The Nature of Computer Science. (first draft)

0. Preface.

In the title of my previous book --"A Discipline of Programming"-- the indefinite article was chosen quite intentionally; the definite article in the title of this one is equally intentional. In the previous book I presented a style of programming and offered a methodology because I had found them helpful; in this book I want to convince my readers of what I now feel to be essential.

I want to give my readers a feeling for the flavour of the quintessence of Computer Science as I see it, and I want to do so because there is too much unclarity and confusion about what Computer Science ought to be. The sources of this unclarity and confusion are many, one of them being the difference of opinion among the so-called "experts", i.e. the people called "computer scientists", who have a quite understandable tendency to regard their work as falling under --or sometimes even: characteristic for-- "Computer Science". And their professional activities cover a very broad spectrum, so broad that the confusion about what constitutes Computer Science is only too understandable. The incoherent diversity of all these activities is, however, fairly easily disposed of: upon closer scrutiny one will often discover that either the major concern is not with computers or the activity has very little to do with science (or both). But in the meantime, the seed of confusion has been sown!

This booklet addresses itself to the most ready and most serious victims of this confusion, of this unclarity, and of all this misunderstanding: the otherwise educated layman. I have mainly two specimina in mind.

The one is the general scientist --the astronomer, the biologist, the chemist, etc., but possibly also the mathematician-- whose scientific education and professional experience have been in areas other than Computer Science, but who now regrets that he has no feeling for the flavour of the subject. The other one is the young student who hasn't yet made up his mind what kind of scientist he would like to become and wonders whether computer

science would be an attractive subject. I shall be very happy if I can assist him, for he has a very difficult choice to make, as he will know its full consequences only many years after he has made it.

A warning to the reader might be appropriate. Knowing that I am addressing the uninitiated and remembering that it is my purpose to make him understand, I shall do my best to be as clear as possible. Even if I would have attained my goal, the reader should not expect a text that is "easy to read" if "ease of reading" is related to the number of pages of print, the contents of which the reader can absorb per unit of time. This measure of the "ease of reading" --although not unusual-- is irrelevant, as it allows the author to score high by the simple device of making his text unnecessarily verbose. The relevant criterion --although much harder to quantify and, therefore, applied less frequently-- would be something like the quotient "insights gained by reading / effort spent on reading". It is in this last sense that I aim at scoring high. In other words: I hope that, after an hour of intense (and possibly hard) reading, my reader will enjoy how much he has learned, rather than be disappointed by how few pages he has progressed. (After all: it is only a slim volume!) Writing for the uninitiated does not imply --as is sometimes thought-- adopting a style that would fit a Sunday paper; doing so would, as a matter of fact, defeat my purpose, as I intend to do serious business.

In order to avoid false expectations (and also to forestall forseeable, but unjustified criticisms) I would like to stress why this booklet did not get the title "An Introduction to Computer Science".

From having read an "Introduction" we can expect very specific forms of assistance when embarking upon a more detailed study of whatever aspect has taken our fancy. It so happens that nowadays, rightly or wrongly, the members of the brotherhood that is associated with each identified intellectual discipline use in their internal communication an extensive jargon that is specific for their brotherhood. In this respect Computer Science --and also even many of its subfields!-- is no exception. (On the contrary, I am tempted to add.) From an "Introduction" one is entitled to expect an adequate coverage of the more important part of the jargon of the field it introduces the reader

to: the apprentice needs to know the meaning of these terms in order to understand what the experts say to each other.

It is my clear purpose, however, to leave my reader the option of not pursuing the topic of Computer Science any further. In consequence I intend to avoid jargon as much as possible.

From an "Introduction" one is similarly entitled to expect a broadness of coverage, a certain completeness: all in some sense "major" areas should be covered. But the transfer of knowledge in the form of transmission of facts is most definitely not the purpose of this book. I shall not hesitate to mention facts when I need them to make my points (for instance, when I feel that I should provide at least one example of what I am talking about); completeness not being my purpose, I shall also not hesitate to omit them when I feel that they are not really necessary.

(Preface to be continued later.)

1. The notion of "algorithms".

> "But if thought can corrupt language, language can also corrupt thought."
>
> George Orwell (in "Politics and the English Language.")

I think the way in which people use --or misuse-- words always most revealing. (At a seminar I recently attended, one of the speakers consistently referred to people as "human beings"; he turned out to have been trained as a psychologist.) Bearing that in mind and, furthermore, remembering that the majority of the people in the field regards computers primarily as tools, we should notice that the English speaking world coined the term "Computer Science". We should do so, because it is very exceptional that a tool gives its name to a discipline: we don't call painting "brush art", nor

surgery "knife science". From these observations we can only conclude that when the term "Computer Science" was coined, computers were regarded --either in fact, or mainly potentially-- as exceptional gadgets. A question to be answered before proceeding is, whether this view of computers as exceptional gadgets is justified or not. (I hope to convince my readers it is.)

