Recursive Hashing Functions for *n*-Grams

JONATHAN D. COHEN
National Security Agency

Many indexing, retrieval, and comparison methods are based on counting or cataloguing n-grams in streams of symbols. The fastest method of implementing such operations is through the use of hash tables. Rapid hashing of consecutive n-grams is best done using a recursive hash function, in which the hash value of the current n-gram is derived from the hash value of its predecessor. This article generalizes recursive hash functions found in the literature and proposes new methods offering superior performance. Experimental results demonstrate substantial speed improvement over conventional approaches, while retaining near-ideal hash value distribution.

Categories and Subject Descriptors: E.2 [**Data**]: Data Storage Representations—*hash-table representations*; G.2.1 [**Discrete Mathematics**]: Combinatorics—*recurrences and difference equations*; G.3 [**Mathematics of Computing**]: Probability And Statistics—*probabilistic algorithms*; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*indexing methods*; H.3.3 [**Information Storage and Retrieval**]: Information Storage; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms: Algorithms, Experimentation, Theory

Additional Key Words and Phrases: Hashing, hashing functions, n-grams, recursive hashing

1. INTRODUCTION

This article concerns itself with the rapid cataloguing, counting, or retrieval of n-grams. As such, it is necessarily about quickly recording or retrieving information. This section first introduces n-grams, then discusses hashing (the most rapid method of information storage and retrieval), then combines the two.

1.1 *n*-Grams

Given a sequence of symbols $S = (s_1, s_2, \ldots, s_{N+(n-1)})$, an n-gram of the sequence is an n-long subsequence of consecutive symbols. The ith n-gram of S is the sequence $(s_i, s_{i+1}, \ldots, s_{i+n-1})$. Note that there are N such n-grams in S.

Author's address: National Security Agency, 9800 Savage Road, Fort Meade, MD 20755-6000; email: jdcohen@afterlife.ncsc.mil.
Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 1997 ACM 1046-8188/97/0700–0291 $03.50

The literature discloses a wide variety of n-gram applications. When the symbols are letters of text, n-grams have been used for text compression to save space and accelerate searches [Shannon 1951; Schuegraf and Heaps 1973; Barton et al. 1974; Lynch 1977; Wisniewski 1987], language recognition [Schmitt 1990; Damashek 1995], topic recognition and abstracting [Damashek 1995; Cohen 1995], spelling error detection and correction [McElwain and Evens 1962; Morris and Cherry 1975; Zamora et al. 1981; Angell et al. 1983], optical character recognition [Vossler and Branston 1964; Thomas and Kassler 1967; Cornew 1968; Hussain and Donaldson 1974; Neuhoff 1975; Hanson et al. 1976; Shinghal et al. 1978; Hull and Srihari 1982], string searching [Harrison 1971; Karp and Rabin 1987; Gonnet and Baeza-Yates 1990; Kotamarti and Tharp 1990], approximate string matching [Ukkonen 1992; Kim and Shawe-Taylor 1992], typing prediction [Darragh et al. 1990], and information retrieval [Burnett et al. 1979; Willet 1979; de Heer 1982; D'Amore and Mah 1985; Cavnar 1993; Pearce 1994; Damashek 1995].

When the symbols are words, n-grams have been exploited for word recognition in speech [Paeseler and Ney 1989; Pietra et al. 1992; Wright et al. 1992] and word categorization [Nakamura and Shikano 1989].

Sequences of phoneme symbols have been analyzed by n-grams to guess missing phonemes [Yannakoudakis and Hutton 1992].

Other n-gram references and applications may be found in the reviews of Suen [1979] and Kukich [1992].

For ease of discussion, this article will assume that the symbols represent characters, though no limitation will result from this assumption. The symbol sequence S will represent a text "document." It will also be assumed that one is interested in examining all n-grams in S for some n.

1.2 Hashing

The central concern of this article is the storage and retrieval of information about a document's n-grams. As in other information systems, the information is represented as a collection of records, and each record is identified by a unique key. In this case a record's key is an n-gram.

Of the many record storage and retrieval schemes, the fastest method is to build a table in memory consisting of sequentially addressed "buckets" and use each record key to directly identify the bucket address for the corresponding record. Suppose that the keys are drawn from a universe U and the table has B buckets. Then as long as $|U| \leq B$, the keys can be mapped directly to bucket addresses in some prearranged manner.

If $|U| > B$, but the number of keys that will actually occur in the records at hand does not exceed B, one may choose to map the keys from U to the bucket addresses using a hash function h. Hash functions, as the name suggests, are designed to pseudorandomly map a large domain to a smaller

range;[1] in this case they map U to the integers 0, 1, . . . , $B - 1$. A record with key k is mapped to address $h(k)$; k is said to be hashed to $h(k)$. The table addressed by a hash function is known as a hash table.[2] If the hash function is chosen well, and the data are cooperative, the likelihood that two distinct keys will map to the same address is small. In the event that such collisions do occur, various methods are available to resolve them, at little added expense.

Even if the number of keys exceeds the number of available buckets, this scheme is still effective: each key is hashed to an address in the hash table, which then serves as the entry point into a data structure capable of holding the multiple records of differing keys that hashed to that position.

The speed of addressing into the table does not depend on the size of the table, but only on the speed of the hash function calculation. On the other hand, the time needed to catalogue multiple records hashed to a single table bucket (by collision handling) increases at least linearly with the number catalogued there, so it happens that collision resolution time, averaged over all buckets, is minimized by choosing a hash function that is most uniform in its distribution over bucket addresses.

The literature of hash functions and collision resolution schemes is extensive. A tutorial is offered by Cormen et al. [1990]. More detailed analysis and an overview of early activity can be found in the classic work by Knuth [1973]. Comprehensive citations to the literature and detailed algorithms have been compiled by Gonnet and Baeza-Yates [1991]. A performance comparison of several hashing techniques is offered by Lum et al. [1971].

1.3 Hashing of n-Grams

Some of the most effective uses of character n-grams have relied on counting the various n-grams in a document under examination. (See, for example, D'Amore and Mah [1985], Pearce [1994], Damashek [1995], and Cohen [1995].) For even modest n, the possible number of such n-grams can be enormous, but the number actually encountered in a text document is relatively small. (A typical number of unique five-grams present in a large English-language document is 2×10^5, less than 2% of the five-gram universe.) This n-gram counting is most efficiently implemented by a hash table: each n-gram is directly taken as a key, is hashed to a table address, and (after possible collision resolution) an accumulator at that address is advanced. For such uses, which are often interactive, speed of counting (and therefore hashing) is of paramount importance.

