# Algorithms and Their Explanations

Benini, M.; Gobbo, F.

*Published in:*
Lecture Notes in Computer Science

*DOI:*
10.1007/978-3-319-08019-2_4

# Algorithms and Their Explanations

Marco Benini[1] and Federico Gobbo[2]

[1] Università degli Studi dell'Insubria, via Mazzini 5, 21100 Varese, Italy,
marco.benini@uninsubria.it
[2] Universiteit van Amsterdam, Spuistraat 210, 1012 VT Amsterdam,
The Netherlands, F.Gobbo@uva.nl

**Abstract.** By analysing the explanation of the classical heapsort algorithm via the method of levels of abstraction mainly due to Floridi, we give a concrete and precise example of how to deal with algorithmic knowledge. To do so, we introduce a concept already implicit in the method, the 'gradient of explanations'. Analogously to the gradient of abstractions, a gradient of explanations is a sequence of discrete levels of explanation each one refining the previous, varying formalisation, and thus providing progressive evidence for hidden information. Because of this sequential and coherent uncovering of the information that explains a level of abstraction—the heapsort algorithm in our guiding example—the notion of gradient of explanations allows to precisely classify purposes in writing software according to the informal criterion of 'depth', and to give a precise meaning to the notion of 'concreteness'.

## 1 Introduction

We are currently living in the age of the zettabyte ($10^{21}$ bytes), a quantity of information "expected to grow fourfold approximately every three years... every day, enough new data is being generated to fill all US libraries eight times over" Floridi [9], page 5. This quantity of information is mostly produced through digital computers, and therefore it is algorithmic in nature, at least in part. Even from a syntactic point of view, algorithmic information is of a very different character than ordinary information: while the latter relies on the classic theory by Shannon and Weaver, the fundamentals of algorithmic information are in the theory of computation as initiated by Turing—see Chapter 14 in Allo *et al* [2].

Furthermore, we need semantics to upgrade information (the agent in the state of being informed) to knowledge (the agent in the state of being able to perform a conscious informational analysis). Chapter 10 of Floridi [8] solves this problem by giving two different logics at the basis of the states above (see also Allo [3, 1]) while Primiero [15] analyses the special case of information locally valid, i.e., when functional information is in charge. Functional information, commonly used in information sciences, particularly in software engineering, entails realisable instructions to obtain reliable data not yet available. The aim of this paper is to analyse knowledge, in the sense above, in the case of algorithms inside this line of research known as Philosophy of Information, see Allo *et al* [2].

To do so, we will analyse the heapsort algorithm. The heapsort algorithm has been chosen as the guiding example for two reasons: in the first place, heapsort is a classical algorithm, deeply studied and used, and non elementary; in the second place, heapsort exhibits in a nutshell all the features that appears in larger and more complex software, so it provides an ideal case study to test and to explain ideas about the epistemology of computing and programming.

The paper is organised as follows: in the next section, the terminology and the basic concepts of the method of levels of abstraction are introduced, tailored to our purposes. Section 3 is devoted to illustrate the heapsort algorithm from three different points of view: the ones of a programmer, of a software designer, and of an algorithm designer. Section 4 introduces the notion of gradient of explanations, showing how the analysis conducted in Sect. 3 generates one, and some consequences are drawn. The paper concludes with a brief summary and discussion of possible future developments.

## 2 The Method of Levels of Abstraction

The method of levels of abstraction comes from modelling, a common practice in science: in its standard presentation, variables model observations of reality, where only necessary details are retained. The method is flexible, as it can be used in qualitative terms without technicalities, as in Floridi [9], where ethical issues are analysed, as well as in the advanced educational settings presented in Gobbo and Benini [10]. Oppositely, the method can be used in a technical sense, as for instance in the case of algorithmic information analysed in this paper.

