Efficient Pairing Functions—and Why You Should Care

(Extended Abstract)

Arnold L. Rosenberg
Department of Computer Science
University of Massachusetts
Amherst, MA 01003
rsnbrg@cs.umass.edu

Abstract

This paper provides a short tour through the world of pairing functions*—bijections between* $\mathsf{N} \times \mathsf{N}$ *and* N*—as models for computational "situations." After a short discussion of the computationally simplest pairing functions—the Cauchy-Cantor "diagonal" polynomials—we describe two specific computational situations in some detail: the use of pairing functions as storage mappings for rectangular arrays/tables that can expand and shrink dynamically; the use of pairing functions as the basis for a mechanism for instilling accountability into Web-computing projects.*

1. Introduction

Entia non sunt multiplicanda praeter necessitatem

Occam's Razor

This famous admonition by William of Occam (14th cent.) to strive for simplicity is worth heeding when seeking mathematical models of computational phenomena. In this spirit, we describe examples of the use of *pairing functions* (*PF*s, for short)—bijections between $\mathsf{N} \times \mathsf{N}$ and N, where N is the set of positive integers—as the basis for such models. Our emphasis is on the varied structures that PFs can enjoy, which allow one to use PFs to model quite disparate computational situations. After due obeisance to the importance of PFs in foundational studies (in the next paragraph), we turn in Section 2 to a short discussion of the computationally simplest PFs, the Cauchy-Cantor polynomials, whose charming mathematical structure—see (2.1)—begs one to ask if other such polynomials exist. (This question remains unresolved!) In the remainder of the paper, we focus on PFs in more "modern" computational situations, discussing just two situations that I have studied personally: the use of PFs as storage mappings for rectangular arrays/tables that can expand and shrink dynamically (Section 3); the use of pairing functions as the basis for a mechanism for instilling accountability into Web-computing projects (Section 4). During our short tour, we shall encounter several intriguing and challenging problems that remain open, despite the apparent simplicity of PFs.

The "classical" importance of PFs. As is well known, PFs played a pivotal role in Cantor's seminal study [1] of infinities, 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 allowed Gödel and Turing to build on the existence of eminently computable—indeed, easily computed—PFs in their famous studies of, respectively, logical systems [5] and algorithmic systems [15]. The uses we propose for PFs, while certainly less profound than these two, also build on the insight that PFs can be used as encoding mechanisms.

Throughout the paper, we illustrate selected values from selected PFs using the following convention. We illustrate a PF $\mathcal{F} : \mathsf{N} \times \mathsf{N} \leftrightarrow \mathsf{N}$ via a two-dimensional array whose entries are the values of $\mathcal{F}$ described in Fig. 1.

$\mathcal{F}(1,1)$	$\mathcal{F}(1,2)$	$\mathcal{F}(1,3)$	$\mathcal{F}(1,4)$	$\mathcal{F}(1,5)$	$\cdots$
$\mathcal{F}(2,1)$	$\mathcal{F}(2,2)$	$\mathcal{F}(2,3)$	$\mathcal{F}(2,4)$	$\mathcal{F}(2,5)$	$\cdots$
$\mathcal{F}(3,1)$	$\mathcal{F}(3,2)$	$\mathcal{F}(3,3)$	$\mathcal{F}(3,4)$	$\mathcal{F}(3,5)$	$\cdots$
$\mathcal{F}(4,1)$	$\mathcal{F}(4,2)$	$\mathcal{F}(4,3)$	$\mathcal{F}(4,4)$	$\mathcal{F}(4,5)$	$\cdots$
$\mathcal{F}(5,1)$	$\mathcal{F}(5,2)$	$\mathcal{F}(5,3)$	$\mathcal{F}(5,4)$	$\mathcal{F}(5,5)$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Figure 1. Our generic sampling from a PF.

2. The Prettiest Pairing Function(s)

It has been known for almost two centuries that there exist bijections between $\mathsf{N} \times \mathsf{N}$ and N. Indeed, it has been known for at least 125 years that there exist such bijections that are *polynomials*. Cauchy [2] pictorially describes, and Cantor [1] symbolically specifies, the *Diagonal* PF

$$\mathcal{D}(x,y) \;=\; \binom{x+y-1}{2} + y \qquad (2.1)$$

(which, of course, has a twin obtained by exchanging x and y); see Fig. 2. A simple double induction based on the fact that $\mathcal{D}$ maps integers in an "upward direction" along the "diagonal shells," $x+y=2,\ x+y=3,\ x+y=4,\ \ldots$ proves that the function specified in (2.1) does, indeed, describe a bijection. A computationally more satisfying proof in [3] finds an explicit recipe for computing $\mathcal{D}$'s inverse.