As a motivating example, Damashek [1995] uses n-gram counting for interactive information categorization and retrieval. In his method, each document is represented by an n-gram spectrum, that is, by a vector

[1]The literature discusses another type of hash function designed to encrypt its input, which is used for tasks such as password protection. Such functions will not be considered here.

[2]The term "scatter table" has also been used, but is largely obsolete. In early literature, hash functions were known as "key-to-address transformations."

containing the counts of every n-gram present in the document; n is chosen in advance. The resulting vectors are compared to recognize similarity among the respective documents. Damashek uses a hash table to form and record the n-gram counts, and the table itself serves as the n-gram vector. In practice, collisions in his counting may be ignored, so that the table need not carry any information other than the counts. The work needed to form a representation of the document is dominated by the hashing operation.

The use of overlapping n-grams as keys offers an unusual opportunity for rapid hash function calculation. Assuming that the n-grams are processed in order of occurrence, each n-gram is similar to its predecessor, and this redundancy can be exploited. Consider hashing of the ith n-gram $S_i = (s_i, s_{i+1}, \ldots, s_{i+n-1})$. For a good hash function, the hash value $h(S_i)$, should depend on every one of the n symbols and should treat each of these symbols differently. But $n - 1$ of those same symbols were used to calculate $h(S_{i-1})$: the new n-gram only introduces s_{i+n-1}, drops s_{i-1}, and "slides" the others over. A natural question is whether there is a form of h that would permit more efficient *recursive* calculation of the hash function, that is, whether one could write

$$h(S_i) = f(h(S_{i-1}), s_{i+n-1}, s_{i-1}),$$

for some function f and get a savings of time.

Such a recursive approach has been described by Karp and Rabin [1987] and Gonnet and Baeza-Yates [1990]. This article presents a generalization of the earlier work to arbitrary linear recursion, offers several new linearly recursive hash functions, and evaluates their performance. (The issue of collision handling will not be addressed directly.) Results of performance experiments are presented, verifying that recursive hashing of every n-gram in a sequence can be achieved at a speed that is independent of n and well in excess of the speed realized by conventional hashing and that use of these hash functions does not come at the expense of uniformity.

The primary contributions of the present work are the formulation of a general framework of linear recursion for hashing n-grams, the introduction of new linearly recursive methods that are faster than previous approaches, and the validation of their performance in an experiment.

A few concepts from abstract algebra are used as motivation in succeeding sections. Readers uncomfortable with such discussions may consult the Appendix.

2. LINEARLY RECURSIVE HASH FUNCTIONS

This section proposes a general form of linearly recursive hash functions. Subsequent sections give specific realizations.

Before beginning, the problem will be relaxed slightly. The hash function will be decomposed into a truly recursive function followed by a nearly trivial "address" function that maps the result of the recursive part into the

appropriate address format. The recursive part will be denoted by H and will obey

$$H(S_i) = f(H(S_{i-1}), s_{i+n-1}, s_{i-1}).$$

The total hash function will then be

$$h(S_i) = A[H(S_i)],$$

where the address function A produces a result confined to the integers $\{0, 1, \ldots, B - 1\}$. The address function will either be trivial or extremely simple.

The recursive calculation of $H(S_i)$ must introduce the new symbol s_{i+n-1} and drop the old symbol s_{i-1}. The idea behind choosing a linear recursion is that the contribution of each symbol will be independent of contributions by other symbols, so that the influence of the old symbol will be known and can be removed n steps later.

A function that is amenable to easy linear recursion is

$$H(S_i) = \sum_{j=0}^{n-1} r^{n-j-1} T(s_{i+j}), \tag{1}$$

where computation is over some ring R; $r \in R$ is some constant; and T maps the symbols into R. The constant r will be referred to as the radix. The transformation T may be trivial, since the symbols $s_1, s_2, \ldots, s_{N+(n-1)}$ can be interpreted mathematically in natural ways. For example, if computation is being done over the integers, the symbols (usually bytes) have obvious ordinal values. Other transformations will be nearly as simple. The address function A will serve to map from R to the integers $\{0, 1, \ldots, B - 1\}$.

A recursive formulation of Eq. (1) is

$$H(S_1) = \sum_{i=1}^{n} r^{n-i} T(s_i)$$

$$H(S_i) = rH(S_{i-1}) + T(s_{i+n-1}) - r^n T(s_{i-1}), \quad 1 < i \leq N. \tag{2}$$

This is the proposed form of H for hashing successive n-grams. By varying ring R, transformation T, and radix r, this form encompasses a surprising variety of hashing methods, some of which will be described below.

The composite function $r^n T(s)$ in Eq. (2) may be combined into a single function $T'(s) = r^n T(s)$. It will be convenient to do so in a few of the descriptions below.

An observation about the choice of ring is appropriate. A common cause of nonuniformity in hashed values can be traced to a bias in the frequency

of keys that are regularly spaced, that is, by keys separated by multiples of some value in the ring. (As an integer example, keys of odd value may predominate over keys of even value.) Such biases present a problem only when the hash function preserves them. This can be avoided by choosing an appropriate ring. As discussed in the Appendix (Section A.5), it is sufficient that the ring be an integral domain[3] for the methods outlined below; a field is desirable.

For any choice of implementation, the questions to be answered will be

(1) How is the computation carried out?
(2) Is the computation fast?
(3) Does the hash function yield a nearly uniform distribution?

In the succeeding sections various methods will be presented, and for each method the questions above will be addressed.

2.1 Hashing by Prime Integer Division

One standard, venerable, and highly regarded hash method is hashing by (integer) division. (The first public description of hashing, offered by Dumey in 1956, describes this approach.) In this method, the key, viewed as an integer, is reduced modulo the number of table buckets B. When accommodating textual keys, the integer representation is formed (either explicitly or implicitly) by treating each character as a digit in a radix r number, for some r. The radix is at least as large as the number of characters in the alphabet. For hashing the ith n-gram using this method, one might choose the numeric key

$$k_i = \sum_{j=0}^{n-1} r^{n-j-1}\mathrm{Ord}(s_{i+j}),$$

summed over the integers, where Ord(·) returns the ordinal value of the symbol in its argument. Note that if one is using characters as they are usually represented in a computer—one character to a byte—then a typical string is already in this form, with $r = 256$. To minimize the size of the key, one usually chooses to use a smaller radix, equal to the number of letters in the alphabet.

If this sum is taken instead over the ring $R = \mathbf{Z}/B$, that is, the integers modulo B, then the sum above *is* the integer division hash value:

$$h(S_i) = \sum_{j=0}^{n-1} r^{n-j-1}\mathrm{Ord}(s_{i+j}).$$

[3]See Appendix Section A.1 for a definition of integral domain.