If there were nothing exceptional about computers, we should regard the term "Computer Science" as a misnomer, and at least the title of this book should have been different, perhaps this book shouln't even have been written at all. If, on the other hand, there is indeed something exceptional about computers, we had better know precisely what. I even go further: in that case we should never forget that --and why-- computers are properly regarded as exceptional gadgets, because then their exceptionality could very well invalidate (and should at least challenge) many opinions and assumptions about computers and their usage that, upon closer scrutiny, are solely supported by analogies from other fields. (When in 1968 the term "Software Engineering" was coined in Europe, I was very happy with it, because I felt that it captured both the programmer's commitment to complete in time the design of a reliable artefact, and the constructive nature of his work very well. Ten years later I have reasons to regret its wide-spread adoption, for, to my taste shallow, analogies with engineering practices established in other fields seem now to have created more confusion than enlightment.) In short, the question why and to what extent computers are exceptional gadgets is a vital one, and the first two chapters will mainly be devoted to it.

The Computer's exceptionality manifests itself in two different ways that can be dealt with separately to a very large extent. The one way is a primarily quantitative aspect, made possible by modern electronic technology: not only can they process information --here, forgive me, I am using still loose terminology-- but they can do so very, very fast indeed, and not only can they store information --here, again, I am using still loose terminology-- but at any moment in time they can store very, very vast amounts of information indeed. In these quantitative aspects, all by themselves, the computer is already quite remarkable, because it is in these aspects that the modern computer can justly be regarded as one of the major triumphs of electronic technology. We shall deal with these quantitative aspects in our next chapter.

The other way in which the computer manifests its exceptionality is of a more qualitative nature and is reflected by its ability to carry out (in a sense) "any" algorithm, and, please, don't ask me now what is meant by that: the remainder of this chapter will be devoted to its explanation. Before embarking upon that explanation, however, it is worth noticing that also in this aspect, all by itself, the computer is already quite remarkable. If it had not been for this qualitative aspect of his work, Charles Babbage, if remembered at all, would only be remembered as one of the last century's major influence on mechanical toolmaking. Now he is remembered as one of the radical and revolutionary minds of the previous century --sadly misunderstood in his time-- , whose vision would require another hundred years to become reality and to have its full impact.

In the preceding paragraphs I have only indicated two very different ways in which the computer is remarkable: it is their combination that makes the computer exceptional. (It is this combination that we find reflected in popular titles such as "Faster than Thought" or "Giant Brains, or Machines that Think" --mentioning these titles does not mean that I recommend the corresponding books-- .) For the sake of clarity we shall deal with these two so different aspects in turn.

* * *

The informal analogue of the algorithm is quite familiar to us, from our fairy tales, from our daily life, and from our professional activities. It is the set of rules that, when followed faithfully, is guaranteed to establish a certain result. I mention a few examples:

--- the magic formula that, when spoken correctly, will cause some wonder to happen (will open a rock, cause a jinnee to return to its bottle or a devil to leave the body it has taken possession of)

--- a recipe for a dish or a cocktail

--- a knitting pattern

--- the answer one gets after having asked for the way from A to B

--- the Instructions for Use that accompany a piece of equipment

--- the rules how to add or multiply two decimal numbers.

Although only informal analogues, they can teach us something about

algorithms. For example, the total result is reached as the cumulative effect of acts, each of a simpler nature than the total one. In an old cookery book I found the following recipe for peppered hare: "One takes a hare and prepares peppered hare from it." It is not actually wrong, but hardly helpful.

The performance of the individual acts must be possible. They must be logically possible: the advice, given after someone had asked for the way: "Go straight ahead and turn to the left at the last traffic lights." presents such a logical impossibility. It must also be physically possible: "Go to the devil!" --or "Walk to the moon!" as we say in Dutch-- is, for its sheer impossibility, not a command, but a curse. A little more subtle is the case of the instructions for use of some heating equipment, telling you that, in order to light it, you have to press button A until a flame appears. If, before getting tired of pressing button A , you see the flame appear, you are OK, but what if it does not? How long are you going to continue to press that button? We cannot solve the problem by saying: "Well, the instructions only apply to a heater in proper condition.", for how do we know that our heater is not in proper condition? We cannot conclude that from the fact that the flame has not appeared yet. We would have been in much better shape if the instructions had said: "Light the flame by pressing button A for at most 10 seconds."