1	3	6	10	[15]	21	28	36	$\cdots$
2	5	9	[14]	20	27	35	44	$\cdots$
4	8	[13]	19	26	34	43	53	$\cdots$
7	[12]	18	25	33	42	52	63	$\cdots$
[11]	17	24	32	41	51	62	74	$\cdots$
16	23	31	40	50	61	73	86	$\cdots$
22	30	39	49	60	72	85	99	$\cdots$
29	38	48	59	71	84	98	113	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Figure 2. The diagonal PF $\mathcal{D}$. The shell $x+y=6$ is highlighted.

Is $\mathcal{D}$ the only polynomial PF? It is so intriguing that there exists a PF that is a *polynomial* that the question of $\mathcal{D}$'s uniqueness (as a *polynomial* PF) is irresistible! This question remains largely open, but there are a few nontrivial beginnings to an answer.

1. There is no quadratic polynomial PF other than $\mathcal{D}$ (and its twin) [4].
2. The preceding assertion remains true if the "onto" condition for bijections is replaced by a "unit density" condition [7].
3. No cubic or quartic polynomial is a PF [8].
4. The development in [8] excludes large classes of higher-degree polynomials from being PFs—e.g., a super-quadratic polynomial whose coefficients are all positive cannot be a PF—yet fails to settle the problem.

3. Pairing Functions and Array/Table Storage

This section is derived from [11, 12].

It has been recognized since the 1950s that many computational scenarios, ranging from scientific applications to databases, benefit from the ability to reshape multidimensional arrays and tables dynamically. While several programming languages allow a user to specify at least some types of reshapings—say, the addition and/or deletion of rows and/or columns in two dimensions—the language processors I am aware of implement the capability quite naively, by completely remapping an array/table with each reshaping. In the mid-1970s, I began to study the use of PFs as storage-mapping functions for two-dimensional rectangular arrays/tables, since the mappings so specified would allow one to add and delete rows and columns dynamically, without ever remapping array/table positions that are unaffected by the reshaping. (Extending this work to higher dimensionalities is immediate.)

3.1. A Methodology for Constructing PFs

Rather than trying to select a single one-form-fits-all storage mapping for dynamically extendible arrays, I decided, in [11], to explore the space of possible PFs, with an eye to seeking ones that optimized various criteria. I used the following procedure to systematize the process of constructing PFs.

Procedure PF-Constructor($\mathcal{A}$)
/*Construct a PF $\mathcal{A}$*/

Step 1. Partition the set $\mathsf{N} \times \mathsf{N}$ of potential array/table positions into finite sets called *shells*. Order the shells linearly in some way: many natural shell-partitions carry a natural order.

/*Here are a few sample shell-partitions that played a role in our study of extendible arrays. For each relevant integer c, shell c comprises all pairs $\langle x, y\rangle$ such that: (a) $x + y = c$ (the diagonal shells that define the PF $\mathcal{D}$ of (2.1) and Fig. 2); (b) $\max(x, y) = c$ (square shells; cf. (3.3) and Fig. 3); (c) $xy = c$ (hyperbolic shells; cf. (3.4) and Fig. 4).*/

Step 2. Construct a PF from the shells as follows.

- **Step 2a.** Enumerate the array positions shell by shell, honoring the ordering of the shells.
- **Step 2b.** Enumerate each shell in some systematic way, say "by columns". This means enumerating the pairs $\langle x, y\rangle$ in the shell in increasing order of y and, for pairs having equal y values, in, say, decreasing order of x. (Increasing order of x works as well, of course.)

/*We have thus extended the partial order imposed by the shell structure to a linear order—which means that we have a PF $\mathcal{A}$.*/

Of course, Procedure PF-Constructor begs the question of how to compute the PF $\mathcal{A}$ efficiently. This issue will be a major concern for the remainder of this paper.

