> [...]
>
> In summary, abstraction has been characterized in philosophy, mathematics, and logic as the process of eliminating specificity by ignoring certain features.
>
> [...]
>
> The interaction patterns that are at once the most critical from a usability point of view and the most difficult to model using abstraction tools are those interaction patterns involving the human element. This means that there are aspects of computer science that cannot be "abstracted away" to make them cleaner, as is done in mathematics. This point is all too often ignored by those who emphasize a mathematical paradigm for computer science. User interface aspects of programs, because they are unpredictable and subject to exceptional behavior, are sometimes characterized as second-class interaction patterns when compared to the interaction patterns of "more interesting" or "cleaner" internal algorithms and data structures, which are much more amenable to mathematical analysis. As we remarked in the Introduction and will elaborate below, abstraction in mathematics often involves a kind of "information neglect" that conveniently ignores aspects of its subject matter that are considered irrelevant. Unfortunately, in most cases it is not possible to consider user interaction with software irrelevant, and the unpredictability of such interaction is a fact of life for the software modeler. As a corollary, a "pure" computer science that analyses only clean, human-free interaction patterns is impoverished at best.
>
> [...] So abstraction in mathematics facilitates inference, while abstraction in computer science facilitates the modeling of interaction. The difference between these activities can also be seen by considering the respective disciplines' use of formalism.
>
> [...]
>
> Moreover, changes in technology drive changes in computer science ontology, with the result being that the kinds of things that make up the interaction patterns are constantly changing.
>
> [...]
>
> [...] there are at least two kinds of abstraction in mathematics: the emphasis of form over content, and the neglect of certain features in favor of others.

Sul piano didattico è interessante osservare che alcuni importanti proposte di corsi introduttivi alla programmazione (e all'informatica) mettono al centro del programma le forme di astrazione. Esempi notevoli:

M. Hailperin, B. Kaiser, K. Knight, "Concrete Abstractions";

R. Shackelford, "Introduction to Computing and Algorithms";

B. Liskov, J. Guttag, "Program Development in Java".

6. Gli sviluppi dell'informatica richiedono l'adozione di un nuovo paradigma basato sulle interazioni

Come accennato precedentemente, Peter Wegner e altri mettono in discussione i modelli più tradizionali della computer science, in particolare per quanto riguarda il concetto di algoritmo. Wegner propone uno spostamento del paradigma computazionale che ponga al centro l'idea di *interazione*. Ecco come i curatori di un volume impostato sul concetto di interazione introducono questo nuovo punto di vista.

D. Goldin, S.A. Smolka & P. Wegner, "Interactive Computation: The New Paradigm", Springer-Verlag, 2006.

> Interaction is an emerging paradigm of models of computation that reflects the shift in technology from mainframes to networks of intelligent agents, from number-crunching to embedded systems to graphical user interfaces, and from procedure-oriented to object-based distributed systems. Interaction based models differ from the Turing-machine-based algorithmic models of the 1960s in interesting and useful ways:
>
> *Problem Solving*: Models of interaction capture the notion of performing a task or providing a service, rather than algorithmically producing outputs from inputs.
>
> *Observable Behavior*: In models of interaction, a computing component is modeled not as a functional transformation from input to output, but rather in terms of observable behavior consisting of interaction steps. For example, interactions may consist of interleaved inputs and outputs modeled by dynamic streams; future input values can depend on past output values.
>
> *Environments*: In models of interaction, the world or environment of the computation is part of the model and plays an active part in the computation by dynamically supplying the computational system, or agent, with inputs, and consuming the output values the system produces. The environment cannot be assumed to be static or even effectively computable; for example, it may include humans or other real-world elements.
>
> *Concurrency*: In models of interaction, computation may be concurrent; a computing agent can compute in parallel with its environment and with other agents.
>
> The interaction paradigm provides a new conceptualization of computational phenomena that emphasizes interaction rather than algorithms. Concurrent, distributed, reactive, embedded, component-oriented, agent-oriented and service-oriented systems all exploit interaction as a fundamental paradigm. This book thus challenges traditional answers to fundamental questions relating to problem solving or the scope of computation. It aims to increase the awareness of interaction paradigms among the wider computer-science community and to stimulate practice and theoretical research in interactive computation.

Qui di seguito riporto alcuni estratti di un articolo. È interessante notare come, a giudizio degli autori, l'adozione del nuovo paradigma comporta anche un cambiamento di natura epistemologica.

Dina Goldin & Peter Wegner, "The Interactive Nature of Computing: Refuting the Strong Church-Turing Thesis", Minds and Machines, 18(1), 2008.

> The classical view of computing positions computation as a closed-box transformation of inputs (rational numbers or finite strings) to outputs. According to the interactive view of computing, computation is an ongoing interactive process rather than a function-based transformation of an input to an output. Specifically, communication with the outside world happens during the computation, not before or after it. This approach radically changes our understanding of what is computation and how it is modeled.
>
> [...]
>
> In the computer science community, it is generally understood that [the Curch-Turing Thesis] applies only to effective computation, in Turing's sense of the word. However, it is not always appreciated that [the Curch-Turing Thesis] applies only to computation of functions, rather than to all computation. Function-based computation transforms a finite input into a finite output in a finite amount of time, in a closed-box fashion. By contrast, the general notion of computation includes arbitrary procedures and processes—which may be open, non-terminating, and involving multiple inputs interleaved with outputs.
>
> The Strong Church-Turing Thesis, which asserts that TMs capture all effective computation, is generally considered to be equivalent to the original [the Curch-Turing Thesis ...].
>
> All versions of the [Strong Church-Turing Thesis] historically evolve from the assumption that all computation is function-based, or algorithmic; by this, we mean that its job is to transform a finite input into a finite output in a finite amount of time. We believe it is time to recognize that today's computing applications, such as web services, intelligent agents, operating systems, and graphical user interfaces, are interactive rather than algorithmic; their job is to provide ongoing services over time (Wegner 1997).
>
> According to the interactive view of computation, interaction (communication with the outside world) happens during the computation, not before or after it. Hence, computation is an ongoing process rather than a function-based transformation of an input to an output. The interactive approach represents a paradigm shift that redefines the nature of computer science, by changing our understanding of what computation is and how it is modeled. This view of computation is not modeled by [Turing Machines], which capture only the computation of functions; alternative models are needed.
>
> [...]
>
> The theoretical nature of computing is currently based on what we call the *mathematical worldview.* We discuss this worldview next, contrasting it with the *interactive worldview.* [...]
>
> *Mathematical worldview*: All computable problems are function-based.