Clarity, in the sense that there is no misunderstanding about the acts to be performed, is also essential. In this respect knitting patterns --at least when published in a ladies journal of high standards-- are as a rule not too bad, and on the average much better than recipes, in which it is not unusual that the instruction that, finally, spices have to be added according to taste, leaves the major decision to the cook. (And, accordingly, as most of us know from sad experience, following cooking instructions as faithfully as we can is not guaranteed to establish the desired result.)

With the exception of the rules for adding or multiplying two decimal numbers, the above examples all have the familiarity of daily life. That familiarity is their strength, but also their weakness, for, if you come to think about it, daily life is very confusing. The worst of such examples is that it is never clear where the example ends and the confusion of daily life begins.

I would, for instance, like to argue that most of the examples given are too simple in the sense that they are algorithms which, when followed, always lead to the same result. The magic formula that opens the rock only works for a very specific rock, and everytime the formula is spoken, the rock opens up in exactly the same manner. But does it? Is opening the rock on a Sunday the same as opening it on a Friday? Are we sure that opening it on a Friday the 13th does not have a deep, hitherto unsuspected, mystical meaning? If we follow the instructions how to walk from A to B twice, have we made the same walk twice? Both times we arrived at B , but what if we did it once on a sunny day, and once in a pouring rain? Let us follow the knitting pattern twice, once with blue and once with red wool: to what extent are the two resulting sweaters the same?

For centuries such questions have been the delight of philosophers. I like to believe that by now most of us would regard them as pretty meaningless. In any case our business is not philosophy, but science.

The scientist's escape from philosophy is --in principle at least!-- a simple one: he replaces the fuzzy, open-ended universe of discourse as inspired by our daily experiences --also referred to as "reality"-- by a postulated universe of discourse of his own invention, in which, by definition, what he talks about is ... what he talks about! It will be remarked that this postulated universe of discourse of his own invention "has been inspired by his intuitive understanding of reality", but to the extent that this postulated universe of discourse is truly his "universe" of discourse, the scientist is entitled to regard that remark as irrelevant.

The oldest example of this technique of liberating ourselves from the shackles of empiricism are of course the "Elements" of Euclid, whom E.T. Bell characterizes as "the great perfector, if not the sole creator, of what is today called the postulational method, the central nervous system of living mathematics." [ ]

[ ] Bell, E.T., The Development of Mathematics, McGraw-Hill Book Company, New York - London, 1945, p.72

(Many people find it hard to swallow that the best thing scientists --even applied scientists!-- can do with "the real world" is to ignore it. As E.T.Bell remarks in this connection:

> "Mathematicians and scientists of the conservative persuasion may feel that a science constrained by an explicitly formulated set of assumptions has lost some of its freedom and is almost dead. Experience shows that the only loss is denial of the privilege of making avoidable mistakes in reasoning. As is perhaps but humanly natural, each new encroachment of the postulational method is vigorously resisted by some as an invasion of hallowed tradition. Objection to the method is neither more nor less than objection to mathematics.")

What Euclid did for points, straight lines and circles, was done in this century for algorithms --or, as they are also called,"effective procedures"-- by A.M.Turing. The algorithm (or procedure) consists of a finite set of rules such that the application of any such rule only requires the distinction between a finite number of different cases; the application of the next rule is called the next "step", and the procedure is called effective when it is guaranteed to terminate after a finite number of steps. The rule for adding two decimal numbers is an example of an effective procedure: besides the rule telling us when we have finished there is almost only one other rule, viz. what to do at each decimal position before moving to the next. In general we are faced at each decimal position with 1 out of 200 , and thus a finite number of possibilities: 10 for the one digit, 10 for the other, and 2 for the absence or presence of a carry from the right.

The above is still much too loose for serving as the basis for a mathematical theory of algorithms: what is, and what is not, a permissible individual rule is, for instance, still much too woolly. Because the individual digits of the numbers to be added are "given", but the carry from the right only emerges as the calculation proceeds, it could be argued that it is more proper to introduce two rules, one for the absence of a carry from the right and one for the presence of a carry from the right, and each of them catering for the 100 two-digit combinations from the numbers to be added.

Turing put an effective end to such discussions by proposing a very rigid scheme, and an algorithm given in accordance with that rigid scheme is now known as "a Turing Machine". In order to drive home the message that an algorithm can be carried out mechanically, i.e. without further insight, guesswork, experience, tacit understanding, and divine inspiration, Turing defined a class of automata and claimed that for each effective procedure an automaton from that class could be designed.