But this is in the form of Eq. (1) and may be implemented recursively, according to Eq. (2), by

$$h(S_1) = \sum_{i=1}^{n} r^{n-i}\mathrm{Ord}(s_i)$$

$$h(S_i) = rh(S_{i-1}) + \mathrm{Ord}(s_{i+n-1}) - r^n\mathrm{Ord}(s_{i-1}), \quad 1 < i \leq N.$$

In this case, the address function is the identity map and has been omitted.[4] Thus, a standard nonrecursive method of hashing strings may be implemented recursively for hashing n-grams efficiently.[5]

In keeping with the observation that the ring should be an integral domain, B should be a prime. To match usual table sizes, one is likely to pick B to be the largest prime smaller than some power of 2.

Some nonrecursive hashing functions of this nature in current use are reviewed and evaluated in McKenzie et al. [1990]. Recursive hashing by prime integer division was first described by Karp and Rabin [1987] as part of a string-matching algorithm.

An implementation of recursive hashing by integer division is illustrated in Table I. The first n-gram is hashed directly; the remainder are recursively hashed. Note that execution time of the main loop will not depend upon n.

Two cautions are in order here. First, one must take care that none of the computations overflow. If the word sizes are insufficient or the radix too large to guarantee that overflow will not occur, then tests and corrections to avoid overflow or additional modulo functions must be performed. Second, the modulo function must return positive values. Some computer implementations of the MOD function produce negative results for negative arguments. This can be corrected by checking for a negative result and adding the modulus if necessary.

Not surprisingly, experimentation shows that poor performance results when the radix is smaller than the biggest difference between symbol ordinal values. The presentation above presumed that the symbols have ordinal values covering a contiguous range. If not, they may be mapped to such a range (by modifying T) to permit the smallest radix choice.

Experimental evaluation of speed and uniformity are presented in later sections.

---

[4]There is some mathematical sloppiness here. Where convenient, the author is treating elements of $\mathbf{Z}/B$ as simple integers rather than entire cosets. The smallest nonnegative member of the coset is used for the address.

[5]By "recursive," the author means only that the hash value for a given key is derived from its predecessors. Many hash functions are recursive in another sense: the hash value for a *single* key is recursively calculated, with each ineration including more of the key.

Table I. Implementation of Integer Division Hashing by Recursion

```
RadixToTheN := (Radix ^ NgramLength) mod B;          { For removing old symbol }
HashValue := 0;                                      { Make initial value }
for i := 1 to NgramLength do
  begin
    HashValue := Radix * HashValue + Ord(S[i]);
    HashValue := HashValue mod B;
  end;
      ⋮                                              { Use the value }
for OldIndex := 1 to N − 1 do                        { Make all other values }
  begin
    NewIndex := OldIndex + NgramLength;
    HashValue := Radix * HashValue;                  { Multiply by radix }
    HashValue := HashValue + Ord(S[NewIndex]);       { Include new symbol }
                                                     { Remove old symbol }
    HashValue := HashValue − RadixToTheN * Ord(S[OldIndex]);
    HashValue := HashValue mod B;                    { Reduce by modulus }
      ⋮                                              { Use the value }
  end;
```

The symbols are in the vector S. The MOD function is assumed to return a number in {0, 1, . . . B − 1}, where B is the number of hash table buckets. See cautions in the text.

2.2 Hashing by Power-of-2 Integer Division

Evaluation of the modulo function found in hashing by prime integer division slows the processing considerably. Gonnet and Baeza-Yates [1990] offer a variation of hashing by integer division in which the modulo function may be eliminated completely, simply by choosing a modulus equal to a power of 2. In this scheme, overflows may be ignored, and the final address is obtained simply by masking unwanted high-order bits. The radix is chosen for long cycle length.[6] For example the choices 27 or 259 will do.

An implementation of recursive hashing by power-of-2 integer division is illustrated in Table II. The first n-gram is hashed directly; the remainder are recursively hashed.

The ring in this case is not an integral domain, since the modulus is composite. In fact, any biases in the data spaced by powers of 2 will be reflected in the hash distribution. In particular, if the number of characters with odd ordinal values differs substantially from those with even ordinal values, distribution of the hash values will suffer.

Experimental evaluation of speed and uniformity is presented in later sections.

[6]Given radix r, the sequence $r^1 \bmod B$, $r^2 \bmod B$, $r^3 \bmod B$, . . . will repeat. The cycle length is the number of values assumed in one such cycle. The cycle length is also equal to the smallest $i > 0$ such that $r^i \bmod B = 0$. The radix is chosen so that this cycle is as long as possible.

Table II. Implementation of Recursive Power-of-2 Integer Division Hashing

```
RadixToTheN := (Radix ^ NgramLength) mod B;          { For removing old symbol }
AddressMask := B − 1;                                { Mask }

HashWord := 0;                                       { Make initial value }
for i := 1 to NgramLength do
  begin
    HashWord := Radix * HashWord + Ord(S[i]);
    HashWord := HashWord mod B;
  end;
HashValue := BAND(HashWord, AddressMask);            { Drop high order bits }
      ⋮                                              { Use the value }

for OldIndex := 1 to N − 1 do                        { Make all other values }
  begin
    NewIndex := OldIndex + NgramLength;
    HashWord := Radix * HashValue;                   { Multiply by radix }
    HashWord := HashWord + Ord(S[NewIndex]);         { Include new symbol }
                                                     { Remove old symbol }
    HashWord := HashWord − RadixToTheN * Ord(S[OldIndex]);
    HashValue := BAND(HashWord, AddressMask);        { Drop high bits }
      ⋮                                              { Use the value }
  end;
```

The symbols are in the vector S. B is the number of hash table buckets and must be a power of 2. BAND performs a bitwise AND. Overflows are ignored.

2.3 Hashing by Cyclic Polynomials

The implementation of recursive integer division hashing using prime moduli requires integer addition and multiplication[7] and calculation of a MOD function, the combination of which may be slow. Moreover, if overflow is possible, its remedy requires even more time. If hashing by power-of-2 division is done, then integer addition and multiplication are still required, and the nonprime modulus may lead to distribution skew. An attempt to deal with these problems leads to another realization.

Consider operating over $R = \mathrm{GF}(2)[x]/(x^w + 1)$, the ring consisting of polynomials in x whose coefficients are binary (drawn from the Galois field of characteristic 2), reduced modulo the polynomial $x^w + 1$.[8] (See the Appendix.) The integer w is chosen equal to the computer's word size, that is, w is the number of bits in each word. The polynomials are represented by w-bit words as one would suppose: the ith bit is the coefficient of x^i, $i = 0, 1, \ldots, w - 1$. There is a one-to-one correspondence between elements in $\mathrm{GF}(2)[x]/(x^w + 1)$ and binary values that can be accommodated by a word of size w.