In fact, algorithmic information presupposes that the informational organisms in charge are computational in nature. In other words, computational informational organisms ($c$-inforgs), are formed by (at least) a human being and by some kind of computing machine—typically a modern digital computer. As Gobbo and Benini [13] argued information can be hidden to the eyes of the observers according to the growth of complexity of the $c$-inforg itself, even if it can be revealed if the agent holds the necessary knowledge to cope with the complexity at the given level of abstraction. In fact, the key feature of a $c$-inforg is being programmable, and not every variable in the given level of abstraction is granted to be completely observable – instead it could be hidden, exactly because of the nature of algorithmic information.

The method distinguishes three kinds of levels: proper *Levels of Abstraction* (LoAs); the *Levels of Organisation* (LoO), the machinery part of the $c$-inforg, and *Levels of Explanation* (LoE). In general, the LoAs and the LoOs are always strictly connected in every kind of informational organisms. In particular, in the case of $c$-inforgs, this connection is particularly clear. In fact, each software abstraction (LoA) is run over a correspondent hardware abstraction (LoO): the history of modern computing shows a continuous drift from hardware to software; in our terms, more LoAs are introduced so to abstract over the hardware, see Gobbo and Benini [12] for details. Moreover, for each pair of LoA/LoO it is possible to identify more than one LoE because $c$-inforgs are programmable,

and at least the programmer's and end-user's views are possible for the same pair LoA/LoO, see Gobbo and Benini [11].

If a range of LoAs is made of discrete variables and each level can be nested into another within a sequence, that range is called nested *Gradient of Abstraction* (GoA)—see subsection 3.2.6 in Floridi [8]. As we will see in the next section, the explanation of algorithms needs a new concept which is implicit in the method presented until now, that is a *Gradient of Explanation* (GoE), which holds if a GoA is in charge. We will justify the epistemic need of GoE inductively, via the examination of heapsort algorithm.

## 3 The Heapsort Algorithm

The heapsort procedure is a classical topic in the study of algorithms, see, e.g., pages 144–148 of Knuth [14]. In the following, the presentation is mainly based on Chapter 6 of Cormen *et al* [5], although we adopt the method of levels of abstraction to make explicit the various hypotheses and building passages, to let the non-technical reader follow our arguments.

Heapsort solves the problem of sorting an array: given an array $A$ with homogeneous elements and a total ordering $\preceq$ among its elements, the problem asks to construct an array $B$ whose elements are ordered by $\preceq$, i.e., $B[1] \preceq B[2] \preceq B[3] \preceq \cdots$, and such that $B$ is a permutation of $A$.

The reason why heapsort is an interesting solution to the sorting problem is its efficiency: the resulting array $B$ is constructed out of $A$ using only a small and constant amount of additional memory, in contrast with mergesort for example, and the computing process takes a number of steps proportional to $n \log n$, with $n$ the number of elements in $A$. The time complexity is optimal, because no comparison-based solution to the sorting problem may be computed in a number of steps whose order of magnitude is less than $n \log n$ in the worst case.

### 3.1 A programmer's view

When a programmer is asked to implement heapsort, he should consider to be part of a $c$-inforg $P$. The LoO $P_O$ and the LoA $P_A$ describing the computing device in $P$ are known to the programmer. For the sake of clarity, let us assume the LoA $P_A$ to be the bare programming language C and the LoO $P_O$ to be the computer memory as seen through the primitives and libraries of the language. The purpose of the programmer is to construct another LoA $S_A$ on the top of $P_A$, providing a new operation, the sorting algorithm, which becomes observable in the corresponding LoO $S_O$ by the Heapsort syntax.

