A story that starts with a very good computer.

Once upon a time, a long time ago, an organization decided to get a computation centre. The organization hired a manager to manage the computation centre, and he was a very competent manager, for he hired a very good computer to do the computing and a very good programmer to do the programming. The manager's high quality was shown by his choice of computer: knowing that in the work of his organization, sorting would play a very big role, he selected the one and only computer on the market that had a very fast, built-in sort instruction — called "SORT"— in its instruction code. The manager's high quality also manifested itself by the choice of the programmer, as will become clear in the sequel.

The machine was installed, and the main application program, in which the instruction SORT occurred 77 times, was written and proved to be correct. The programmer could do so because for each of the instructions of the order code —SORT included— the reference manual gave him the functional specifications on which to base his correctness proof. The main application program was put in operation and everybody in the whole organization was instantaneously happy... until, after the first month of operation, the electricity bill arrived! The bill was very high...

Suspicion, quite naturally, fell on the new computer and the manager inspected its power consumption more closely. He discovered that the SORT-instruction was the culprit, and asked his programmer, whether he could reduce the power consumption of his program. The programmer made a more detailed study of the power consumption of the SORT-instruction and discovered that it rose steeply —more than quadratically, as a matter of fact— with the length of the array to be sorted. And as almost all his 77 calls of the SORT-instruction were on rather long arrays, he understood the height of the bill immediately, and also realized his only hope for reducing the power consumption: shortening the length of the arrays supplied to the SORT-instruction.

He decided to replace all 77 occurrences of the SORT-instruction in his main application program by calls on a subroutine (still to be written) that he modestly called "save0", and in order that the correctness proof of the main program would remain valid, he decided that the functional specifications of save0 would be identical to those of SORT.

He thought for a long time how to construct the body of save0. He then came up with the following idea. If the array consists of less than two elements, it is sorted by definition, and control can return immediately. Otherwise, by (if necessary, repeatedly) swapping two values when the larger was to the left of the smaller, he managed to rearrange and divide the array in such a way, that the largest element in the left-hand section did not exceed the smallest element in the right-hand section; thereafter he gave two SORT-instructions, one for each section.

The programmer was very pleased by what he had done: the correctness proof for the main application program remained automatically valid, his only additional proof obligation had been to prove the correctness of the body of save0 —but he had already some experience in proving the correctness of programs using the SORT-instruction and that helped—.

Also the manager was very pleased, for this minor program change —it was hardly a "change": it was almost only an addition— indeed had cut the electricity bill by more than a factor two! But improvement, like all novelty, wears out, and after a few months the manager asked the programmer whether he could reduce the still high power consumption yet further. This time the programmer said instantaneously "Oh, yes.", for now he knew the trick: he introduced a subroutine save1, the body of which was a copy of the body of save0, and thereafter replaced in the body of save0 the two occurrences of the SORT-instruction by calls on save1. The programmer was extremely pleased with himself, for this time he had reduced the power consumption by a further factor of two, but he had done so without any further proof obligations!

The manager was also pleased, but only for a month or two. When he asked his programmer again, whether he could reduce the power consumption still further, the programmer, again, said immediately "Oh yes." but went to his desk to do some sensible coding. He could have repeated the trick by introducing a new subroutine save2, etc., but by now he knew that, a few months later, the manager would come again. Besides that, he did not like the prospect of filling more and more of the store with almost equal copies of the same subroutine. He decided to map the texts of save0, save1, save2 etc. on the same general text — which he called saven— at the expense of a global variable n — initialized in the main program at zero— the value of which should indicate whether a call on saven should act as save0, save1, save2 etc. The body of saven was derived from the ones of save0, save1, etc.: upon entry, n was increased by 1, just before return, n was decreased by 1, and the internal calls on the next save or on SORT were replaced by

$$\underline{\text{if}}\ n < N \rightarrow \text{saven} \quad ! \quad n = N \rightarrow \text{SORT}\ \underline{\text{fi}} \qquad (1)$$

and he satisfied his manager by setting the constant N = 3. As he had foreseen, a month later he was asked to reduce the power consumption still further: he just increased N by 1.

Having thus mechanized the optimization process that reduced the power consumption, the programmer gladly increased N by 1, every time he was asked to reduce the power consumption, and that was about once a month.

After a year or so, the manager discovered that, lately, his programmer's optimizations had become less and less effective. As he was a very competent manager, he investigated the matter; in the course of his investigations he discovered that the SORT-instruction was hardly invoked at all! This discovery worried him, because for that SORT-instruction his organization paid a lot of money: for a much lower rental price the manufacturer offered a model without SORT-instruction, but otherwise identical. The manager went to the programmer, telling him his observation that the SORT-instruction was hardly exercised: could the programmer avoid its use completely? For then they could replace their expensive machine by a cheaper model!

This time, the programmer had to think again. Looking at (1) —the only place left, where the SORT-instruction still occurred— he realized that <u>if</u> n remained under an upper bound, he could choose N larger than that upper bound, with the result that the second alternative of (1) would never be selected! By inspecting his main application he could prove that N = 25 would be large enough, and he replaced (1) by

$$\underline{\text{if}}\ n < 25 \rightarrow \text{saven}\ \underline{\text{fi}} \qquad (2)$$

Later he realized that, having proved that the guard would always be true, he could simplify the program still further by replacing (2) just by

$$\text{saven} \qquad (3)$$

Now he was completely happy: with the last simplification the correctness of his program was no longer dependent on the exact value of the upper bound, but only on its existence. The machine was replaced by the simpler model and the manager, too, was happy ever after.

*   *
*

The above fairy tale —like all fairy tales, for that matter— has been written for educational purposes. It deserves to be remembered because it is a sobering thought that, upon investigation of his manager, a programmer engaged on optimization could have discovered all this —with the exact nature of the proof obligation included!— long before mathematicians called it Recursion.

Plataanstraat 5 prof.dr.Edsger W.Dijkstra
5671 AL NUENEN BURROUGHS Research Fellow
The Netherlands

---

transcribed by Corrado Cantelmi
revised Mon, 12 Sep 2005