> [...]
>
> The mathematical worldview can be contrasted with the interactive worldview, where computation is viewed as an ongoing process that transforms inputs to outputs—e.g., control systems, or operating systems. The question "what do operating systems compute?" has been a conundrum for the adherents of the mathematical worldview, since these systems never terminate, and therefore never formally produce an output. Yet it is clear that they do compute, and that their computation is both useful and important to capture formally.
>
> While the Church-Turing Thesis remains true, the mathematical worldview no longer reflects the nature of computational problems [...].
>
> The interactive approach to conceptualizing the notion of computation and of computable problems is distinct from either the theory of computation and the concurrency theory. It represents a paradigm change to our understanding of what is computation, and how it should be modeled. This conceptualization of computation allows, for example, the entanglement of inputs and outputs, where later inputs to the computation depend on earlier outputs. Such entanglement is impossible in the mathematical worldview, where all inputs precede computation, and all outputs follow it.
>
> [...]
>
> Algorithms originated in mathematics as "recipes" for carrying out function-based computation, that can be followed mechanically. Algorithms capture what it means for a computation to be effective.
>
> [...]
>
> The 1960s decision by theorists and educators to place algorithms at the center of CS was clearly reflected in early undergraduate textbooks. However, there was no explicit standard definition of an algorithm and various textbooks chose to define this term differently.
>
> [...]
>
> The result is a dichotomy, where the computer science community thinks of algorithms as synonymous with the general notion of computation ("what computers do") yet at the same time as being equivalent to Turing Machines. This dichotomy can be found in today's popular textbooks [...]. Their discussion of algorithms is very broad, but the equivalence with [Turing Machines] is taken for granted [...].
>
> This dichotomy is canonized in the Strong Church-Turing Thesis, which can be found throughout the computing literature [...]: "A TM can do anything that a computer can do."
>
> [...]
>
> *The Paradigm Shift to Interaction*
>
> Sequential interaction is only one form of interactive computation. The paper "Interactive Foundations of Computing" by one of the authors (Wegner 1998) explores more general notions of interaction, including analog, real-time, and multiagent interaction. It conjectures that these forms of interaction are more expressive than sequential interaction. It shows that the semantics of streams cannot be expressed by that of strings, that interaction includes nonfunctional non-algorithmic behavior, that persistent agents are not algorithmically describable, that extending algorithms to interaction transforms "dumb" to "smart" problem-solving behaviors. It furthermore shows that increased expressive power may in many cases decrease or eliminate formalizability—the ability to state a problem (or describe a computing system) formally and completely. Dijkstra's "GOTO considered harmful" article (Dijkstra 1968) suggested eliminating the GOTO statement because its expressiveness decreased formalizability. Interaction likewise increases expressiveness at the expense of formalizability, and could therefore be considered harmful in the Dijkstra sense. We believe that expressiveness should be encouraged and that neither interaction nor GOTO statements should be considered harmful because they decrease formalizability.
>
> Interactive systems are incomplete in the sense that they cannot be modeled by sound and complete first-order logics. Incompleteness contributes to expressiveness at the expense of formalizability, supporting non-formalizable error checking, emergent behavior, open systems, object-oriented programming, and robustness. Programming in the large produces non-formal interactive behaviors that are more expressive than algorithmic programming in the small. The paradigm shift from algorithms to interaction suggests that mathematical formalizability must necessarily be restricted in realizing important goals of expressiveness. Interactive models extend formal rationalist systems that limit expressiveness to non-formal empiricist systems that are more expressive (Wegner 1999). Thus rationalist mathematical arguments that formalizability is an essential tool of problem solving must be eliminated in achieving the broader goal of empiricist interactive problem solving.

L'intepretazione didattica di questo approccio, classificabile nell'ambito *object-first*, ma specificamente caratterizzato, è promossa in particolare da Lynn Stein, di cui riporto un estratto della prefazione di un libro.

Lynn Stein, "Interactive Programming in Java", Franklin W. Olin College of Engineering, 2003.

> *Interactive Programming* shifts the foundation on which the teaching of Computer Science is based, treating computation as *interaction* rather than *calculation* [...]. *Interactive Programming* provides an alternate entry into the computer science curriculum. It teaches problem decomposition, program design, construction, and evaluation, beginning with the following premises: A program is a community of interacting entities. Its "pieces" are these implicitly or explicitly concurrent entities: user interfaces, databases, network services, etc. They are combined by virtue of ongoing interactions which are constrained by interfaces and by protocols. A program is evaluated by its adherence to a set of invariants, constraints, and service guarantees—timely response, no memory leaks, etc.