To perform the implementation, the programmer needs a complex and structured amount of knowledge. First, he knows the syntax of the programming language that will be used to implement heapsort—C, in our illustration; also, he knows how the various instructions and constructions of the language modify the state of the machine, the so-called operational semantics of the language. These pieces of knowledge come from being a programmer. Then, he needs to know the

```
Heapsort(A : array) ≡
    BuildMaxHeap(A)
    for i ← len(A) downto 2 do
        exchange(A[1], A[i])
        heapsize(A) ← heapsize(A) − 1
        MaxHeapify(A, 1)

BuildMaxHeap(A : array) ≡
    heapsize(A) ← len(A)
    for i ← ⌊len(A)/2⌋ downto 1 do
        MaxHeapify(A, i)

left(i: ℕ) ≡ 2i

right(i: ℕ) ≡ 2i + 1

MaxHeapify(A: array, i: ℕ) ≡
    m ← i
    l ← left(i)
    r ← right(i)
    if l ≤ heapsize(A) & A[l] ≻ A[m] then
      m ← l
    if r ≤ heapsize(A) & A[r] ≻ A[m] then
      m ← r
    if m ≠ i then
      exchange(A[i], A[m])
      MaxHeapify(A, m)

exchange(A[i]: element, A[j]: element) ≡
    x ← A[i];
    A[i] ← A[j];
    A[j] ← x
```

**Fig. 1.** The heapsort algorithm

description of the heapsort algorithm in some (semi-)formal notation which is the *specification* of his task. For example, we may assume the programmer knows the pseudo-code in Fig. 1. Of course, he needs to understand the notation: specifically, he has to know that $\leftarrow$ means assignment; that **for** $x \leftarrow e$ **downto** $n$ **do** $B$ means a loop; that indentation is used for grouping instructions; and so on.

This knowledge is not yet enough. For example, the presented pseudo-code assumes the array $A$ to be a data structure having two operations: $\text{len}(A)$ which tells the number of elements in the array $A$, and $A[i]$ which gives access to the $i$-th element of $A$, provided $1 \leq i \leq \text{len}(A)$. A careful inspection of Fig. 1 reveals that the pseudo-code assumes that the ordering relation $\succ$ is embedded into the algorithm rather than a parameter.

It is clear that the amount of information described so far is enough to allow the programmer to fulfil his implementation task. Thus, this amount of information forms an *explanation* of heapsort, the one allowing to implement the algorithm inside the $c$-inforg $P$, producing a new pair $(S_A, S_0)$ of LoA and LoO, respectively. This new abstraction allows the programmer to use the computing device in $P$ in a new way, because a new concept is available, heapsort.

The new LoA $S_A$ and LoO $S_O$ are explained by a corresponding LoE $S_{E;P}$ which can be stated in natural language as follows: 'the LoE is all that is needed to fulfill the purpose of encoding in C language the heapsort algorithm'.

The amount of knowledge necessary to fulfil the purpose is the one sketched above, used by the programmer to implement $S_A$ and $S_O$ from $P_A$ and $P_O$. As we have seen, this kind of knowledge can be adequately represented in what Primiero [15] calls functional information. Also, it should be clear that almost no creativity is involved here, and so at this rather low level of abstraction programmers are not exploiting the artistic possibilities inside the act of programming – see Gobbo and Benini [11].

### 3.2 A software designer's view

The point of view of a prototypical software designer is a step beyond the basic programmer's: in our terms, the LoE $S_{E;P}$ is *nested* in another, broader LoE. In fact, the software designer has to provide a specification to the programmer so that the implementation shall be coherent with the needs of a software which contains many other algorithms. Thus, the designer is aware of the sorting problem and recognises the problem as a node in a larger network of problems, whose overall solution forms the software where the heapsort implementation is only a small component.

Hence, the software designer is part of the $c$-inforg $P$ together with the programmer, and they share the same LoA $P_A$ and LoO $P_O$. The programmer and the software designer are similar to the carpenter whose LoE is to make a chair (example in Floridi [8], section 3.4): the functional organisation (the blueprint) and the realisation (the chair) are similar to the design of the algorithm and its actual implementation in our example. The 'only' difference between making the chair and the design and development of heapsort is the need for two agents with different degrees of knowledge and specialisation. In fact, the designer and the programmer have distinct LoEs, because their purposes are very different. Precisely, the designer has to choose an algorithm which solves the problem he is examining, the sorting problem in our example. The choice is guided by many issues: how frequently the problem has to be solved inside the complete application; the size of the array to be sorted; how the array data structure has to be organised. All these issues, and maybe others, shape the particular instance of the 'right' algorithm to choose. For example, the designer may choose heapsort because he knows it is efficient—a feature which is relevant when arrays may be large or it is not possible to predict their size in advance—and because *stability*, the relative order of equal elements with respect to the ordering should not necessarily be preserved by the sorting process. Also, he may choose heapsort instead of mergesort or quicksort because a destructive manipulation of the array is acceptable, and because a good performance in the worst case is preferred to a better performance in the average case, which is the difference between heapsort and quicksort, see Cormen *et al* [5].