[7]Karp and Rabin [1987] point out that multiplication can be reduced to shifting if the radix is a power of 2. While this will result in some speed improvement, it is likely to worsen distribution.

[8]See Appendix Section A.2 for a definition of the Galois field of two elements, Section A.3 for a discussion of a ring modulo an element, and Section A.4 for a discussion of polynomial rings.

Addition of two such polynomials is performed by an exclusive-or of the corresponding words. And for any polynomial $q \in R$ and any integer Δ, the multiplication $x^{\Delta}q$ is implemented in the computer as a barrel shift on the word representing q, as shown below. The simplicity of these operations compels one to consider hashing in GF(2)$[x]/(x^w + 1)$, using the recursion

$$H(S_1) = \sum_{i=1}^{n} x^{\Delta(n-i)}T(s_i)$$

$$H(S_i) = x^{\Delta}H(S_{i-1}) + T(s_{i+n-1}) + x^{n\Delta}T(s_{i-1}), \quad 1 < i \le N.$$

This is Eq. (2) with $r = x^{\Delta}$. (Note that addition and subtraction are the same in this ring.)

In this ring, since everything is reduced modulo $x^w + 1$, the polynomial $x^w + 1$ is equal to the zero polynomial; in other words $x^w + 1 = 0$, or $x^w = 1$. Consider an arbitrary member $q \in R$ with $q(x) = q_{w-1}x^{w-1} + q_{w-2}x^{w-2} + \cdots + q_0$. Then

$$xq(x) = q_{w-1}x^w + q_{w-2}x^{w-1} + \cdots + q_0x$$
$$= q_{w-2}x^{w-1} + q_{w-3}x^{w-2} + \cdots + q_0x + q_{w-1}.$$

So multiplication by x amounts to a barrel shift of the word holding the coefficients. Hence the name "cyclic polynomials."

The elements that have not yet been defined are the transformations T and A and the choice of Δ. First, consider the address transformation A. The polynomials represented by binary words have obvious interpretations as integer addresses; however, the number of polynomials, 2^w, will (likely) be larger than the number of hash buckets, B. By choosing $B = 2^t$, $t \le w$, the bucket addresses can be obtained from H simply by masking (ignoring) the upper $w - t$ bits. The address function consists of this masking.

The symbol transformation is a lookup table containing "random" elements of GF(2)$[x]/(x^w + 1)$, indexed by the ordinal value of its argument. In the author's practice, the table consists of words whose contents are filled from a computer's random-number generator. In this manner, each symbol's contribution is scattered across the word, changing about half of the bits, on average. (One could propose many strategies for filling these words, some of which would undoubtedly be better.) Note that for removing the contribution of the old symbol, the transformation T may be combined with its multiplication by $x^{n\Delta}$ to form a new transformation

$$T'(s) = x^{n\Delta}T(s).$$

T' may also be precomputed and be implemented by a lookup table.

The method of hashing by cyclic polynomials was suggested to the author by J. M. Kubina (private communication, 1993).

Table III. Implementation of Recursive Hashing by Cyclic Polynomials

```
for i := 0 to MaxSymbolOrd do                          { Symbol transformation table }
    TransformationT[i] := RandomWord;
for i := 0 to MaxSymbolOrd do                          { Table to remove old symbol }
    TransformationTPrime[i] := BROTL(TransformationT[i], Delta*NgramLength);
AddressMask := B − 1;                                  { Mask; B must be power of 2 }
HashWord := 0;                                         { Make initial value }
for i := 1 to NgramLength do
  begin
    HashWord := BROTL(HashWord, Delta);                { Multiply by radix }
                                                       { Include new symbol }
    HashWord := BXOR(HashWord, TransformationT[Ord(S[i])]);
  end;
HashValue := BAND(HashWord, AddressMask);              { Drop high order bits }
      ⋮                                                { Use the value }
for OldIndex := 1 to N − 1 do                          { Make all other values }
  begin
    NewIndex := OldIndex + NgramLength;
    HashWord := BROTL(HashWord, Delta);                { Multiply by radix }
                                                       { Include new symbol }
    HashWord := BXOR(HashWord, TransformationT[Ord(S[NewIndex])]);
                                                       { Remove old symbol }
    HashWord := BXOR(HashWord, TransformationTPrime[Ord(S[OldIndex])]);
    HashValue := BAND(HashWord, AddressMask);          { Drop high order bits }
      ⋮                                                { Use the value }
  end;
```

The symbols are in the vector S. The number of hash table buckets, B, must be a power of 2. The function BROTL implements a left barrel shift; BXOR performs a bitwise exclusive-or; and BAND performs a bitwise AND. RandomWord is a function that returns a new random word with each call. For most applications, MaxSymbolOrd will equal 255.

Table III shows an implementation of hashing by cyclic polynomials, as described just above. At initialization, lookup tables for transformations T and T' are built. Also, an address mask is established to remove the high-order bits for the address transformation.

Note that the expensive integer functions have been avoided, as well as concerns about overflows. One might be concerned, however, with the cost of table lookups.

The choice of Δ, the number of bits shifted at each step of recursion, was examined experimentally. While no choice was clearly superior, some choices were clearly bad, such as the choice of 8 for a word size of 32 bits. The choice $\Delta = 1$ was found to be as good as any and was used to produce the results presented later in this article.

At least three causes for concern may arise here. First, the capricious choice of table lookup words implementing T is worrisome. Clearly, there are bad choices possible. Experiment suggests that this concern is unwar-

ranted. (Conversely, a nearly limitless family of hash functions can be had, each member of which is identified with a different choice of T.) Second, the polynomial $x^w + 1$ is composite, so that the ring is not an integral domain. As the Appendix points out (Section A.5), for most implementations the lack of an integral domain here does not matter. Finally, one may note that the recursion has a cycle length of $c = w/\mathrm{GCD}(w, \Delta)$, so that if an n-gram consists of c symbols followed by those same c symbols, the hash value will be zero. This last concern applies only to long n-grams.

Experimental performance measurement of this approach is described in later sections.

Before going on, one may note that a *nonrecursive* version of this method of hashing has appeared in Pryor et al. [1993].

2.4 Hashing by General Polynomial Division

Hashing by cyclic polynomials overcomes the objections to hashing by integer division, but it is more restrictive than necessary. This section introduces, by analogy to integer division, a generalized polynomial division scheme which is nearly as fast, permits use of any polynomial, and may be computed over an integral domain.