7. L'informatica può considerarsi una scienza nella misura in cui adotta il metodo scientifico

Per Neil Stewart, la chiave della caratterizzazione dell'informatica come scienza sta nel rigore del metodo. Nell'estratto che segue, egli critica alcune debolezze della pratica corrente alla luce delle condizioni formulate da Carl Popper riguardo al metodo scientifico (razionalizzazione, ripetibilità e falsificabilità), ma riconoscendo nei problemi che l'informatica affronta tutte le potenzialità per una pratica scientifica.

Neil F. Stewart, "Science and Computer Science", ACM Computing Surveys, 27(1), 1995.

> It is clear that there are components of computer science that could be viewed as subfields of mathematics or engineering. Indeed, competent work in theoretical computer science meets rigorous mathematical standards, and competent work in applied computer science meets the standards of good-quality engineering work (prototypes are built, proof-of-concept projects are conducted, and results are evaluated on the basis of their usefulness in practice). The open question is the extent to which the remaining parts of computer science can be viewed as science. To study this question is not necessarily to engage a sterile debate: it may be, for example, that the field is not yet a science, but that it could become one, and that certain policy changes could accelerate this process.
>
> Hartmanis argues that computer science is different enough from the other sciences to permit different standards in experiemental work, and that computer science "demos" can be viewed as a replacement for the experimentation found in other fields. I do not agree. Computer science is, or has the potential to be, a science similar in character to physics and the other naturale sciences. However, its traditions, in the area of experimentation and formulation of theories, may delay its acceptance and inhibits its development (as a science).
>
> It is stated [...] that in computer science there are no "... new theories developed to reconcile theory with experimental results that reveal unexplained anomalies or new, unexpected phenomena..." [... However,] there are many areas of computer science where we might propose theories that could be decided on the basis of experiment. For example, what we understand as "intelligence" is very poorly understood. It seems reasonable, therefore, to introduce various theories of intelligence, or models of intelligent behaviour, and to test them using the methods and ethics of modern experimental science.
>
> [...]
>
> One important answer to the question of what constitutes science was given by Popper [...].
>
> The first condition given by Popper is this: even to engage in rational discourse requires that we *state clearly the problem we wish to solve*. [...]
>
> The second requirement, relating to empirical science, is that experiments should be *repeatable*. [...]
>
> Finally, a scientific system must, at least in principle, be *falsifiable*. [...]
>
> For many subfields of computer science, it cannot be said that they meet even Popper's first condition, defining "rational discussion". For example, in computer vision, the problem is frequently left unstated; indeed, there is often no distinction made between the problem to be solved and the algorithm that solves it. [...]
>
> More recently, the introduction of Abstract Data Types can be viewed as a significant step towards putting computer science on a scientific footing, since it directly addresses the question of problem definition. [...]
>
> Popper's second condition is that experiments should be repeatable. Here too computer science is far from the best tradition of scientific work. Reading about the results of putatively scientific experiment in computer science, it is almost certain that there will not be enough information given to permit repetition, and there will therefore be no possibility of refuting the results. [...]

Punto abbastanza interessante: Neil Stewart osserva anche che alcuni problemi affrontati dall'informatica riguardano l'interpretazione della realtà naturale, per esempio quando si occupa dei modelli della cognizione o dell'intelligenza, e in questo senso il celebre test di Turing propone un esperimento propriamente scientifico, che potrebbe essere vagliato sulla base dei principi di Popper. Pertanto, questi problemi appartengono all'ambito scientifico inteso nel senso più puntuale. Inoltre, pur tenendo conto della natura inestricabilmente interdisciplinare di questo tipo di studi, messa in luce da Gordana Dodig-Crnkovic come caratteristica ineluttabile delle scienze moderne, osservazioni analoghe potrebbero estendersi a gan parte dei modelli di simulazione dei fenomeni naturali, sociali o economici, realizzati con mezzi informatici.

In una prospettiva didattica, può essere interessante osservare a questo proposito che l'attività di debugging di un programma, cioè la diagnostica e la correzione degli errori, se svolta in maniera strutturata, esplicitando le ipotesi (teorie in miniatura sulle cause dell'errore), motivando i test (pianificazione di esperimenti per confermare o invalidare le ipotesi) e argomentando le conclusioni che se ne traggono dagli esiti, può essere sfruttata come palestra per esercitare il metodo scientifico.

8. Natura della disciplina e rilevanza educativa

Qualche ulteriore spunto può essere tratto da un articolo di J. Hromkovič, che si è occupato anche di aspetti didattici a livello accademico e nella scuola superiore. Le accezioni matematica, scientifica e ingegneristica, già affrontate da A.H. Eden, sono interpretate con delle sfumature diverse in questo articolo, di cui vengono riportati alcuni estratti qui di seguito. Inoltre, l'autore riflette sul ruolo formativo della disciplina, in relazione alla natura stessa che ancora sfugge a un semplice inquadramento, non solo in quanto palestra di metodo e approccio interdisciplinare, ma anche considerando gli aspetti connessi al linguaggio già sottolineati da G.J. Sussman.

Juraj Hromkovič, "Contributing to General Education by Teaching Informatics", International Conference on Informatics in Secondary Schools – Evolution and Perspectives (ISSEP), 2006.

What Is Informatics?

Let us first attempt to answer the question *"What is computer science?"*

It is difficult to provide an exact and complete definition of a scientific discipline. A commonly accepted definition is: *Computer science is the science of algorithmic processing, representation, storage and transmission of information.*

This definition presents information and algorithm as the main objects investigated in computer science. However, it neglects to properly reveal the nature and methodology of computer science. Another question regarding the substance of computer science is: *"To which scientific discipline does computer science belong? Is it a meta science such as mathematics and philosophy, a natural science or an engineering discipline?"*

An answer to this question serves not only to clarify the objects of the investigation, it also must be determined by the methodology and contributions of computer science. The answer is that computer science cannot be uniquely assigned to any of these disciplines. Computer science includes aspects of mathematics, and natural sciences as well as of engineering [...].