3.2. Pursuing Compact PFs

When one considers using a PF such as $\mathcal{D}$ for mapping arrays/tables into storage, one notes immediately the poor resulting management of storage. For instance, one sees in Fig. 2 that $\mathcal{D}$ spreads the 64-position 8×8 array/table over 113 addresses, and, even worse (percentage-wise), it spreads the 8-position 1×8 array/table over 36 addresses. I would argue that this loss of "compactness" is a more serious deficiency than is the loss of the bidirectional arithmetic progressions enjoyed by the ubiquitous row- or column-major indexings, since the waste of storage plagues one no matter how one intends to access the array/table. Therefore, in [11, 12], I actively sought PFs $\mathcal{A}$ that were *compact*, as measured by small growth rates of their *compactness function*

$$\mathbf{C}_{\mathcal{A}}(n) \stackrel{\text{def}}{=} \max\{\mathcal{A}(x,y) \mid xy \le n\}. \tag{3.1}$$

Here is what I discovered.

3.2.1 Favoring One Fixed Aspect Ratio. Say that one moderates one's demands on the PF $\mathcal{A}$ by focusing only on its compactness when storing arrays/tables of a fixed aspect ratio $\langle a, b\rangle$, i.e., arrays/tables whose dimensions have the form $ak \times bk$ for some k. Then one can manage storage perfectly, in the sense that there exists a PF $\mathcal{A}_{a,b}$ such that

$$\begin{aligned}\mathbf{C}_{\mathcal{A}_{a,b}}(n) &\stackrel{\text{def}}{=} \max\{\mathcal{A}_{a,b}(x,y) \mid [x \le ak] \wedge [y \le bk] \\ &\qquad\qquad \wedge [abk^2 \le n]\} \\ &= n. \end{aligned} \tag{3.2}$$

In other words, $\mathcal{A}_{a,b}$ maps every position (x,y) of an $ak \times bk$ array/table having n or fewer positions to an address $\le n$. It is easy to construct $\mathcal{A}_{a,b}$ via the shells specified as follows.

1. Shell 1 comprises the positions of the $a \times b$ array, i.e., the set
$$\{\langle x,y\rangle \mid [x \le a] \wedge [y \le b]\}.$$
2. Shell $k+1$ comprises the positions of the $a(k+1) \times b(k+1)$ array that are not elements of the $ak \times bk$ array, i.e., the set
$$\{\langle x,y\rangle \mid [ak < x \le a(k+1)] \vee [bk < y \le b(k+1)]\}.$$

While the preceding guarantee of compactness ignores the issue of ease of computation, there are at least some PFs that utilize storage perfectly in the sense of (3.2) and are quite easy to compute. One useful such PF favors *square* arrays/tables, i.e., those whose aspect ratio is given by $a = b = 1$. This "square-shell" PF, $\mathcal{A}_{1,1}$, which seems to have originated in [11], is specified by the following explicit expression.

$$\begin{aligned}\mathcal{A}_{1,1}(x,y) &= m^2 + m + y - x + 1 \\ \text{where} \quad m &\stackrel{\text{def}}{=} \max(x-1, y-1).\end{aligned} \tag{3.3}$$

Once having noted that $\mathcal{A}_{1,1}$ maps integers in a counter-clockwise direction along the "square shells" $m = 0,\ m = 1, \ldots$ (see Fig. 3) one verifies its bijectiveness via a simple double induction. (Of course, $\mathcal{A}_{1,1}$ has a twin that proceeds in a clockwise direction along the square shells.)

1	4	9	16	25	36	49	64	$\cdots$
2	3	8	15	24	35	48	63	$\cdots$
5	6	7	14	23	34	47	62	$\cdots$
10	11	12	13	22	33	46	61	$\cdots$
17	18	19	20	21	32	45	60	$\cdots$
26	27	28	29	30	31	44	59	$\cdots$
37	38	39	40	41	42	43	58	$\cdots$
50	51	52	53	54	55	56	57	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Figure 3. The square-shell PF $\mathcal{A}_{1,1}$. The shell $\max(x,y) = 5$ is highlighted.

3.2.2 Favoring Finite Sets of Aspect Ratios. We show in [12] how to "dovetail" any set $\{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_m\}$ of m PFs to arrive at a PF $\mathcal{A}$ whose compactness is at worst m times that of the most compact $\mathcal{A}_i$; i.e., for all n,

$$\mathbf{C}_{\mathcal{A}}(n) \le m \cdot \min_i \mathbf{C}_{\mathcal{A}_i}(n).$$

The dovetailing is performed in two steps.

1. Alter each $\mathcal{A}_k$ to be a bijection $\mathcal{A}_k^{(m)}$ between $\mathsf{N} \times \mathsf{N}$ and the congruence class $k-1 \bmod m$, i.e., the set of integers of the form $mx + k - 1$. Specifically, define the PF $\mathcal{A}_k^{(m)}$ via: $\mathcal{A}_k^{(m)}(x,y) = m \cdot \mathcal{A}_k(x,y) + k - 1$.
2. Define the PF $\mathcal{A}$ via: $\mathcal{A}(x,y) = \min_k\{\mathcal{A}_k^{(m)}(x,y)\}$.