In this approach, symbols are again represented by polynomials of binary coefficients, but they are reduced modulo a general polynomial p. The ring of computation is, then, $R = \mathrm{GF}(2)[x]/p(x)$. If p is irreducible, R is an integral domain.[9] Let $B = 2^t$, and let the degree of p be d, with $d \geq t$. The 2^d elements of R have an obvious one-to-one representation in binary words of d bits; words of $w > d$ bits with the $w - d$ high-order bits set to zero will also serve. In this representation, the ith bit is the coefficient of x^i, $i = 0, 1, \ldots, d - 1$.

The recursion radix is chosen to be the polynomial x; the computer implementation of multiplication by x in this ring is extremely simple and fast. Let $p(x) = x^d + p_{d-1}x^{d-1} + p_{d-2}x^{d-2} + \cdots + p_0$ and consider an arbitrary member $q \in R$ with $q(x) = q_{d-1}x^{d-1} + q_{d-2}x^{d-2} + \cdots + q_0$. Then, since $p(x) = 0$ in this ring, it follows that $x^d = p_{d-1}x^{d-1} + p_{d-2}x^{d-2} + \cdots + p_0$, so that

$$xq(x) = \begin{cases} q_{d-2}x^{d-1} + q_{d-3}x^{d-2} + \cdots + q_0x, & q_{d-1} = 0 \\ (q_{d-2} + p_{d-1})x^{d-1} + (q_{d-3} + p_{d-2})x^{d-2} + \cdots + p_0, & q_{d-1} = 1 \end{cases}.$$

As an example, consider the case $p = x^3 + x + 1$, $q = x^2 + 1$. Using the fact that $x^3 = x + 1$ in this ring, one finds that

$$xq = x(x^2 + 1) = x^3 + x = (x + 1) + x = 1.$$

So multiplication by x amounts to a shift (no carry) of the word holding the coefficients, and, depending upon a test of one bit, a possible exclusive-or with a word representing the polynomial p.

---

[9]See Appendix Section A.5.

The recursion to be implemented for this case is

$$H(S_1) = \sum_{i=1}^{n} x^{n-i} T(s_i)$$

$$H(S_i) = xH(S_{i-1}) + T(s_{i+n-1}) + x^n T(s_{i-1}), \quad 1 < i \leq N,$$

leaving only the issues of input and address transformation. Just as for the cyclic polynomials example, the bucket address $h(S_i)$ is obtained from $H(S_i)$ by taking the lower t bits of the word representing the polynomial $H(S_i)$. This is done simply by masking.

Also as for hashing by cyclic polynomials, symbol transformation is performed by consulting a lookup table containing "random" elements of GF(2)$[x]/p(x)$, indexed by the ordinal value of its argument. To remove the contribution of the old symbol, the transformation T may be combined with multiplication by x^n to form a new transformation

$$T'(s) = x^n T(s).$$

T' may also be precomputed and be implemented by a lookup table.

An implementation of hashing by polynomial division is illustrated in Table IV. Without change, this implementation is suitable for any power-of-2 table size up through 2^{19} and any word size greater than or equal to 20 bits. Other polynomials may be chosen for different applications.

The author chose $p(x)$ to be a primitive polynomial of degree 19 drawn from Peterson [1961]:

$$p(x) = x^{19} + x^{18} + x^{17} + x^{16} + x^{12} + x^7 + x^6 + x^5 + x^3 + x + 1.$$

This polynomial has 11 nonzero coefficients, causing about half of the bits to change state each time the polynomial is added in.

The author chose to build the table for T by using $p(x)$ to define a recursion. Given some seed polynomial $\theta(x) \neq 0$, T assigns to each symbol s the polynomial

$$T(s) = x^{(n+1)\mathrm{Ord}(s)} \theta(x).$$

The example of Table IV employs a seed of $\theta(x) = x^d + x^{d-1} + x^{d-2} + \cdots + 1$. Alternatively, T may be produced by repeated calls to a random-word generator.

Any polynomial accommodated by the word size may be used in this method. Indeed, this approach makes practical a (nonrecursive) hash function proposed by Hanan and Palermo [1963] and Schay and Raver [1963]. Their suggestion was to regard the key and hash value as vectors of polynomial coefficients (as done here), letting the hash value be the remainder after division by the generating polynomial of a Bose-Chaudhuri-Hocquenghem (BCH) code (see Berlekamp [1984] for a complete

Table IV. Implementation of Recursive Hashing by Polynomial Division

```
Polynomial := $F10EB;                                  { Modulus polynomial (in hex) }
PolyDegree := 19;                                      { Degree of modulus polynomial }
AddressMask := B − 1;                                  { Mask; B must be power of 2 }

HashWord := −1;                                        { Symbol transformation tables }
for i := 0 to MaxSymbolOrd do
  for j := 0 to NgramLength do
    begin
      HashWord := BSL(HashWord, 1);                    { Multiply by radix }
      if BTST(HashWord, PolyDegree) then
        HashWord := BXOR(HashWord, Polynomial);
      if j = 1 then                                    { Record in tables }
        TransformationT[i] := HashWord
      else if j = NgramLength then
        TransformationTPrime[i] := HashWord;
    end;

HashWord := 0;                                         { Make initial value }
for i := 1 to NgramLength do
  begin
    HashWord := BSL(HashWord, 1);                      { Multiply by radix }
    if BTST(HashWord, PolyDegree) then
      HashWord := BXOR(HashWord, Polynomial);
                                                       { Include new symbol }
    HashWord := BXOR(HashWord, TransformationT[Ord(S[i])]);
  end;
HashValue := BAND(HashWord, AddressMask);              { Drop high order bits }
      ⋮                                                { Use the value }

for OldIndex := 1 to N − 1 do                          { Make all other values }
  begin
    NewIndex := OldIndex + NgramLength;
    HashWord := BSL(HashWord, 1);                      { Multiply by radix }
    if BTST(HashWord, PolyDegree) then
      HashWord := BXOR(HashWord, Polynomial);
                                                       { Include new symbol }
    HashWord := BXOR(HashWord, TransformationT[Ord(S[NewIndex])]);
                                                       { Remove old symbol }
    HashWord := BXOR(HashWord, TransformationTPrime[Ord(S[OldIndex])]);
    HashValue := BAND(HashWord, AddressMask);          { Drop high order bits }
      ⋮                                                { Use the value }
  end;
```

The symbols are in the vector S. The number of hash table buckets, B, must be a power of 2. The function BSL implements a left shift with no carry; BXOR performs a bitwise exclusive-or; BTST tests a single bit (returning true if the bit is 1); and BAND performs a bitwise AND. For most applications, MaxSymbolOrd will equal 255. The polynomial here is primitive. As an alternative, the input transformation may be made from calls to a random number generator as in Table III.

discussion of BCH codes). In this setting, the hash value would correspond to the syndrome of the key codeword. BCH codes enjoy a minimum-distance property whose manifestation in this application is that any distinct keys differing in fewer than some number of bits (polynomial coefficients)

produce different hash values. The idea was to break up "clusters" of similar key values, sending them to different table buckets. Their proposal seems to have remained unpopular because the computation was not amenable to fast-software implementation. Recursive hashing of n-grams offers another (and practical) approach to BCH hashing, since computation is just as fast as for any other choice of modulus polynomial.