Similar to philosophy and mathematics, computer science investigates general categories such as *determinism, nondeterminism, randomness, information, truth, untruth, complexity, language, proof, knowledge, communication, approximation, algorithm, simulation, etc.* and contributes to the understanding of these categories. Computer science has shed new light on and brought new meaning to many of these categories.

A natural science, in contrast to philosophy and mathematics, studies concrete natural objects and processes, determines the border between possible and impossible and investigates quantitative rules of natural processes. It models, analyzes, and confirms the credibility of hypothesized models through experiments. These aspects are similarly prevalent in computer science. The objects are information and algorithms (programs, computers) and the investigated processes are the physically existing computations. Let us try to document this by looking at the development of computer science. Historically, the first important research question in computer science was the following one with philosophical roots. *"Are there well-defined problems that cannot be automatically (by a computer, regardless of the computational powers of contemporary computers or futuristic ones) solved?"*

Efforts to answer this question led to the founding of computer science as an independent science. The answer to this question is positive. We are now aware of many practical problems that we would like to solve algorithmically, but which are not algorithmically solvable. This conclusion is based on a sound mathematical proof of algorithmic nonsolvability (i.e., on a proof of the nonexistence of algorithms solving the given problem), and not on the fact that no algorithmic solution has been discovered so far.

After developing methods for classifying problems according to their algorithmic solvability, one asks the following scientific question: *"How difficult are concrete algorithmic problems?"*

This [...] difficulty is measured in the amount of work necessary and sufficient to algorithmically compute the solution for a given problem instance. One learns of the existence of hard problems, for which computing solutions needs energy exceeding that of the entire universe. There are algorithmically solvable problems such that the execution of any program solving them would require more time than has passed since the Big Bang. Hence the mere existence of a program for a particular problem is not an indication that this problem is solvable within practical limits. Efforts to classify problems into practically solvable (tractable) and practically insolvable led to the most fascinating scientific discoveries of theoretical computer science.

As an example, let us consider randomized algorithms [...]. In contrast to deterministic programs that reliably deliver the right solution for any input, randomized programs may give erroneous results. The aim is to suppress the probability of such false computations, which under some circumstances means to decrease the proportion of false computations.

At first sight, randomized programs may seem unreliable, as opposed to their deterministic counterparts. Why then the necessity for randomized programs? There are important problems whose solution by the best known deterministic algorithm require more computer work than one can realistically execute. Such problems appear to be practically insolvable. But a miracle can happen: this miracle can be a randomized algorithm that solves the problem within minutes, with a minuscule error probability of one in a trillion. Can one ban such a program as unreliable? A deterministic program that requires a day's computer work is more unreliable than a randomized program running in a few minutes, because the probability that a hardware error occurs during this 24 hours of computation is much higher than the error probability of the fast randomized program.

A concrete example of utmost practical significance is primality testing. In the ubiquitous use of cryptographic public-key protocols, huge prime numbers (approximately 500 digits long) must be generated. The first deterministic algorithms for primality testing were based on testing the divisibility of the input *n*. Alone, the number of primes smaller than [...] such huge values of *n* exceeds the number of protons in the universe. Hence, such deterministic algorithms are practically useless. Recently, a new deterministic algorithm for primality testing running in time $O(m^{12})$ for *n* of binary length *m* was developed. But it needs to execute more than 10^{32} computer instructions in order to test a 500-digit number and so the amount of time since the Big Bang is not sufficient to execute such a computation on the fastest computers. However, there are several randomized algorithms that test primality of such large numbers within minutes or even seconds on a regular PC.

Another spectacular example is a communication protocol for comparison of the contents of two databases, stored in two distant computers [...]. For a database with 10^{16} bits, this would prove to be tedious. A randomized communication protocol can test this equivalence using a message of merely 2000 bits. The error probability of this test is less than one in the number of all protons in the universe.

How is this possible? It is difficult to explain this without some basic knowledge of computer science. The search for the explanation behind the strengths of randomized algorithms is a fascinating research project, going into the deepest fundamentals of mathematics, philosophy, and natural sciences. Nature is our best teacher, and randomness plays a larger role in nature than one would imagine.

Computer scientists can cite many systems where the required characteristics and behaviors of such systems are achievable only through the concept of randomization. In such examples, every deterministic reliable system is made up of billions of subsystems, and these subsystems must interact correctly. Such a complex system, highly dependent on numerous subcomponents, is not practical. In the case that an error occurs, it would be almost impossible to detect it. Needless to say, the costs of developing such a system is also astronomical. On the other hand, one can develop small randomized systems with the required behavior. Because of their small size, such systems are inexpensive and the work of their components is easily verifiable. And the crucial point is that the probability of a wrong behavior is so minuscule that it is negligible.

The presented concept of randomized algorithms is only one among many concepts developed in informatics that influence our view on fundamentals of science at all.

Despite its above-illustrated scientific aspects, computer science is a typical problem-oriented and practical engineering discipline for many scientists. Computer science not only includes the technical aspects of engineering such as: *organization of development processes (phases, milestones, documentation), formulation of strategic goals and limits, modeling, description, specification, quality assurance, tests, integration into existing systems, reuse, and tool support*, it also encompasses the management aspects such as: *team organization and team leadership, costs estimation, planning, productivity, quality management, estimation of time plans and deadlines, product release, contractual obligations, and marketing.*

A computer scientist should also be a true pragmatic practitioner. When constructing complex software (for instance writing programs of several hundred thousand instructions) or hardware systems, one must often make decisions based on one’s experience, because one does not have any opportunity to model and analyze the highly complex reality. All the features of engineering are involved in the design and in the development of final products. To mention at least some of them – modularity in the design processes, testing reliability in a structured way, etc.