Thus, if one wants a PF to be compact on arrays of aspect ratios $\langle a_1,\ b_1\rangle,\ \langle a_2,\ b_2\rangle, \ldots, \langle a_m,\ b_m\rangle$, then one can craft a PF $\mathcal{A}_{a_1,b_1;a_2,b_2;\ldots;a_m,b_m}$ that maps every position (x,y) of an array/table which has one of these m aspect ratios, and which has n or fewer positions, to an address $\le mn$.

3.2.3 Maximizing Worst-Case Compactness. The preceding shape-based guarantees do not help much with applications such as relational databases, wherein one cannot limit *a priori* the potential shapes of one's tables. This fact led me to the question of how compact a PF could be on arrays/tables of arbitrary shapes. We show in [12] that there exists a PF $\mathcal{H}$ whose compactness is given by

$$\mathbf{C}_{\mathcal{H}}(n) \;=\; O(n \log n),$$

and no PF can beat this compactness (in the worst case) by more than a constant factor. The PF $\mathcal{H}$ maps integers along the "hyperbolic shells" defined by $xy = 1$, $xy = 2$, $xy = 3, \ldots$. More specifically, if we let $\delta(n)$ denote the number of divisors of the integer n, then

$$\begin{aligned} \mathcal{H}(x,y) \;\;=\;\; & \sum_{k=1}^{xy-1} \delta(k) \\ & +\; \text{the position of } \langle x,y \rangle \text{ among} \\ & \quad \text{2-part factorizations of } xy, \text{ in} \\ & \quad \text{reverse lexicographic order} \end{aligned} \tag{3.4}$$

See Fig. 4.

1	3	5	8	10	[14]	16	$\cdots$
2	7	[13]	19	26	34	40	$\cdots$
4	[12]	22	33	44	56	69	$\cdots$
6	18	32	48	64	81	99	$\cdots$
9	25	43	63	86	108	130	$\cdots$
[11]	31	55	80	107	136	165	$\cdots$
15	39	68	98	129	164	200	$\cdots$
17	47	79	116	154	193	235	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Figure 4. The hyperbolic PF $\mathcal{H}$. The shell $xy = 6$ is highlighted.

4. Pairing Functions and Web-Computing

This section is derived from [13].

Evolving technology has given rise to a new modality of cooperative computing, which we call *Web-Based Computing* (*WBC*, for short). WBC proceeds roughly as follows. "Volunteers" register with a WBC website. After having registered, each volunteer visits the website from time to time to receive a task to compute. Some time after completing a task, the volunteer returns the results from that task and receives a new task. And the cycle continues.

As typically implemented, WBC is vulnerable to malicious, or careless, volunteers returning false results. When the WBC computations relate to sensitive matters such as security [14] or medical testing [6, 10], such false results could have dire consequences. In [13], we have proposed a computationally lightweight scheme for keeping track of which volunteer computed which task(s), thereby enabling the head of the WBC project to ban frequently errant volunteers from continued participation in the project. The proposed scheme builds on the strategy of assigning positive-integer indices to (a) the set of all tasks, (b) all volunteers, (c) the set of tasks reserved for each volunteer v, and using a PF $\mathcal{T}$ (which we call a *task-allocation function*) to link volunteers with their assigned tasks. In other words, the tth task that volunteer v receives to compute is task $\mathcal{T}(v,t)$. Since the practicality of such a scheme demands that $\mathcal{T}$, $\mathcal{T}^{-1}$, and $\mathcal{S}(v,t) \stackrel{\text{def}}{=} \mathcal{T}(v,t+1) - \mathcal{T}(v,t)$ all be easily computed, the primary focus in [13] is on PFs that are *additive* (*APFs*, for short): an APF assigns each volunteer v a *base task-index* B_v and a *stride* S_v; it then uses the formula $\mathcal{T}(v,t) = B_v + (t-1)S_v$ to determine the workload task-index of the tth task assigned to volunteer v. From a system perspective, APFs have the benefit that a volunteer's stride need be computed only when s/he registers at the website and can be stored for subsequent appearances.