This is not the place to present the theory of Turing Machines in any detail; yet I hope that the following sketch will suffice to give the reader a first glimpse of Turing's achievement. A Turing Machine consists of a finite part and an infinite part (of which in each terminating computation only a finite section will be used).

The infinite part consists of the so-called "tape", an infinite sequence of "squares", each of them capable of holding one character from a finite alphabet. In the beginning all squares of the tape hold the same neutral character --usually called "blank"-- with the exception of a finite sequence of squares that may hold other characters from the alphabet, thus presenting "the input". The contact between the infinite tape and the finite part of the Turing Machine consists of the circumstance that one of the squares enjoys the privilege of being "the scanned square".

The construction of the finite part embodies the algorithm proper. The finite part is a so-called "finite state machine", with a state for each rule. Its construction can be given in tabular form, as the permissible rules follow a rigid format: a rule prescribes for each of the values of the character currently held in the scanned square

1) the new character to be written in the scanned square

2) whether thereafter the tape has to be moved one place to the left or one place to the right --i.e. which of its two neighbours will be the scanned square on the next step

3) the (number of the) rule to be applied in the next step.

One special state of the finite part is called "the halting state"; when it is reached, the computation is regarded as successfully executed and the contents of the squares of the tape at that stage are regarded as the

final result. That is all!

Turing Machines are in two ways a very remarkable conception. As we described them, squares hold a character from a finite alphabet, but nothing essential is lost if we restrict ourselves to alphabets of two characters only. More precisely: given a Turing Machine in a multi-character alphabet, we can derive --at the "expense" of introducing many more states in the finite part-- an equivalent Turing Machine in which communication at the scanned square takes place via a two-character alphabet only.

As we described it, the finite part comprises a finite number of states: at the "expense" of greatly enlarging the alphabet we can deduce an equivalent Turing Machine with only two states (besides the halting state).

The theorems mentioned in the two preceding paragraphs are far from trivial, and to any mathematician of some maturity their mere existence strongly suggests that the notion of Turing Machines is something much more deep and fundamental than might appear at first sight. For practical purposes of computation Turing Machines are without significance; the notion of Turing Machines is, however, highly significant from a more theoretical point of view. The rigidity of the regime they embody --their "austerity", so to speak-- made it possible to define functions for which it could be proved that no Turing Machine can compute them. (This is usually proved by a reductio ad absurdum: one defines a function --using, admittedly, the notion of Turing Machines-- and shows that the assumed existence of a Turing Machine computing it leads to a contradiction. The argument of Turing's is very ingenious --M.L.Minksy [ ] introduces the final contradiction with the four words "Now for the killer!"-- ; it would never have been possible, had not

---

[ ] Minsky, M.L., Computation: Finite and Infinite Machines, Prentice-Hall, Inc., Englewood Cliffs, 1967, p.149

---

the notion of an algorithm been reduced by Turing to its bare essentials.)

The discovery of well-defined functions that cannot be computed by a Turing Machine would lose much of its significance if it could be argued

that Turing Machines as models for algorithms --and, hence, the existence of a Turing Machine as criterion for computability-- are unnecessarily restrictive. The second way in which Turing Machines are very remarkable is that no one has been able to provide such an argument. Quite a few quite "reasonable" alternative definitions of computability of functions have been proposed, but on closer inspection all these proposals turned out to be equivalent to the question whether a Turing Machine computing the function was conceivable or not.

Of Turing we can really say that with his rigid regime he reduced the notion of an algorithm to its bare essentials: the carefully chosen constraints that make the concept mathematically manageable don't seem to impair the concept's generality.

Now, almost half a century after Turing's breathtaking work, the notion of effective computability of a function is equated with the conceivability of a Turing Machine computing it. In the beginning, when the fuzzy notion of "effective computability" was still given an independent right of existence, this equivalence was regarded as an article of faith; nowadays this equivalence is used to define the notion of effective computability.

Despite the fact that Turing's paper of 1936 "On computable numbers, with an application to the Entscheidungsproblem" was published in such a respectable journal as the Proceedings of the London Mathematical Society, and despite the fact that he had done for the theory of algorithms what Euclid had done for plane geometry, the mathematical world at large has been amazingly slow in recognizing the profound significance of Turing's work. (For instance, E.T.Bell's "The Development of Mathematics" of 1945 does not mention Turing at all, whereas it mentions K.Gödel about ten times!) It seems to have hardly attracted any attention for almost fifteen years. One of the explanations offered for this curious state of affairs is the very perfection of Turing's original article, in which he created a new theory but completed it at the same time: for lesser souls he had left nothing to be added by them, and, hence, his paper did not start an avalanche of subsequent ones in which it was quoted. (And, as we all know, the standard criterion for a successful paper is that it gives rise to at least ten subsequent ones.)