Why Teach Computer Science?

Considering our definition of computer science, one may get the impression that the study of computer science is too difficult for secondary school. One needs mathematical knowledge as well as the understanding of the way of thinking in natural sciences, and on topofthat, one needs to be able to work like an engineer. This may really be a strong requirement, but it is also the greatest advantage of this education. The main drawback of current science is in its overspecialization, which leads to an independent development of small subdisciplines. Each branch has developed its own language, often incomprehensible even for researchers in a related field. It has gone so far that the standard way of arguing in one branch is perceived as superficial and inadmissible in another branch. This slows down the development of interdisciplinary research. Computer science is interdisciplinary at heart. It is focused on the search for solutions for problems in all areas of sciences and in everyday life, wherever the use of computers is imaginable. While doing so, it employs a wide spectrum of methods, ranging from precise formal mathematical methods to experience-based “know-how” of engineering. The opportunity to concurrently learn the different languages of different areas and the different ways of thinking, all in one discipline, is the most precious gift conferred on a computer science student.

Teaching informatics in secondary or even primary schools has to start with programming. Programming is more than just a useful skill of a computer scientist. Learning programming means learning a language of communication with technical systems, learning to tell a machine what activity we would like to have from it. Since machines do not have any intelligence, our instructions must be so clearly and unambiguously formulated that no mistake can arise. In this way, the pupils learn to describe ways and methods for achieving aims that can be correctly followed by everybody without needing to provide the knowledge why they successfully achieve these goals. The development of this skill essentially contributes to the pupils’ natural language skills by motivating pupils to properly think about how to best express what they would like to communicate. After supplementing the programming courses with some elementary data structures and algorithms, we propose to switch to the fundamentals.

Why teach the fundamentals? Theoretical computer science is a fascinating scientific discipline. Through its spectacular results and high interdisciplinarity, it has made great contributions to our view of the world. However, theoretical computer science is not the favorite subject of university students, as statistics would confirm. Many students even view theoretical computer science as a hurdle that they have to overcome in order to graduate. There are several reasons for this widespread opinion. One reason is that amongst all areas of computer science, theoretical computer science is the mathematically most demanding part and hence the lectures on theoretical fundamentals belong to the hardest courses in computer science. Not to forget, many computer science students start their study with a wrong impression of computer science, and many lecturers of theoretical computer science do not present their courses in a sufficiently attractive way. Excessive pressure for precise representation of the minute technical details of mathematical proofs plus a lack of motivation, a lack of relevance, a lack of informal development of ideas within the proper framework and a lack of direct implementation and usage, can ruin the image of any fascinating field of science. Is theory really suitable at the secondary school level? Is it so important that we have to invest huge efforts to master its teaching? We try to answer both these questions affirmatively.

[...] There are several important reasons for the indispensability of theoretical fundamentals in the study of computer science [...]:

Philosophical depth
Theoretical computer science explores knowledge and develops new concepts and notions that influence science at its very core. Theoretical computer science gives partial or complete answers to philosophical questions [...]. It is important to note that many of these questions cannot be properly formulated without the formal concepts of algorithm and computation. Thus, theoretical computer science has enriched the language of science through these new terms, contributing to its development. Many known basic categories of science, such as determinism, chance, and nondeterminism have gained new meanings, and through this, our general view of the world has been influenced.

Applicability and spectacular results
Theoretical computer science is relevant to practice. On one hand, it provides methodological insights that influence our first strategic decision over the processing of algorithmic problems. On the other hand, it provides particular concepts and methods that can be applied during the whole process of design and implementation. Moreover, without the knowledge and concepts of theoretical computer science many applications would be impossible. [...] This not only shows that, thanks to theory, things are made possible though previously they were believed to be impossible, it also shows that research in theoretical computer science is exciting and full of surprises, and so one can be inspired and enthused by theoretical computer science.

Lifespan of knowledge
Through the rapid development of technology, the world of applied computer science continuously evolves. Half of the existing information about software and hardware products is obsolete after 5 years. Hence, an education that is disproportionately devoted to system information and current technologies, does not provide appropriate job prospects. Whereas the concepts and methodology in theoretical computer science have a longer average lifespan of several decades. Such knowledge will serve its owner well for a long period of time.

Interdisciplinary orientation
Theoretical computer science is interdisciplinary in its own right and can take part in many exciting frontiers of research and development — genome projects, medical diagnostics, optimization in all areas of economy and technical sciences, automatic speech recognition, and space exploration, just to name a few. As much as computer science contributes to all other fields, it also benefits from the contributions from other fields. The study of computations on the level of elementary particles, [e.g.,] whose behavior follows the rules of quantum mechanics, focuses on the efficient execution of computations in the microworld whose execution in the macroworld has failed.

Way of thinking
Mathematicians attribute the special role mathematics play in education through development, enrichment and shaping the way of thinking, i.e., through contributing to the general development of one's personality. If this contribution by mathematics is so highly regarded, then one must also acknowledge the importance of computer science for the general education and the enrichment of the way of thinking. Theoretical computer science encourages creating and analyzing mathematical models of real systems and searching for concepts and methods to solve concrete problems. Remember that precisely understanding which features of a real system are exactly captured by one's model and which characteristics are only approximated or even neglected is the main assumption for a success in science and engineering. Because of this, theoretical computer science calls attention to teaching the evolution of mathematical concepts and models in a strong relation to real problems. Thus, by studying computer science, one has the chance of learning how to combine theoretical knowledge with practical experience and hence develop a way of thinking that is powerful enough to attack complex real-world problems. [...]