The complete scheme described in [13] has a "front end" which ensures that, even when volunteers may arrive and depart dynamically, faster volunteers are always assigned smaller indices. Given our focus on APFs, and given this mode of assigning indices to volunteers, one can argue that the management of the memory where tasks reside is simplified if we devise APFs whose strides S_v grow slowly as a function of v, hence, that are compact in the sense of (3.1). This observation sets the agenda for [13] and for the remainder of this section. In Section 4.1, we present a methodology for designing easily computed APFs; in Section 4.2, we present a sequence of APFs that represent a tradeoff between ease of computation and slowness of stride growth.

We henceforth abstract the preceding discussion from the WBC scenario by replacing "volunteer" by "row" and "base task-index" by "base row-entry." We also revert to our generic uses of x and y, instead of v and t.

4.1. A Methodology for Designing *Additive* PFs

Easily, any APF must have infinitely many distinct strides; i.e., S_x, viewed as a function of x, must have infinite range. Despite this, we now show that there do exist easily computed APFs. Our strategy for designing such APFs builds on the following well-known property of the set O of positive odd integers.

Lemma 4.1 ([9]) *For any positive integer c, every odd integer can be written in precisely one of the 2^{c-1} forms:*

$$2^c n + 1,\; 2^c n + 3,\; 2^c n + 5, \ldots,\; 2^c n + (2^c - 1),$$

for some nonnegative integer n.

We build on Lemma 4.1 to construct APFs as follows.

Procedure APF-Constructor($\mathcal{T}$)
/*Construct an APF $\mathcal{T}$*/

Step 1. Partition the set of row-indices into *groups* whose sizes are powers of 2 (with any desired mix of equal-size and distinct-size groups). Order the groups linearly in some (arbitrary) way.

/*We can now talk unambiguously about group 0 (whose members share *group-index* $g = 0$), group 1 (whose members share group-index $g = 1$), and so on.*/

Step 2. Assign each group a distinct copy of the set O, via a *copy-index* $\kappa(g)$ expressed as a function of the group-index g.

/*We can now talk unambiguously about group g's copy $\mathsf{O}_{\kappa(g)}$ of the odd integers.*/

Step 3. Allocate group g's copy $\mathsf{O}_{\kappa(g)}$ to its members via the $(c = \kappa(g))$ instance of Lemma 4.1, using the multiplier 2^g as a *signature* to distinguish group g's copy of the set O from all other groups' copies.

An explicit expression for $\mathcal{T}$. If we denote the $2^{\kappa(g)}$ rows of group g as $x_{g,1}, x_{g,2}, \ldots, x_{g,2^{\kappa(g)}}$, then for all $i \in \{1, 2, \ldots, \kappa(g)\}$,

$$\mathcal{T}(x_{g,i}, y) \stackrel{\text{def}}{=}$$

$$2^g \left[2^{1+\kappa(g)}(y-1) + (2x_{g,i} + 1 \bmod 2^{1+\kappa(g)})\right] \quad (4.1)$$

Theorem 4.1 *Any function $\mathcal{T} : \mathsf{N} \times \mathsf{N} \leftrightarrow \mathsf{N}$ that is designed via Procedure* APF-Constructor*, hence is of the form (4.1), is a valid APF whose base row-entries and strides satisfy*

$$\begin{array}{rcl} B_x & \leq & 2^{1+g+\kappa(g)} \\ S_x & = & \mathcal{T}(x, y+1) - \mathcal{T}(x, y) \;=\; 2^{1+g+\kappa(g)}. \end{array} \quad (4.2)$$

Proof sketch. **(1)** Any such $\mathcal{T}$ maps $\mathsf{N} \times \mathsf{N}$ *onto* N, because every positive integer equals *some* power of 2 times *some* odd integer. $\mathcal{T}$ is *one-to-one* because it has a functional inverse $\mathcal{T}^{-1}$. To wit, the trailing 0's of each image integer $k = \mathcal{T}(x, y)$ identify x's group g, hence the operative instance $\kappa(g)$ of Lemma 4.1. Then:

1. We compute

$$x \;=\; \frac{1}{2}\left[(2^{-g}k \bmod 2^{1+\kappa(g)}) - 1\right],$$

which is an integer because the division by 2^g produces an odd number.