Eventually the significance of Turing's work became more widely recognized, but this was after Turing had died as an apparently disillusioned and perhaps even somewhat embittered man. [See note 1 on EWD682 - 13.]

For the theoretically inclined the theory of computability has an intellectual fascination in its own right; here we have used Turing's invention only for giving the reader some idea of what a formal theory of computation looks like and a somewhat better appreciation of what a mathematician talks about when he talks about an algorithm.

The algorithm is a finite set of rules from a well-defined repertoire --or, equivalently, a finite text in some suitable notation-- and corresponds to the finite part of the Turing Machine.

The input --also called "the argument" of the function to be computed-- corresponds to the finite sequence of characters from a finite alphabet, as represented by the initial contents of the squares on the tape.

The output --also called "the result" of the computation or "the value" of the function to be computed-- corresponds similarly to the final state of the tape (under the assumption that the Turing Machine has reached its halting state).

By its very construction a Turing Machine defines a final character sequence for each initial character sequence that would lead to a properly terminating computation of that Turing Machine: for all those "arguments" it defines a "result". Different arguments may correspond to the same result, but no argument may correspond to different results; in other words it is a many-to-one-correspondence, i.e. a function.

It is --and this is important to remember-- characteristic for such a function to be defined for very many --usually even for infinitely many-- different values of the argument. Loosely speaking, the algorithm "captures" what all computations from that very large --and, as said, usually infinite-- class "have in common".

For the designer of algorithms this has a far-reaching consequence when combined with a requirement mentioned earlier, viz. that the algorithm, "when followed faithfully, is guaranteed to establish a certain result". If the number of different arguments for which a result is prescribed is small enough, such a guarantee could be based on experiment: one applies the proposed algorithm in turn to the distinct arguments and inspects each time whether the actual result equals the prescribed one. This simple method for creating confidence in the correctness of a proposed algorithm is, however, almost never feasible, because the function to be computed is almost always defined for a prohibitively large number of different arguments: the set of individual computations that can be tried is almost always an absolutely negligible fraction of the class of possible computations. Under such circumstances we would prefer to base our confidence on a reasoning that is independent of the specific argument value with which the computation started. Being independent of that specific argument value, such a reasoning applies to the whole class of computations. Remembering the loose remark that, in a sense, "the algorithm captures what all computations from that whole class have in common", we shouldn't be surprised to see the algorithm in the form of "a finite text in some suitable notation" playing the central role in such reasonings. We shall return to this later; for the time being we confine ourselves to the remark that this paragraph reveals why competent design of algorithms is intrinsically an activity of a mathematical nature.[Note 2, next page]

Remark. In retrospect I am willing to regret the terminology in which Turing couched his message: the metaphor of the Turing Machine (with its infinite tape of squares moving to the left or to the right) is too vivid. It is so vivid that too often it blurs the fact that the true status of a Turing Machine is that of a mental experiment, neither more nor less: it is a concept to which the attributes of space and time are not applicable. A Turing Machine never "exists" to the extent that it could make sense to ask "Where did that computation of that Turing Machine take place?" or "How much time took that computation of that Turing Machine?". It does not even make sense to ask "How many steps took that computation?" A meaningful question could be "To how many steps amounts that computation?".

I don't blame Turing too much for the vividness of his metaphor: even

if he had chosen a more sober terminology with fewer connotations, he could probably not have prevented senseless comments such as that "Turing Machines are too slow and inefficient for practical purposes". Such a remark makes as much sense as talking about the honesty of a painting, the temperature of a poem, or the health of a thunderstorm. (End of remark)

---- note 1, to be inserted EWD682 - 11

Still in 1978, Turing is supposed to be unknown among the members of the Association for Computing Machinery. In the following quotation (Comm. ACM, 21, 9 (Sep. 1978), p.798) the indefinite article tells the whole story all by itself: "The Turing Award is presented in commemoration of Dr.A.M. Turing, an English mathematician who made many important contributions to the field of computing."

---- note 2, to be inserted EWD682 - 12

To present a contrary opinion, picked at random (Comm.ACM, 21, 9, (Sep. 1978), p.800): "A panel chaired by Stanley Winkler, IBM, concluded that computer science is primarily an experimental discipline in which students require a spectrum of facilities to develop their understanding. Drawing an analogy to the medical profession in which anatomy is learned by dissection of cadavers, Winkler's panel suggested that the computer science field should adopt the case-study method for teaching. The computer science curriculum should stress a braod background, goal orientation, and team experience. Universities should recognize that projects and the teaching of the design concept are quite important." etc.