As for hashing by cyclic polynomials, one may be concerned with the choice of "random" input transformation polynomials for input transformation. Experiment suggests that this is not a critical issue.

Some performance measures of this method are provided in the following sections.

2.5 Self-Annihilating Recursive Methods

The approach of methods outlined above has been to introduce a symbol, let its contribution be multiplied by the radix n times, and then remove the symbol's contribution. For special cases of ring and radix, one may arrange things such that the contribution of each symbol vanishes after n iterations, so that no removal of its contribution is needed. A substantial improvement in computation speed may result.

This self-annihilation is guaranteed if $r^n = 0$ in R. Moreover, if $r^i \neq 0$ for $i < n$, then this scheme will directly implement hashing of n-grams without n appearing explicitly in the recursion. Calculation of the hash values is done by the simpler procedure

$$H(S_1) = \sum_{i=1}^{n} r^{n-i} T(s_i)$$

$$H(S_i) = rH(S_{i-1}) + T(s_{i+n-1}), \quad 1 < i \leq N.$$

The fact that the radix must be a zero divisor precludes R being an integral domain. One can, however, find choices of r, n, and R for which self-annihilating recursion appears to work well.

Example 2.5.1. A modification of hashing by cyclic polynomials offers self-annihilating recursion for the special case $n = 2^L$, for any integer L. Assume that the word size is $w = 2^D$. Then in $R = \mathrm{GF}(2)[x]/(x^w + 1)$, the radix choice of $r = 1 + x^{w/n}$ will do, since $r^n = (1 + x^{2^{D-L}})^{2^L} = 1 + x^w = 0$ and $r^i \neq 0$ for $i < n$. Table V illustrates an implementation of this approach. Note that this should be faster than the method outlined in Table III, since the symbol stream and lookup table are consulted only once per iteration, rather than twice.

Results of using self-annihilating recursion are presented at the end of Sections 3.1.2 and 3.2.

3. PERFORMANCE MEASUREMENT EXPERIMENTS

Two objective measures of a hash function's performance are speed of computation and uniformity of resulting address distribution. This section

Table V. Implementation of Self-Annihilating Recursive Hashing by Cyclic Polynomials

```
for i := 0 to MaxSymbolOrd do                          { Symbol transformation table }
    TransformationT[i] := RandomWord;
AddressMask := B − 1;                                  { Mask; B must be power of 2 }
HashWord := 0;                                         { Make initial value }
for i := 1 to NgramLength do
  begin
                                                       { Multiply by radix }
    HashWord := BXOR(HashWord, BROTL(HashWord, Delta));
                                                       { Include new symbol }
    HashWord := BXOR(HashWord, TransformationT[Ord(S[i])]);
  end;
HashValue := BAND(HashWord, AddressMask);              { Drop high order bits }
      ⋮                                                { Use the value }
                                                       { Make all other values }
for NewIndex := NgramLength + 1 to NgramLength + N − 1 do
  begin
                                                       { Multiply by radix }
    HashWord := BXOR(HashWord, BROTL(HashWord, Delta));
                                                       { Include new symbol }
    HashWord := BXOR(HashWord, TransformationT[Ord(S[NewIndex])]);
    HashValue := BAND(HashWord, AddressMask);          { Drop high order bits }
      ⋮                                                { Use the value }
  end;
```

The symbols are in the vector S. The number of hash table buckets, B, must be a power of 2. Both NgramLength and the word size W must be powers of 2. The shift DELTA is equal to W/NgramLength. The function BROTL implements a left barrel shift; BXOR performs a bitwise exclusive-or; and BAND performs a bitwise AND. RandomWord is a function that returns a new random word with each call. For most applications, MaxSymbolOrd will equal 255.

describes experiments to measure both and presents the results of those experiments.

3.1 Nonuniformity Experiments

3.1.1 *A Nonuniformity Statistic in Two Guises.* This section describes and justifies a commonly used statistic for measuring nonuniformity. The goal is to develop a sensitive indication of nonuniformity whose behavior can be characterized in the "random" case, that is, when the hash function randomly produces an output, uniform over the table addresses, regardless of its input. Given this characterization, one can assess the significance of each result.

Suppose that an experiment to evaluate the performance of hash function h consists of examining $h(k_1)$, $h(k_2)$, . . . , $h(k_N)$, where each of the N keys is unique. Let the hash function assume values 0, 1, . . . , $B - 1$, where B is the number of hash table "bins." The counts $\mathbf{C} = \{C_0, C_1, \ldots, C_{B-1}\}$ give

the number of times each hash address is produced in the experiment, that is,

$$C_i = \sum_{j=1}^{N} \mathbf{I}\{h(k_j) = i\},$$

where $\mathbf{I}\{\ \}$ is the indicator function, which assumes a value of 1 when its argument is true and zero otherwise. The ratio

$$\alpha = N/B$$

is the table's so-called "load factor."

Assume for now the hypothesis that the hash function assigns to each key a random address, so that the probability of assigning any particular key to any particular address is $1/B$, and each such assignment is independent. This is the "uniform hashing" model introduced in Peterson [1957]. Under these assumptions, $\mathbf{C}$ is multinomial with $\mathrm{E}\{C_i\} = N/B = \alpha$.