But heapsort is an algorithm, i.e., an abstract, ideal computation, and before becoming a valid specification, a number of decisions have to be taken. For example, in Fig. 1 it is clear that the length of an array $A$ is considered an attribute of $A$; alternatively, the length could have been passed to the heapsort procedure as an additional parameter. Moreover, although it is clear that the heap data structure is a sub-type of array, having to obey an additional constraint, and this fact is rendered by adding the additional attribute heapsize to heaps only, there is no explicit 'recasting' of types in the specification, which means that this piece of information in not required to be made explicit, e.g., to the language type-checker. Furthermore, the ordering relation $\succ$, see Fig. 1, is not structurally linked to the array data structure, suggesting that $\succ$ is not a parameter of the heapsort procedure—while it may be, if different orderings are required in distinct contexts of the same program. Finally, the decision to ab-

stract over exchanges of elements and the identification of left and right branches in the heap suggests that the designer may want to leave space for changing the heapsort procedure in future maintenance releases of the program.

All the above descriptions of the possible reasons behind the shape of the presented pseudo-code, and possibly others, form the LoE $S_{E;D}$ which interprets the LoA $S_A$ and the LoO $S_O$. According to the designer's perspective, the programmer's LoE could be fulfilled by other pairs LoA/LoO.

### 3.3 An algorithm designer's view

Given a problem, usually arising from a concrete application, an algorithm designer is faced with the task to conceive a computable method to calculate its solutions. Often, software designers pick algorithms off the shelf, using and adapting the vast literature Computer Science is continuing to produce. But, ideally, an algorithm designer is another human agent in the same $c$-inforg $P$ we previously introduced: in fact, each instance of an algorithm has been conceived for a specific computational architecture which has to be shared among the algorithm designer, the software designer and the programmer to be effectively implemented in the context (program) where it is intended to be used—recall the parallel with the carpenter and the chair.

The first step in designing an algorithm is to polish the problem, to abstract over what is not needed to solve it, and to eventually reveal the inner structure which will be used to calculate the solution. This abstraction process is guided by ingenuity and a well-established set of techniques (again Cormen *et al* [5] for a comprehensive introduction). We will follow our example of heapsort to illustrate the design process of an algorithm. At the first stage, the elements in the array $A$ can be considered to be composed by two distinct parts: the *key* and the *datum*. Ordering considers just the key, so this is the only relevant part with respect to the sorting problem. It is not important that elements really have two distinct parts: as far as it is possible to extract the key value from an element, the abstraction is fair, and this possibility amounts to have an effective, computable predicate $\preceq$, representing the ordering relation. It is worth noticing that the above abstraction leaves a trace in the properties of a sorting algorithm: it is exactly the separation between the key and the datum that identifies the stability of an algorithm. In fact, if in the input array $A$, $A[i] \preceq A[j]$ and the key of the $i$-th element equals the key of the $j$-th element, stability says that in the sorted array $B$, $A[i] \equiv B[k] \preceq B[h] \equiv A[j]$ exactly when $i \leq j$. Thus, elements which are equal with respect to $\preceq$, may be different when considered as elements, revealing how they have been rearranged. In general, the fact that equality may not be identity in Computer Science has a number of consequences, discussed in Gobbo and Benini [10].