What to Teach and How to Teach It?

[...] Computer science courses have to start with programming. If we want to build a language for instructing (communicating with) a machine that does not have any intelligence, and hence any ability to improvise, we have to be very careful. We have to teach programming as a skill to describe possibly complex behaviors by a sequence of clear, simple instructions [...]. One has to start with a very small number of computer instructions available and build new, more powerful instructions by combining the simple instructions available. In this way one not only follows the historical development, one also learns the principle of modularity that is fundamental for all engineering sciences. To be in context with other subjects, the choice of algorithmic problems in the programming course has to include tasks encountered in mathematics, physics, and possibly other subjects.

The programming course can be followed by the introduction of some basic concepts of data structures and [...] algorithms [...]. Teaching fundamentals could start with automata theory. The reason is that finite automata provide the simplest model of computation, and hence one can learn to understand to some extent, the meaning of the fundamental notions such as computation, simulation, configuration [...].

The last part of our courses of informatics in secondary schools is devoted to computability. Many peers are of the opinion that this is too hard at preuniversity level. Everything is a matter of didactic mastery. Our experience is that this topic is considered to be the most fascinating one by the pupils. They visit the core of sciences by learning what infinity is, that there are infinities of different sizes, and finally that there are interesting algorithmic problems that cannot be solved automatically (algorithmically). The pupils are fascinated because we enable them to track the discovery of the fundamentals of informatics and to even walk this path step by step with full understanding of the nature of the discovery processes.

[...]

We would like to build and influence the student’s ways of thinking. We are interested in the historical development of computer science concepts and ways of thinking, and the presentation of definitions, results, proofs, and methods is only a means to the end. Hence, we are not overly concerned about the amount of information, preferring to sacrifice 10 to 20% of the teaching material. In return, we dedicate more time to the motivation, aims, connection between practice and theoretical concepts, and especially to the internal context of the presented theory. We place special emphasis on the creation of new terms. The notions and definitions do not appear out of the blue, as seemingly so in some lectures using the formal language of mathematics. The formally defined terms are always an approximation or an abstraction of intuitive ideas. The formalization of these ideas enables us to make accurate statements and conclusions about certain objects and events. They also allow for formal and direct argumentation. We strive to explain our choice of the formalization of terms and models used and to point out the limitations of their usage. Learning to work on the level of terms creation (basic definitions) is very important, because most of the essential progress happens exactly on this level.

9. Gli algoritmi sono il cuore della disciplina, di cui attraversano e unificano le varie ramificazioni

Un contributo di interesse anche storico, introdotto da un preambolo spiritoso nello stile dell’autore, da parte di uno dei padri dell’informatica. In particolare, si può osservare che questo punto di vista si contrappone la posizione più recente di Wegner e colleghi menzionata sopra.

Donald E. Knuth, “Computer Science and Its Relation to Mathematics”, The American Mathematical Monthly, 81(4), 1974.

What is Computer Science?

Since Computer Science is relatively new, I must begin by explaining what it is all about. At least, my wife tells me that she has to explain it whenever anyone asks her what I do, and I suppose most people today have a somewhat different perception of the field than mine. In fact, no two computer scientists will probably give the same definition; this is not surprising, since it is just as hard to find two mathematicians who give the same definition of Mathematics. Fortunately it has been fashionable in recent years to have an “identity crisis,” so computer scientists have been right in style.

My favorite way to describe computer science is to say that it is the study of *algorithms*. An algorithm is a precisely-defined sequence of rules telling how to produce specified output information from given input information in a finite number of steps. A particular representation of an algorithm is called a program, just as we use the word “data” to stand for a particular representation of “information”. Perhaps the most significant discovery generated by the advent of computers will turn out to be that algorithms, as objects of study, are extraordinarily rich in interesting properties; and furthermore, that an algorithmic point of view is a useful way to organize knowledge in general. G.E. Forsythe (1968) has observed that “the question ‘What can be automated?’ is one of the most inspiring philosophical and practical questions of contemporary civilization”.

From these remarks we might conclude that Computer Science should have existed long before the advent of computers. In a sense, it did; the subject is deeply rooted in history [...].

But computers are really necessary before we can learn much about the general properties of algorithms; human beings are not precise enough nor fast enough to carry out any but the simplest procedures. Therefore the potential richness of algorithmic studies was not fully realized until general-purpose computing machines became available.

When I say that computer science is the study of algorithms, I am singling out only one of the “phenomena surrounding computers” (Newell et al., 1967), so computer science actually includes more. I have emphasized algorithms because they are really the central core of the subject, the common denominator which underlies and unifies the different branches.

[...]

Is Computer Science Part of Mathematics?

[… If] we restrict our study to algorithms, isn’t this merely a branch of mathematics? [...]

The difference is in the subject matter and approach --- mathematics dealing more or less with theorems, infinite processes, static relationships, and computer science dealing more or less with algorithms, finitary constructions, dynamic relationships.

[...]

Educational side-effects.

It has been often said that a person does not really understand something until he teaches it to someone else. Actually a person does not *really* understand something until he can teach it to a *computer*, i.e., express it as an algorithm. “The automatic computer really *forces* that precision of thinking which is alleged to be a product of any study of mathematics” (Forsythe, 1968). [... Several examples] have convinced me of the pedagogic value of an algorithmic approach; it aids in the understanding of concepts of all kinds.