The counts may be regarded as entries in a contingency table. A standard statistic for evaluating hypotheses of structure in contingency tables is the Pearson χ^2 statistic (see, for example, Bishop et al. [1975, Ch. 4], or Bickel and Doksum [1977, Ch. 8]). It is based on the maximum-likelihood estimates $\hat{\mathrm{E}}\{C_i\}$ of the table means, under the hypothesis being examined[10] and is given by

$$\chi^2 = \sum_{i=0}^{B-1} \frac{(C_i - \hat{\mathrm{E}}\{C_i\})^2}{\hat{\mathrm{E}}\{C_i\}} = \frac{1}{\alpha} \sum_{i=0}^{B-1} (C_i - \alpha)^2.$$

The Pearson χ^2 does not actually have a chi-squared distribution, though it is approximately so with $B - 1$ degrees of freedom. The approximation becomes better with larger α; $\alpha \gg 1$ is recommended for measuring significance in the general case. Regardless of α, $\mathrm{E}\{\chi^2\} = B - 1$ and $\mathrm{Var}\{\chi^2\} = 2(B - 1)$. Moreover, for large B, χ^2 is well approximated as normal. These considerations suggest the uniformity statistic

$$U = \frac{\chi^2 - (B - 1)}{\sqrt{2(B - 1)}}.$$

For the uniform-hashing assumption, U has unit variance and zero mean and, with appropriately large α and B, is approximately normal.

To examine a candidate hash function h with respect to the set of distinct keys $\{k_1, k_2, \ldots, k_N\}$, the hash values $h(k_1), h(k_2), \ldots, h(k_N)$ are determined; the associated counts $\mathbf{C} = \{C_0, C_1, \ldots, C_{B-1}\}$ are formed;

[10]The hypothesis under test assumes all of the means equal to the load factor, so that the maximum-likelihood "estimates" of the means degenerate to α, requiring no estimating at all.

and the statistic U is produced from the counts. A value near zero indicates performance comparable with uniform hashing. A large positive value corresponds to a more nonuniform distribution; a large negative value indicates uniformity in excess of that expected by random assignment.[11]

The nonuniformity statistic may also be interpreted as a measure of excess time needed to handle table collisions. Suppose that each table bucket is an independent linked list. The work required to construct one of those lists having m items is proportional to $m(m + 1)$. To construct all of the lists in the table takes work proportional to

$$W = \sum_{i=0}^{B-1} C_i(C_i + 1) = \alpha(\chi^2 + N + B).$$

On the other hand, the expected work required to construct all lists for the uniform-hashing case is

$$W_0 = \mathrm{E}\left\{ \sum_{i=0}^{B-1} C_i(C_i + 1) \right\} \Bigg|_{\substack{\text{uniform} \\ \text{hashing}}} = \alpha[(B - 1) + N + B].$$

The excess fractional work required for hash function h, relative to uniform hashing, is

$$\omega = \frac{W}{W_0} - 1 = \frac{\sqrt{2(B - 1)}}{2(B - 1) + N + 1} U \approx \frac{\sqrt{2}\ U}{\sqrt{B}(2 + \alpha)}.$$

So extra work is proportional to distribution nonuniformity U; the proportionality constant depends upon hash table size and loading.

Both U and ω report the same measurement χ^2. U is a very sensitive measure for revealing small differences between hashing distributions and is characterized in the uniform case; ω keeps the difference between hashing distributions in perspective, since it is normalized to the total work. Both of these numbers will be reported in uniformity tests.

3.1.2 *Nonuniformity Tests.* This section applies the proposed hashing functions to samples of text in an effort to measure hash uniformity. Two samples of text were used. The first, labeled hereafter as "English," consisted of a concatenation of 144 pages of *Time*, *Newsweek*, and *Science News*, obtained by optical character recognition. The combined length of these documents was on the order of 587,000 characters. The second, hereafter called "Japanese," was a concatenation of 431 documents drawn from the DARPA Tipster Japanese joint venture database, for a total size of

[11]Operationally, the smaller the uniformity statistic is, the better. Practically, though, large negative values, indicating superflatness, suggest a dependency on the key distribution that one would not want to count on.

Table VI. Nonuniformity Score U and Excess Fractional Work ω for 3-Grams in English Text

Method	Nominal Table Size		
	8K	32K	128K
Prime integer division	−17.1 (−9.7%)	−24.2 (−8.6%)	−12.1 (−2.3%)
Power-of-2 integer division	−17.9 (−10.1%)	−24.2 (−8.6%)	−12.1 (−2.3%)
Cyclic polynomials	+0.8 (+0.5%)	−2.7 (−0.9%)	−0.9 (−0.2%)
Polynomial division	+1.9 (+1.1%)	−0.4 (−0.1%)	−2.5 (−0.5%)

The excess work factor is in parentheses. The text sample consisted of 549,665 valid *n*-grams, of which 6194 were unique. The table size for prime integer division was equal to the largest prime less than the nominal size.

approximately 384,000 bytes. The Japanese documents were in their original form: shifted JIS format, two bytes per character. To process the Japanese data, each *byte* was treated as a different symbol, so that each three-gram, for example, held one and a half Japanese characters.[12]

Four methods of recursive hashing were examined. Prime integer division was carried out using a radix of 27 for the English text and 257 for the Japanese text. The choice of 27 accommodated English alphabetic characters and space (lowercase letters were mapped to uppercase); the space character's ordinal value was mapped to be adjacent to the alphabet, coming right after "Z." The choice of 257 permitted the use of all Japanese codes. It was quickly observed that the more obvious radix choice of 256 caused great nonuniformity, while 257 worked quite well. Hashing by power-of-2 integer division used the radix 27 for English and the radix 259 for Japanese. Both choices give maximum cycle length.

Hashing by cyclic polynomials employed the algorithm shown in Table III, with Delta = 1. Computation was carried out with a word size of 32 bits.

Hashing by polynomial division was conducted using the algorithm illustrated in Table IV and employing the primitive polynomial modulus given there.

For each method of hashing, three table sizes were tried: 8192 (8K), 32,768 (32K), and 131,072 (128K), offering a range of table loading. In the case of hashing by prime integer division, the table sizes were chosen to be the largest primes less than these nominal sizes.

For each table size and hashing method, all valid n-grams were counted for $n = 3, 4, 5, 6$, and 10. For the English data, valid n-grams were those that consisted only of characters of the alphabet and the space character. All Japanese n-grams were considered valid. The range $3 \leq n \leq 6$ covers most of the work in the literature; the value $n = 10$ was included to see if behavior changed for larger sizes.

[12]Breaking up characters this way may strike one as a mistake, but processing done in this fashion has been very successful. See Damashek [1995] and Cohen [1995] for details.

Table VII. Nonuniformity Score U and Excess Fractional Work ω for 4-Grams in English Text

Method	Nominal Table Size		
	8K	32K	128K
Prime integer division	−5.7 (−1.7%)	−2.6 (−0.7%)	−8.4 (−1.5%)
Power-of-2 integer division	−5.0 (−1.5%)	+4.9 (+1.3%)	−10.2 (−1.8%)
Cyclic polynomials	−1.3 (−0.4%)	−2.4 (−0.7%)	−2.1 (−0.4%)
Polynomial division	+1.6 (+0.5%)	+2.8 (+0.8%)	+0.7 (+0.1%)

The excess work factor is in parentheses. The text sample consisted of 535,428 valid n-grams, of which 27,310 were unique. The table size for prime integer division was equal to the largest prime less than the nominal size.

