Chapter 8
Enrichment Topic: “Efficient” Pairing Functions, with Applications

—Entia non sunt multiplicanda praeter necessitatem.
(Occam’s razor, William of Occam)

We have already seen the preceding admonition by William of Occam (fourteenth century), in Section 6.1.2. The principle that underlies the admonition—always to strive for simplicity—is particularly worth heeding when one seeks mathematical models of computational phenomena, for it is always tempting to embellish one’s models with “real” features of the phenomenon or structure being modeled.

The (mathematical) success of models such as the Turing machine testifies eloquently to how far a truly bare-bones model can take you, even when studying sophisticated notions such as computation; cf. [104] and Section 3.3.

Within the spirit of Occam’s razor, this chapter provides a (very) short guided tour through the world of pairing functions—bijections between $\mathbb{N}^+ \times \mathbb{N}^+$ and $\mathbb{N}^+$—as tools for reducing the representational complexity of complex computational “situations.” In a word, pairing functions allow one to represent families of structures that seem inherently to have multidimensional structure as sets of integers. We illustrate the benefits that can accrue from such simplification via two examples:

1. It is not clear how to devise efficient mappings into computer memory (or computer storage) for multidimensional arrays/tables that can change their shapes dynamically, i.e., at run time. Focus, for instance, on an extendible $2^n \times n$ table (which could represent a relation in a database). Say that one wanted to add a row to the table in the course of a calculation. (Many programming languages allow such dynamic changes to the dimensions of multidimensional arrays/tables.) How should this change in the “logical” table be accommodated in its “physical” storage layout? A naive approach would relocate/remap the entire new table, but this option—which is the one adopted by many programming-language implementations—would force one to remap roughly $n2^n$ table entries in order to accommodate a change to only $n$ entries! What is the alternative? Section 8.3 is devoted to a sophisticated approach to storage mappings that is based on the
use of pairing functions and that obviates reallocating any already-stored table entries.

2. The Internet has given rise to new modalities of computing wherein the owners of computers “volunteer” their computing resources to others, for reasons ranging from curiosity to charity to the hope of compensatory computing support; examples appear in [8, 16, 54], among other sources. One hallmark of many of these Internet-based computing projects is that the participants are unknown to one another, hence untrusted. Indeed, David Anderson, the director of the well-known SETI@home project [54] has been reported as saying:

“Fifty percent of the project’s resources have been spent dealing with security problems … the really hard part has to do with verifying computational results.” The report said that Anderson went on to elaborate: “Seti@home software had been hacked – some were malicious, others not – to make it run faster, to spoof positive results and to make it look [as if] more work had been performed to improve leader board rankings.”

http://www.wired.com/news/technology

Given a project with such experiences, one could imagine a desire to keep track of which “volunteers” produced which results, so that one could ban repeat offenders from subsequent participation in the project. Putting aside thorny issues such as how to reliably identify “volunteers” (IP addresses can easily be spoofed), one is left with the challenge of efficiently associating “volunteers” with the results they have produced. In Section 8.4, we somewhat simplify this accountability problem, by using pairing functions to translate the (volunteer, result-index) ordered pairs to single indices.

Of course, we are not going to abandon the reader with just a couple of proposed uses for pairing functions in modern computing settings. We shall, in fact, embellish these proposals with discussions of how to craft pairing functions that are particularly appropriate—mostly in terms of some notion of efficiency—for the proposed application area. We shall further augment these “applied” enrichment topics with one “pure” one: a short discussion of the computationally simplest pairing functions, the Cauchy–Cantor “diagonal” polynomials that we introduced in Section 7.1, via the specification (7.3).

8.1 Background

Because we shall be using the phrase “pairing function” so often in this chapter, we shall henceforth abbreviate the phrase—just for this chapter—by “PF.”

As we have described earlier, PFs have played a major role in a variety of studies that are “classical” within the context of computation theory. They played a pivotal role in Cantor’s seminal study of infinities [10], supplying a rigorous formal basis for asserting the counterintuitive “equinumerousness” of the integers and the rationals. It took revolutionary thinkers such as Gödel and Turing to recognize that the correspondences embodied by PFs can be viewed as encodings, or translations, of ordered pairs (and thence of arbitrary finite tuples or strings) as integers. This insight