Since elements can be identified with their keys when solving the sorting problem, the idea behind heapsort is to define a sub-class of arrays, the *heaps*, whose members have a distinctive property of interest. Posing $\text{parent}(i) = \lfloor i/2 \rfloor$, $\text{left}(i) = 2i$ and $\text{right}(i) = 2i + 1$, we can interpret the elements in an array as

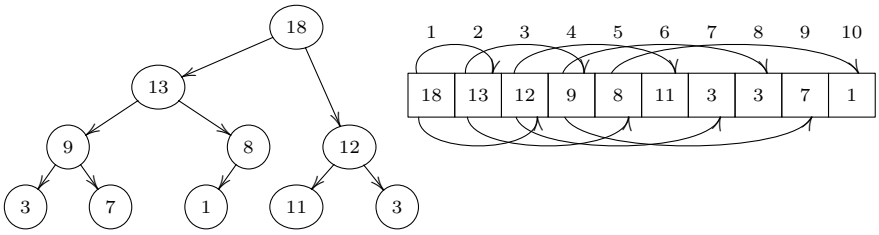


**Fig. 2.** A heap represented as a tree and as an array

if they were nodes in a tree, as illustrated in Fig. 2. Now, a heap $H$ is an array satisfying, for each valid index $i > 1$:

$$H[i] \preceq H[\text{parent}(i)] \ . \tag{1}$$

By abusing terminology, an array $A$ will be called a heap when its initial segment running from index 1 to a known index heapsize($A$) satisfies (1). Hence, the root node $H[1]$ in a heap $H$ is the greatest element in $H$, see Fig. 2, and moreover, its left and right sub-trees are heaps, too. Thus, we can sort the array by moving the root past the end of the heap, and then we can combine the left and the right sub-trees to obtain a new heap. In fact, this is exactly how heapsort operates, see the Heapsort procedure in Fig. 1.

Therefore, the algorithm designer is left with two sub-problems: building an initial heap out of a given array, and constructing a heap given two heaps. The former problem can be easily reduced to the latter: given an array $A$, its elements beyond $\lfloor \text{len}(A)/2 \rfloor$ are leaves in the tree representation, so they are trivially heaps. Each node $i$ which is a parent of some other node, must satisfy (1): it is immediate to see that exchanging the parent node with the greatest node among $A[i]$, $A[\text{left}(i)]$, and $A[\text{right}(i)]$ forces (1) to hold, except for the sub-tree whose root has been changed—something which can be recursively restored. Thus, recursively applying this process, coded by the MaxHeapify procedure in Fig. 1, from the sub-trees with lowest depth, eventually the whole array $A$ is rearranged in a heap, as done by BuildMaxHeap in Fig. 1. Some caution should be taken since the tree may not be complete, and so the first conjunct in the **if** statements of MaxHeapify. It is clear that induction on the structure of trees allows to prove that MaxHeapify, BuildMaxHeap, and Heapsort operate correctly, so the algorithm designer can formally derive the first essential property of heapsort, namely that it solves the sorting problem.

The second property the algorithm designer wants to establish is the computational complexity of heapsort: a detailed analysis of the derivation is presented in Chapter 6 of Cormen *et al.* [5]. For our aims, it suffices to notice that recursion plays a fundamental role in the algorithm, and thus computational complexity gets calculated as the solution of a system of recurrence equations.

So, the LoE $S_{E;A}$ of the algorithm designer contains at least the theory of binary trees, because of the induction and recursion principles used to derive the properties of heapsort, the mathematics of recurrence equations, to calculate the complexities, and most of what is needed by the programmer and the software designer, to ensure that heapsort will be implementable in the given architecture.

## 4 Gradients of Explanations

The heapsort example shows a very simple GoA: the $c$-inforg $P$, after the programmer has implemented the sorting procedure, exhibits two LoAs: the pair $(P_A, P_O)$ describing the computing device, and the pair $(S_A, S_O)$ having heapsort as a primitive. The LoA $S_A$ abstracts over $P_A$ by *hiding* the implementation of heapsort, and provides a new action, sorting, along with its specification; at the same time, $S_O$ extends $P_O$ by adding a new observable, the reference to the procedure Heapsort, which reifies the action. The relation between $(P_A, P_O)$ and $(S_A, S_O)$ is given by the implementation, which is the *encoding* of the action 'sorting' of $S_A$ into $P_A$. It is the presence of an encoding that justifies to consider the sequence $\langle (P_A, P_O), (S_A, S_O) \rangle$ as a GoA: encodings are an essential aspect of $c$-inforgs, as discussed in Gobbo and Benini [10, 11].