2. We end up with a linear expression of the form $ay + b$, from which we easily compute y.

Finally, we read the relations (4.2) directly from (4.1). ■

In order to implement Procedure APF-Constructor completely, one must express both the group-indices g and their associated copy-indices $\kappa(g)$ as functions of x. This is accomplished by noting that all x whose indices lie in the range

$$\begin{array}{l} 2^{\kappa(0)} + 2^{\kappa(1)} + \cdots + 2^{\kappa(g-1)} + 1 \;\leq\; x \\ \qquad \leq\; 2^{\kappa(0)} + 2^{\kappa(1)} + \cdots + 2^{\kappa(g-1)} + 2^{\kappa(g)} \end{array} \quad (4.3)$$

share group-index g and copy-index $\kappa(g)$. Translating the range (4.3) into an efficiently computed expression of the form $g = f(x)$ may be a simple or a challenging enterprise, depending on the functional form of $\kappa(g)$ that results from the grouping of row-indices.

4.2. A Sampler of Explicit APFs

Theorem 4.1 assures us that Procedure APF-Constructor produces a valid APF no matter how the copy-index $\kappa(g)$ grows as a function of the group-index g. However, the ease of computing the resulting APF, and its compactness, depend crucially on this growth rate. We now illustrate how one can use this growth rate as part of the design process, in order to stress either the ease of computing an APF or its compactness.

4.2.1 APFs that Stress Computation Ease. We first implement Procedure APF-Constructor with *equal-size groups*, i.e., with $\kappa(g) = constant$. For each $c \in \mathsf{N}$, let $\mathcal{T}^{<c>}$ be the APF produced by the Procedure with $\kappa^{<c>}(g) \equiv c - 1$. One computes easily that

$$\mathcal{T}^{<c>}(x, y) \stackrel{\text{def}}{=}$$

$$2^{\lfloor (x-1)/2^{c-1} \rfloor}\left[2^c(y-1) + (2x - 1 \bmod 2^c)\right].$$

Proposition 4.1 *Each $\mathcal{T}^{<c>}$ is a valid APF whose base row-entries and strides are given by*

$$B_x^{<c>} \;\leq\; S_x^{<c>} \;=\; 2^{\lfloor (x-1)/2^{c-1} \rfloor + c}. \quad (4.4)$$

Each $\mathcal{T}^{<c>}$ is easy to compute but has base row-entries and strides that grow *exponentially* with row-indices. Increased values of c (= larger fixed group sizes) decrease the base of the growth exponential, at the expense of modest increase in computational complexity. Computing a few sample values illustrates how a larger value of c penalizes a few low-index rows but gives all others significantly smaller base row-entries and strides; cf. Fig. 5.

$\langle x, g\rangle$	$\mathcal{T}^{<1>}(x,y)$					
$\langle 14, 13\rangle$	8192	24576	40960	57344	73728	$\cdots$
$\langle 15, 14\rangle$	16384	49152	81920	114688	147456	$\cdots$
$\langle x, g\rangle$	$\mathcal{T}^{<3>}(x,y)$					
$\langle 14, 3\rangle$	24	88	152	216	280	$\cdots$
$\langle 15, 3\rangle$	40	104	168	232	296	$\cdots$
	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$\langle 28, 6\rangle$	448	960	1472	1984	2496	$\cdots$
$\langle 29, 7\rangle$	128	1152	2176	3200	4224	$\cdots$
$\langle x, g\rangle$	$\mathcal{T}^{\#}(x,y)$					
$\langle 28, 4\rangle$	400	912	1424	1936	2448	$\cdots$
$\langle 29, 4\rangle$	432	944	1456	1968	2480	$\cdots$
$\langle x, g\rangle$	$\mathcal{T}^{\star}(x,y)$					
$\langle 28, 3\rangle$	328	840	1352	1864	2376	$\cdots$
$\langle 29, 3\rangle$	344	856	1368	1880	2392	$\cdots$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Figure 5. Sample values by several APFs.

4.2.2 APFs that Balance Computation Ease and Compactness. The functional form of the exponent of 2 in (4.4) suggests that one can craft an APF whose base row-entries and strides grow subexponentially by allowing the parameter c to grow with x, in a way that balances the (common) growth rate of B_x and S_x. This strategy leads us to the copy-index $\kappa^{\#}(g) = g$. When we implements Procedure APF-Constructor with $\kappa^{\#}$, we arrive at an APF $\mathcal{T}^{\#}$ that is rather easy to compute and whose base row-entries and strides grow only *quadratically* with row-indices.

