February 1994

Preface to Quicksort

I have been very lucky. What a wonderful way to start a career in Computing, by discovering a new sorting algorithm! But it was only a matter of chance. In 1960 I was studying in Kolmogorov's department at Moscow State University, when I received an offer of a post as Senior Scientific Officer from the UK National Physical Laboratory, to work on a new project for Machine Translation from Russian to English.

So I was strongly motivated to study the current Russian literature in this field. The first task of a machine translation algorithm was to look up the words of each sentence in a dictionary held in alphabetic order on magnetic tape. To achieve this in a single pass, it was necessary first to sort the words into dictionary order. My first thought on how to do this was bubblesort and, by an amazing stroke of luck, my second thought was Quicksort.

Another stroke of luck was the discovery of the exact formula estimating the average running time for the algorithm. It was this that encouraged me to write up the method as a longer article and submit it as my first published article in English (after one in Russian and one in ALGOL).

But the luckiest break of all was that I did not take the job with NPL: it turned out that my classical education disqualified me from a permanent post in the Scientific Civil Service; and anyway I had convinced myself that machine translation of natural languages was impractical with the technology of that day. Instead, I obtained a job with a small computer manufacturer, where I engaged in the translation of artificial languages, in particular the programming language ALGOL 60.

Preface to Proof of Correctness of Data Representations

After completion of a compiler for ALGOL 60, my next job in Industry was the design and implementation of an operating system to go with it. The project was a failure; and when I moved to University I started on a course of research to try to understand the subject better. At the same time, I was studying the fundamental class concept of Simula 67, the world's first Object-Oriented language. My research method was to try to identify the relevant concept of correctness, and then formalise the proofs that would be needed to establish it. As a case study, I chose the design of a simple paging system (virtual memory).

Quite suddenly, these three themes of my research led to a solution: an abstraction function which relates the concrete detail of an efficient implementation (partly on main store and partly on disc) with the simpler abstract concepts (of a contiguously addressed memory), which more directly correspond to the user's view of the data. I realised at once that this single idea was crucial to the verification of computer programs of significant size. The value of the idea has been subsequently confirmed by its application in the specification notations Z and VDM, and in programming the notations of Modula, Ada and other object-oriented languages.

I am now more enthusiastic about the idea than ever. It is the same idea that appears as a coordinate change in the calculus, or a homomorphism in algebra; its fundamental role in Computing Science is quite similar. Like all scientists, our main concern is explanation of phenomena at many different levels of granularity and abstraction. In physics, these range from quarks and particles through atoms and molecules, to cannonballs and planetary systems. In computing they range from electronic circuits and assemblies through machine architectures and high level languages to windows, databases and safety-critical real-time control systems. In all cases it is the abstraction function (generalised sometimes to a relation) that makes explicit the necessary conceptual link between the conceptual levels, and enables the professionally motivated engineer to ensure that each level provides a simple interface and a reliable implementation for the levels above it.

Preface to An Axiomatic Definition of the Programming Language PASCAL

One of my goals in the study of program proof techniques is to assist in the design of better programming languages, ones which make it easier to write correct programs and harder to write incorrect ones. I hoped that this would provide not only an objective methodology but also a scientific criterion for future research into programming language design by other workers in the field. The axiomatic basis, pioneered by Floyd, gave the insight that I followed up in a series of research studies, treating various standard features of high level programming languages. I was quite surprised that already by 1972 I felt competent to tackle a reasonable subset of an existing programming language, PASCAL, – admittedly one which had been extremely well designed to achieve the same goal of making correct programs easier to design and write.

It is a matter of continuing regret that so few languages have ever been designed to meet that goal, or even to make significant concessions towards it. For example, the programming language C was designed to assist in writing a small single-user operating system (UNIX) for a real-time minicomputer (PDP 11), now thankfully obsolete. For this purpose, its low level of abstraction and plethora of machine-oriented features are entirely appropriate. For all other purposes, they are a nuisance. The successful propagation of the language can be explained by accidental, commercial, historical and political factors; it is hardly due to any inherent quality as a tool for the reliable creation of sophisticated programs.

So is there any point in the study of program correctness? I think there is, for two reasons: scientific curiosity and professional integrity. Science is driven by human curiosity, the burning desire to know not only what happens, but how and why. Everyone knows that computer programs work, just as they know that the light of the sun and moon are sometimes obscured. It is the scientists' job to explain why; and to predict when it is going to happen. The prediction of eclipses requires an understanding of the fundamental concepts of force and gravity, and of abstruse methods of mathematical calculation. Though highly unpopular at first, these later turn out to be far more generally applicable, even useful.

Professional integrity is the driving force that leads to reliable practice in

all branches of engineering, and in other professions like medicine and law. Surgeons and lawyers spend years in theoretical study as well as in practical exercises, to make sure that they get it right first time, every time. No serious engineer would tolerate the trail-and-error methods of program debugging which are so widely taught and practiced today by coders in Industry. It is my experience that when programmers discover that they have the knowledge and the skill to write programs that predictably contain no errors, they will take increased professional pleasure and pride in doing so, with beneficial effects both on productivity and on quality of the product.

Although correctness is much easier to achieve in the clean controlled laboratory conditions of PASCAL, with a moderate amount of additional care, the same methods of assertions and abstraction can be well applied in the more polluted software environments which are more widely available today.

Preface to An Axiomatic Basis

In 1960, it was still possible to describe a complete programming language in thirty pages of mixed formalism and English prose (Naur 1960). I was then working for a small computer manufacturer, and the description of the language was clear and complete enough for me to design an implementation entirely without consultation with the original language designers. And our customers could write programs in the language, also without consultation with me or with the language designers. And when the customers' programs were submitted to the implementation, it was possible for them to run first time. A large part of the credit goes to the simple concise notation for formal specification of the syntax of the language, due to Backus. This inspired the search for a similar formal notation to express equally simply the semantics of future languages.

In the early nineteen sixties, the favoured approach was to construct a detailed mathematical model of how the program should be executed. I was disappointed at the length and tedium of such definitions, well illustrated in the case of PL/I by the painstaking but valuable work of IBM Vienna Development Laboratory. In 1964 I suggested that it would be simpler (and more useful to programmers) to describe the intended purpose and effect of each language feature, rather than its method of implementation. But it was not until I saw the work of Floyd (1967) that I knew how to achieve this goal.

In 1968 I moved to an academic post and could write this paper as a Professor in Belfast; I considered this more appropriate than a job in Industry for research in an area that was unlikely to find much practical application till after my date of retirement. Meanwhile, I predicted that a more formal approach to programming (like the traditional Classical education, which I enjoyed) might serve as a good academic training in rigorous thought, and not just for computer professionals. Both of these judgments seem still to be valid.