The sequence $\langle S_{E;P}, S_{E;D}, S_{E;A} \rangle$ given by the LoEs of the programmer, the software designer, and the algorithm designer as described in Sect. 3, behave similarly, being nested one into the other: in fact, each LoE explains $(S_A, S_O)$ and, in turn, each LoE explains some parts of the preceding LoEs in the sequence. For example, the algorithm designer proves that heapsort solves the sorting problem and this statement together with its proof is in $S_{E;A}$; the software designer knows the statement but has no need for the proof, so just the statement is in $S_{E;D}$; also, the programmer knows the statement because this has to become part of $S_A$ since the observable Heapsort has to be paired with a specification but, again, the programmer has no need for the proof. Hence, the sequence $\langle S_{E;P}, S_{E;D}, S_{E;A} \rangle$ is a gradient, and since it relates LoEs, it is called a *Gradient of Explanations* (GoE).

The relations between the LoEs in the gradient is similar to encodings in GoAs, but subtler. To understand these relations, it is useful to compare them with encodings between pairs of LoAs: instead of considering an encoding as the way a concept of the abstract LoA $A$ gets implemented in the concrete LoA $C$, we may think that the encoding is the way to construct the concept in $A$ from $C$. This construction resembles mathematical induction, since new objects are built starting from simpler ones. But there is a dual construction, *coinduction*, see [4]: starting from a given, large universe, coinduction operates by progressively discarding the elements of the universe not satisfying the construction property. In a GoE, the LoEs are linked to each other via a relation that follows the coinductive pattern: starting from a very general, wide and abstract universe of concepts, each LoE in the gradient refines the next one in the sequence in the sense of hiding what is not strictly needed to provide a coherent explanation of the purpose the LoE has to fulfil. The first LoE in the sequence, in turn, is re-

fined by the LoA it is called to explain. This fact suggests that each LoA should be considered a LoE with no content deputed to explain.

In the heapsort example, the LoA $S_A$ is explained by itself in an empty way—a coherent explanation, but not so useful. In turn, $S_A$ is a refined explanation of $S_{E;P}$ where the contribution of the programmer is thrown away. In the opposite sense, which is the usual one when presenting LoEs, $S_{E;P}$ explains $S_A$ by the knowledge a programmer has to provide in order to build the $(S_A, S_O)$ pair from $(P_A, P_O)$. It is important to remark that the process of constructing gradients is not necessarily inductive or coinductive: although, as a rule of thumb, GoAs follow an inductive pattern, while GoEs follow a coinductive one, it is not difficult to imagine counterexamples of any sort. It is the 'direction' of construction that matters, not the instrument, exactly as in the case of the carpenter: a chair can be made using different instruments, but the style and therefore the 'personal touch' of the carpenter is given by the direction of construction.

An immediate consequence of having a GoE is the possibility to give a *measure*—in the sense stated by Gobbo and Benini [10]—of concreteness of the various concepts that explain, in some sense, a piece in a $c$-inforg. Consider the lowest LoE $E$ in a gradient $G$ over a LoA $A$ that uses the concept $C$ under examination: we define the measure of $C$ with respect to $A$ as the distance from $A$, calculated by counting the number of LoEs separating $E$ from $A$. This measure is loosely related to Krull dimension in commutative algebra (Eisenbud [6]), but this relation is out of the scope of the present paper. This measure is a direct expression of the level of concreteness of $C$ with respect to the LoA $A$: by definition, it shows the distance between the 'concrete' basis of the GoE, the LoA $A$, and the first occurrence of the concept $C$ in the gradient. Also, the same measure can be used to classify purposes. Since a purpose becomes explained in some point of a GoE, the distance $d$ between the concrete realisation of the purpose, which is the explanation behind the LoA, and the first LoE that explains the purpose, classifies the purposes. It is important to remark that the suggested measure is relative to a gradient and based upon a LoA. It does not make sense to use this measure to compare objects not pertaining to the same LoA or not part of the same GoA.