10. Un informatico ha il compito di realizzare strumenti per soddisfare le esigenze di chi dovrà usarli

Nel seguente estratto viene ben delineata una visione in un certo senso estrema, prettamente ingegneristica, dell’informatica. Secondo questa prospettiva perdono rilievo le motivazioni rivolte alla crescita delle conoscenze come fine di per se stessa, motivazioni che sono tipiche degli ambiti scientifici e matematici.

Frederick P. Brooks, Jr., “The computer scientist as toolsmith II”, Communications of the ACM, 39(3), 1996.

> [The] scientist builds in order to study; the engineer studies in order to build. I submit that by any reasonable criterion the discipline we call “computer science” is in fact not a science but a synthetic, an engineering, discipline. We are concerned with making things, be they computers, algorithms, or software systems. If we perceive our role aright, we then see more clearly the proper criterion for success: a toolmaker succeeds as, and only as, the users of his tool succeed with his aid.
>
> [...]
>
> [As] we honor the more mathematical, abstract, and “scientific” parts of our subject more, and the practical parts less, we misdirect young and brilliant minds away from a body of challenging and important problems that are our peculiar domain, depriving these problems of the powerful attacks they deserve.
>
> [...]
>
> Especially important for us are system design problems characterized by arbitrary complexity. [...] The arbitrariness is inherent — the requirements and constraints spring from a host of independent minds. [...] These problems scandalize and discourage those who approach them from backgrounds of mathematics and natural science, and for different reasons. Mathematicians are scandalized by the complexity — they like problems which can be simply formulated and readily abstracted, however difficult the solution. The four-color problem is a perfect example. Physicists or biologists, on the other hand, are scandalized by the arbitrariness. Complexity is no stranger to them. The deeper the physicists dig, the more subtle and complex the structure of the “elementary” particles they find. But they keep digging, in full faith that the natural world is not arbitrary, that there is a unified and consistent underlying law if they can but find it. No such assurance comforts the computer scientist. Arbitrary complexity is our lot, and here more than anywhere else we need the best minds of our discipline fashioning more powerful attacks on such problems.
>
> [...]
>
> It is time to recognize that the original goals of AI were not merely extremely difficult, they were goals that, although glamorous and motivating, ‘sent the discipline off in the wrong direction’. [... A] machine and a mind can beat a mind-imitating machine working by itself.

11. Paradigma computazionale

Si riportano, infine, alcuni passi di un articolo molto citato in questi anni, specialmente in relazione alla didattica dell'informatica, che porpone di orientarsi verso una visione più ampia e "culturale" della disciplina. Alla luce di questa nuova prospettiva, sono stati proposti curricula di introduzione all'informatica concepiti sulla base di criteri diversi da quelli tradizionali.

Jeannette M. Wing, "Computational Thinking", Communications of the ACM, 49(3), 2006.

> Computational Thinking [...] represents a universally applicable attitude and skill set everyone, not just computer scientists, would be eager to learn and use.
>
> [...]
>
> Thinking like a computer scientist means more than being able to program a computer. It requires thinking at multiple levels of abstraction.
>
> [...]
>
> Computational thinking is thinking recursively. It is parallel processing. It is interpreting code as data and data as code. It is type checking as the generalization of dimensional analysis. [...] It is judging a program not just for correctness and efficiency but for aesthetics, and a system's design for simplicity and elegance. [...] It is using invariants to describe a system's behavior succinctly and declaratively. It is having the confidence we can safely use, modify, and influence a large complex system without understanding its every detail. [...] Computational thinking is thinking in terms of prevention, protection, and recovery from worst-case scenarios through redundancy, damage containment, and error correction.
>
> [...]
>
> We have witnessed the influence of computational thinking on other disciplines. [... E.g.,] Computer science's contribution to biology goes beyond the ability to search through vast amounts of sequence data looking for patterns. The hope is that data structures and algorithms --- our computational abstractions and methods --- can represent the structure of proteins in ways that elucidate their function. Computational biology is changing the way biologists think.
>
> [...]
>
> – *Conceptualizing, not programming*
>
>   Thinking like a computer scientist [...] requires thinking at multiple levels of abstraction […].
>
> – *Fundamental, not rote skill*
>
>   A fundamental skill is something every human being must know to function in modern society […].
>
> – *A way that humans, not computers, think*
>
>   Computational thinking is a way humans solve problems, it is not trying to get humans to think like computers […].
>
> – *Complements and combines mathematical and engineering thinking* […].
>
> – *Ideas, not artifacts* […].
>
> – *For everyone, everywhere*
>
>   Computational thinking will be a reality when it is so integral to human endeavors it disappears as an explicit philosophy […]. We especially need to reach the pre-college audience, including teachers, parents, and students, sending them [a main message]:
>
> – *Intellectually challenging and engaging scientific problems remain to be understood and solved* — The problem domain and solution domain are limited only by our own curiosity and creativity.
>
> [...]
>
> Professors of computer science should teach a course called "Ways to Think Like a Computer Scientist".

9. Riepilogo

Proviamo a riepilogare i principali punti di vista emersi dagli estratti riportati sopra:

1. G. J. Sussman

l'informatica pertiene all'ambito linguistico?

L'informatica, più che una scienza, è un *linguaggio* che ci permette di esprimere rigorosamente un nuovo tipo di conoscenza (epistemologia procedurale) che prima poteva solo essere indotta da esempi.

In realtà, la stessa osservazione si potrebbe fare riguardo alla matematica. Di fatto, la conoscenza che sottende un nuovo linguaggio diventa oggetto di studio in se stessa (vedi la logica, ma non solo).

2. R. Bornat

l'informatica pertiene all'ambito matematico-logico?