The copy-index $\kappa^{\#}(g) = g$ aggregates row-indices into groups of exponentially growing sizes: each group g comprises row-indices $2^g, 2^g + 1, \ldots, 2^{g+1} - 1$. By (4.3), then, one computes easily that[1]

$$\kappa^{\#}(g) \;=\; g \;=\; \lfloor \log x \rfloor. \tag{4.5}$$

Instantiating (4.5) in the definitional scheme (4.1), we find that

$$\mathcal{T}^{\#}(x,y) \;=$$

$$2^{\lfloor \log x \rfloor}\left(2^{1+\lfloor \log x \rfloor}(y-1) + (2x+1 \bmod 2^{1+\lfloor \log x \rfloor})\right) \tag{4.6}$$

Proposition 4.2 *The function $\mathcal{T}^{\#}$ specified in (4.6) is a valid APF whose base row-entries and strides (as functions of x) are given by*

$$B_x^{\#} \;<\; \mathcal{S}_x^{\#} \;=\; 2^{1+2\lfloor \log x \rfloor} \;\leq\; 2x^2,$$

hence, grow quadratically with x.

[1]Throughout, all logarithms have base 2.

Comparing $\mathcal{T}^{\#}$ and the $\mathcal{T}^{<c>}$. For sufficiently large x, the (exponentially growing) strides of any of the APFs $\mathcal{T}^{<c>}$ will be dramatically larger than the (quadratically growing) strides of the APF $\mathcal{T}^{\#}$. However, it takes a while for $\mathcal{T}^{\#}$'s superiority to manifest itself; for instance,

- it is not until $x = 5$ that $\mathcal{T}^{<1>}$'s strides are always at least as large as $\mathcal{T}^{\#}$'s;
- the corresponding number for $\mathcal{T}^{<2>}$ is $x = 11$;
- the corresponding number for $\mathcal{T}^{<3>}$ is $x = 25$.

4.2.3 APFs that Stress Compactness. By choosing a copy-index $\kappa(g)$ that grows superlinearly with g, one can craft APFs whose base row-entries and strides grow subquadratically, thereby beating the compactness of $\mathcal{T}^{\#}$. But one must choose $\kappa(g)$'s growth rate judiciously, since faster growth need not yield more compactness.

Achieving subquadratic growth. Many copy-index growth rates yield APFs with subquadratic compactness. However, all of the APFs we know of that achieve this goal are rather difficult to compute and actually achieve the goal only asymptotically, hence are likely more of academic than practical interest.

Consider, for each $k \in \mathsf{N}$, the APF $\mathcal{T}^{[k]}$ specified by the copy-index $\kappa^{[k]}(g) = g^k$. By (4.3), the row-indices x belonging to group g now lie in the range

$$1 + 2 + 2^4 + \cdots + 2^{(g-1)^k} \;<\; x \;\leq\; 1 + 2 + 2^4 + \cdots + 2^{g^k},$$

so that $g = (1 + o(1))\lceil (\log x)^{1/k} \rceil$. We actually use the simplified, albeit slightly inaccurate, expression $g = \lceil (\log x)^{1/k} \rceil$ in our asymptotic analyses of the $\mathcal{T}^{[k]}$, since the $o(1)$-quantity decreases very rapidly with growing x. Although closed-form expressions for $\mathcal{T}^{[k]}$ in terms of x have eluded us, we can verify that each $\mathcal{T}^{[k]}$ does indeed enjoy subquadratic stride growth.

Proposition 4.3 *Each function $\mathcal{T}^{[k]}$ produced by Procedure* APF-Constructor *from the copy-index $\kappa^{[k]}(g) = g^k$ is a valid APF whose base row-entries and strides (as functions of x) are given by*

$$\begin{aligned} B_x^{[k]} \;\leq\; \mathcal{S}_x^{[k]} \;&=\; 2^{O\left((\log x)^{1/k} + \log x\right)} \\ &=\; x 2^{O\left((\log x)^{1/k}\right)} \end{aligned} \tag{4.7}$$

hence, grow subquadratically with x.

We illustrate a close relative of $\mathcal{T}^{[2]}$ which exhibits its subquadratic compactness at much smaller values of x than $\mathcal{T}^{[2]}$ does, namely, the APF $\mathcal{T}^{\star}$ that Procedure APF-Constructor produces from the copy-index

$$\kappa^{\star}(g) = \left\lceil \frac{1}{2} g^2 \right\rceil. \tag{4.8}$$

Mimicking the development with $\kappa^{[k]}$, we see that now $g = (1 + o(1))\lceil\sqrt{2\log x}\rceil + 1$, which we simplify for analysis to the slightly inaccurate expression

$$g \;=\; \left\lceil \sqrt{2\log x} \right\rceil + 1. \tag{4.9}$$

We can easily compute $\mathcal{T}^\star$ from (4.8), in the presence of (4.1, 4.3).