## 5 Conclusions

By using the methods of levels of abstractions by Floridi [7], our analysis has naturally driven the reader toward a novel concept, extending the aforementioned method, which coherently fits into the epistemological framework. The newly synthesised concept, called Gradient of Explanations (GoE) is analogous to the Gradient of Abstractions (GoA) explained in paragraph 2.2.6 of Floridi [8], but applied to levels of explanation instead of levels of abstractions. Despite this similarity, which justifies why the new concept is conservative with respect to the method, a GoE has a rather different epistemological status.

The consequences of the introduction of the GoE are not yet fully explored and its formalisation is still preliminary. However, throughout this paper differ-

ences in the status between GoE and GoA were clarified, permitting to derive some of its consequences, using the heapsort example as a concrete guideline.

**Acknowledgements**

Marco Benini has been supported by the project *Correctness by Construction*, EU 7$^{\text{th}}$ framework programme, grant n. PIRSES-GA-2013-612638, and by the project *Abstract Mathematics for Actual Computation: Hilberts Program in the 21$^{st}$ Century* from the John Templeton Foundation.

F. Gobbo holds the Special Chair in Interlinguistics and Esperanto at the University of Amsterdam (UvA), Faculty of Humanities, on behalf of the Universal Esperanto Association (UEA, Rotterdam).

The content and opinions expressed in this article are the authors' and they do not necessarily reflect the opinions of the institutions supporting them.

## References

1. Allo, P.. Information and Logical Discrimination. In Cooper, B.S.; Darwar, A.; Löwe B. (eds.). *How the World Computes*. LNCS 7318. Springer-Verlag. (2012)
2. Allo, P.; Baumgaertner, B.; D'Alfonso, S.; Fresco, N.; Gobbo, F.; Grubaugh, C.; Iliadis, A.; Illari, P.; Kerr, E.; Primiero, G.; Russo, F.; Schulz, C.; Taddeo, M.; Turilli, M.; Vakarelov, O.; Zenil, H., (eds.) *The Philosophy of Information: An Introduction*. Version 1.0. Society for the Philosophy of Information. (2013)
3. Allo, P.. The Logic of 'Being Informed' Revisited and Revised. Philosophical Studies 153(3), 417–434. (2011)
4. Barwise, J.; Moss, L.: *Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena*. CSLI Publications. (1996)
5. Cormen, T.H.; Leiserson, C.E.: Rivest, R.L.; Stein, C.: *Introduction to Algorithms*. Second edition. MIT Press. (2001)
6. Eisenbud, D.: *Commutative Algebra with a View Toward Algebraic Geometry*. Graduate Texts in Mathematics 150. Springer-Verlag. (1995)
7. Floridi, L.: The Method of Levels of Abstraction. Minds & Machines 18(3), 303–329. (2008)
8. Floridi, L.: *The Philosophy of Information*. Oxford University Press. (2011)
9. Floridi, L.: *The Ethics of Information*. Oxford University Press. (2013)
10. Gobbo, F.; Benini, M.: What Can We Know of Computational Information? The Conceptual Re-Engineering of Measuring, Quantity, and Quality. Topoi. (forthcoming)
11. Gobbo, F.; Benini, M.: Why Zombies Cant Write Significant Source Code: The Knowledge Game and the Art of Computer Programming. Journal of Experimental & Theoretical Artificial Intelligence. (in publication)
12. Gobbo, F.; Benini, M.: From Ancient to Modern Computing: A History of Information Hiding. IEEE Annals of the History of Computing 35(3), 33–39. (2013)
13. Gobbo, F.; Benini, M.: The Minimal Levels of Abstraction in the History of Modern Computing. Philosophy & Technology. (2013)
14. Knuth, D.E.: *The Art of Computer Programming, Volume 3, Sorting and Searching*. Second edition. Addison-Wesley. (1998)
15. Primiero, G.: Offline and Online Data: on Upgrading Functional Information to Knowledge. Philosophical Studies. (2012)