L'approccio scientifico sta nella motivazione del ricercatore, se questo mira a conoscere in senso idealistico (la verità), indipendentemente dall'utilità, piuttosto che a risolvere problemi pratici, come invece fa l'ingegnere.

A margine, l'evocazione della visione Platonica ci ricorda che solo se crediamo nell'esistenza ontologica della realtà studiata dalla matematica o dall'informatica, allora queste discipline possono essere intese come scienze della realtà.

3. H. Eden

l'informatica pertiene all'ambito scientifico?

L'*argomento della complessità* degli attuali sistemi software rende poco realistico fondare il processo di validazione dei programmi esclusivamente sul metodo analitico-deduttivo del paradigma razionalista.

Inoltre, le caratteristiche di *non-linearità* e *auto-modificabilità* di questi sistemi costituiscono ulteriori elementi a sostegno della sperimentazione scientifica.

4. G. Dodig-Crnkovic

i paradigmi del passato riescono a caratterizzare la natura dell'informatica?

I paradigmi disciplinari che hanno guidato l'inquadramento epistemologico delle scienze classiche non sembrano più adeguati a rappresentare l'evoluzione di tutte le discipline scientifiche moderne, in termini di complessità, pluridisciplinarità e indistricabile relazione con le tecnologie.

Emerge la ricchezza di stimoli offerti dalla disciplina.

5. T. Colburn

l'astrazione ha la stessa natura nell'informatica e nella matematica?

La dicotomia astratto/concreto (algoritmo/macchina), così come si manifesta nell'informatica, ha implicazioni filosofiche di rilievo, riminescenti del dualismo mente/corpo, che comprendono il piano ontologico e hanno conseguenze di interesse pratico, per esempio giuridiche.

Le capacità di astrazione sono certamente fra le più importanti nella formazione informatica, sia a fini professionali che di ricerca.

6. P. Wegner

Il concetto classico di algoritmo è ancora centrale per l'informatica?

I modelli funzionali, caratteristici delle teorie matematiche della calcolabilità, non sono più sufficientemente espressivi per descrivere la maggior parte delle computazioni che hanno luogo nei moderni sistemi. Ciò rende necessario un cambiamento di paradigma che assuma le interazioni come entità centrali.

I nuovi modelli si prestano meno a un trattamento analitico, e implicano anche un cambiamento di natura epistemologica: da un approccio fondato sul razionalismo (matematico) a uno basato piuttosto sull'empirismo (scientifico?).

7. N. F. Stewart

l'informatica applica il metodo scientifico per vagliare i risultati?

L'approccio scientifico sta nelle caratteristiche e nel rigore del metodo attraverso cui i risultati sono vagliati a beneficio della comunità dei ricercatori.

Alcuni dei problemi affrontati in ambito informatico si propongono di interpretare la realtà naturale, e in questo senso possono essere affrontati con il metodo delle scienze empiriche.

8. J. Hromkovič

La natura dell'informatica ha rilievo per l'educazione secondaria?

L'informatica sfugge a una classificazione secondo le categorie disciplinari tradizionali: comprende aspetti che la caratterizzano come meta-scienza, alla stregua della matematica o della filosofia, come scienza della natura e, allo stesso tempo, come disciplina ingegneristica.

L'informatica ha un importante valore formativo perché è interdisciplinare alla radice e dà l'opportunità di apprendere i linguaggi di diverse aree, nonché modi di pensare diversi, tutto all'interno di una stessa disciplina.

9. D.E. Knuth *La centralità dell'algoritmo fa dell'informatica un ramo della matematica?*	Al cuore dell'informatica si colloca l'algoritmo, concetto che attraversa e unifica I diversi ambiti della disciplina e che assume un rilievo particolare anche dal punto di vista pedagogico. mentre la matematica si occupa prevalentemente di relazioni, intese staticamente, l'informatica affronta gli aspetti dinamici dei processi.
10. F.P. Brooks, Jr. *Le ragioni dell'informatica sono in ultima istanza le ragioni degli utilizzatori?*	L'informatica non è una scienza, bensì una disciplina sintetica, ingegneristica, il cui successo dipende dalla soddisfazione degli utilizzatori degli strumenti sviluppati. L'approccio ai problemi dell'informatica è caratterizzato da complessità ed arbitrarietà (che lascia spazio alla creatività), distinguendosi decisamente sia da quello matematico, che tende ad evitare astrazioni complesse, sia da quello scientifico, che si aspetta che le leggi della natura non siano arbitrarie.
11. J.M. Wing *Quali sono i modi di pensare di un informatico?*	Pensare come un informatico significa essere in grado di destreggiarsi fra una molteplicità di livelli di astrazione. È opportuno de-enfatizzare gli aspetti più tecnici, come la pratica della programmazione, al fine di riuscire ad apprezzare il peculiare contributo che la disciplina può offrire per affrontare problemi intelletualmente stimolanti, anche in connessione con altri ambiti della conoscenza.

Come si può osservare, la risposta a domande apparentemente contraddittorie può essere data in senso positivo o negativo, a seconda di quali aspetti si sta privilegiando. L'informatica può essere l'insieme di tutte queste risposte "positive", oppure può essere qualcosa di diverso da ciò a cui eravamo abituati, ma per ora la prospettiva dipende dalle inclinazioni del soggetto che vi riflette e non ci si può aspettare una definizione condivisa da tutti i membri della comunità che praticano l'informatica.

10. Ulteriori riferimenti

T. Colburn, "Philosophy and Computer Science", Sharpe, 2000.

L. Floridi (Ed.), "The Blackwell Guide to the Philosophy of Computing and Information", Blackwell, 2003.