Proposition 4.4 *The base row-entries and strides of the APF $\mathcal{T}^\star$ satisfy*

$$B^\star_x \;\le\; \mathcal{S}^\star_x \;=\; 2^{1+g+\kappa^\star(g)} \;\approx\; 8x4^{\sqrt{2\log x}}.$$

Comparing $\mathcal{T}^\star$ and $\mathcal{T}^\#$. Any function that grows quadratically with x will eventually produce significantly larger values than a function that grows only as $x4^{\sqrt{2\log x}}$. Therefore, $\mathcal{T}^\star$'s strides will eventually be dramatically smaller than $\mathcal{T}^\#$'s. Fig. 5 indicates that this difference takes effect at about the same point as the exponential vs. quadratic one noted earlier, albeit at the cost of greater computational complexity.

The danger of excessively fast growing κ. If $\kappa(g)$ grows too fast with g, then the base row-entries and strides of the resulting APF grow *super*quadratically with the row-indices x, thereby confuting our goal of beating quadratic growth. We exemplify this fact by supplying Procedure APF-Constructor with the copy-index $\kappa(g) = 2^g$; the reader can readily supply other examples. By (4.3), we see that in this case, $g = \lfloor \log\log x \rfloor + O(1)$. Therefore, whenever x is the smallest row-index with a given group-index g (of course, infinitely many such x exist) we have

$$x \;=\; 2^{\kappa(0)} + 2^{\kappa(1)} + \cdots + 2^{\kappa(g-1)} + 1 \;\approx\; \sqrt{2^{\kappa(g)}},$$

while the stride associated with x is (cf. (4.2))

$$S_x \;=\; 2^{1+g+\kappa(g)} \;>\; 2^{\kappa(g)}\kappa(g) \;\approx\; x^2 \log x.$$

We do not yet know the growth rate at which faster growing $\kappa(g)$ starts hurting compactness.

Acknowledgments. The research described in Sections 2 and 3 was done while the author was with the IBM Watson Research Center. The research described in Section 4, and the preparation of this paper were supported in part by NSF Grant CCR-00-73401.

References

[1] G. Cantor (1878): Ein Beitrag zur Begrundung der transfiniter Mengenlehre. *J. Reine Angew. Math. 84*, 242–258.

[2] A.L. Cauchy (1821): *Cours d'analyse de lÉcole Royale Polytechnique, 1ère partie: Analyse algébrique*. l'Imprimerie Royale, Paris. Reprinted: Wissenschaftliche Buchgesellschaft, Darmstadt, 1968.

[3] M. Davis (1958): *Computability and Unsolvability*. McGraw-Hill, N.Y.

[4] R. Fueter and G. Pólya (1923): Rationale Abzählung der Gitterpunkte. *Vierteljschr. Naturforsch. Ges. Zürich 58*, 380–386.

[5] K. Gödel (1931): Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I. *Monatshefte für Mathematik und Physik 38*, 173–198.

[6] *The Intel Philanthropic Peer-to-Peer program.* ⟨`www.intel.com/cure`⟩.

[7] J.S. Lew and A.L. Rosenberg (1978): Polynomial indexing of integer lattices, I. *J. Number Th. 10*, 192–214.

[8] J.S. Lew and A.L. Rosenberg (1978): Polynomial indexing of integer lattices, II. *J. Number Th. 10*, 215–243.

[9] I. Niven and H.S. Zuckerman (1980): *An Introduction to the Theory of Numbers.* (4th Ed.) J. Wiley & Sons, New York.

[10] *The Olson Laboratory Fight AIDS@Home project.* ⟨`www.fightaidsathome.org`⟩.

[11] A.L. Rosenberg (1974): Allocating storage for extendible arrays. *J. ACM 21*, 652–670.

[12] A.L. Rosenberg (1975): Managing storage for extendible arrays. *SIAM J. Comput. 4*, 287–306.

[13] A.L. Rosenberg (2002): Accountable Web-computing. Submitetd for publication. See an extended abstract in *IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS'02)*.

[14] *The RSA Factoring by Web Project.* ⟨`http://www.npac.syr.edu/factoring`⟩ (with Foreword by A. Lenstra). Northeast Parallel Architecture Center.

[15] A.M. Turing (1936): On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.* (ser. 2, vol. 42) 230–265; Correction *ibid.* (vol. 43) 544–546.