Results of the uniformity tests appear in Tables VI through XV. For each test, the tables report the standardized nonuniformity statistic U and excess fractional work factor ω. The varying choices of n-gram length and table size resulted in load factors ranging from 0.047 to 34.4.

As the tables show, the hash functions did a good job on the whole. Even the largest nonuniformity score resulted in an excess work factor of only 7.3%. While the integer division methods often returned the flattest distributions, they were also the most erratic and responsible for the least uniform results.[13] Hashing by cyclic polynomials and polynomial division worked equally well on English and Japanese; hashing by integer division worked (suspiciously) well on English, about the same as the other methods on long Japanese n-grams, and badly on short Japanese n-grams in big tables.

A summary of the nonuniformity measurements is offered in Table XVI. As remarked earlier, integer division exhibited the most erratic behavior, often outscoring the other methods, but occasionally doing relatively badly. Integer division by a power of 2 turned in the best mean behavior. Hashing by cyclic polynomials worked slightly better than hashing by polynomial division. Both, however, were even performers, operating near the "ideal" of uniform hashing.

The author discovered that U was very sensitive. Minor errors in implementation or measurement caused drastic changes in the statistic. This suggests that the good scores shown above are all that much more significant.

It should be pointed out that the numbers above are dependent upon the choice of data. Indeed, given any hashing function, an adversary may always choose data that will result in an appalling distribution of hash values. The numbers are also a function of the parameters of the algo-

[13]It is the author's belief that the erratic behavior of the integer division methods stems from the fact that such hashing is more a "folding" than a randomization. Successive keys are mapped to successive addresses until the modulus is reached, whereupon the addresses "fold" over. It is easy to see how such a scheme could result in a very flat distribution or in a bad one, depending on the key distribution.

Table VIII. Nonuniformity Score U and Excess Fractional Work ω for 5-Grams in English Text

Method	Nominal Table Size		
	8K	32K	128K
Prime integer division	−7.0 (−1.0%)	−4.9 (−0.9%)	−3.4 (−0.5%)
Power-of-2 integer division	−5.2 (−0.8%)	−5.3 (−1.0%)	−8.2 (−1.3%)
Cyclic polynomials	−1.8 (−0.3%)	−0.4 (−0.1%)	+1.6 (+0.2%)
Polynomial division	+1.4 (+0.2%)	+3.4 (+0.6%)	+2.9 (+0.4%)

The excess work factor is in parentheses. The text sample consisted of 521,661 valid *n*-grams, of which 71,291 were unique. The table size for prime integer division was equal to the largest prime less than the nominal size.

Table IX. Nonuniformity Score U and Excess Fractional Work ω for 6-Grams in English Text

Method	Nominal Table Size		
	8K	32K	128K
Prime integer division	−3.7 (−0.3%)	−2.4 (−0.3%)	−1.8 (−0.2%)
Power-of-2 integer division	−2.9 (−0.3%)	−3.0 (−0.4%)	−2.3 (−0.3%)
Cyclic polynomials	−2.5 (−0.2%)	−1.3 (−0.2%)	−1.7 (−0.2%)
Polynomial division	−0.4 (−0.0%)	+0.9 (+0.1%)	−0.3 (−0.0%)

The excess work factor is in parentheses. The text sample consisted of 508,091 valid *n*-grams, of which 134,011 were unique. The table size for prime integer division was equal to the largest prime less than the nominal size.

Table X. Nonuniformity Score U and Excess Fractional Work ω for 10-Grams in English Text

Method	Nominal Table Size		
	8K	32K	128K
Prime integer division	−0.9 (−0.0%)	−0.7 (−0.0%)	+0.6 (+0.0%)
Power-of-2 integer division	−1.3 (−0.0%)	−0.4 (−0.0%)	+0.2 (+0.0%)
Cyclic polynomials	+1.0 (+0.0%)	+1.3 (+0.1%)	+1.3 (+0.1%)
Polynomial division	+0.5 (+0.0%)	+0.6 (+0.0%)	+0.3 (+0.0%)

The excess work factor is in parentheses. The text sample consisted of 549,665 valid *n*-grams, of which 339,633 were unique. The table size for prime integer division was equal to the largest prime less than the nominal size.

rithms (the radix and modulus for integer division, the polynomial for polynomial division) and the "random" polynomials selected for the two polynomial methods. The author conducted many more experiments on different data and with different choices of parameters. In particular, a variety of polynomials were tried in testing the polynomial division method. The results presented are representative. Runs with different choices resulted in similar results that were different only in the details. Such variation makes the choice of a clear "winner" between integer division and cyclic polynomials on the basis of uniformity difficult.

Table XI. Nonuniformity Score U and Excess Fractional Work ω for 3-Grams in Japanese Text

Method	Nominal Table Size		
	8K	32K	128K
Prime integer division	−4.9 (−1.3%)	−3.7 (−1.0%)	+42.0 (+7.3%)
Power-of-2 integer division	−4.7 (−1.2%)	−2.3 (−0.6%)	−3.5 (−0.6%)
Cyclic polynomials	−0.1 (−0.0%)	−0.9 (−0.2%)	−2.0 (−0.3%)
Polynomial division	+0.3 (+0.1%)	−0.6 (−0.2%)	−1.9 (−0.3%)

The excess work factor is in parentheses. The text sample consisted of 384,547 valid n-grams, of which 30,574 were unique. The table size for prime integer division was equal to the largest prime less than the nominal size.

Table XII. Nonuniformity Score U and Excess Fractional Work ω for 4-Grams in Japanese Text

Method	Nominal Table Size		
	8K	32K	128K
Prime integer division	−3.9 (−0.6%)	−2.3 (−0.4%)	+26.9 (+4.1%)
Power-of-2 integer division	−1.2 (−0.2%)	+2.2 (+0.4%)	+9.8 (+1.5%)
Cyclic polynomials	+1.0 (+0.2%)	−1.4 (−0.3%)	−2.6 (−0.4%)
Polynomial division	+0.4 (+0.1%)	−1.3 (−0.3%)	−2.2 (−0.3%)

The excess work factor is in parentheses. The text sample consisted of 384,116 valid n-grams, of which 70,183 were unique. The table size for prime integer division was equal to the largest prime less than the nominal size.

Table XIII. Nonuniformity Score U and Excess Fractional Work ω for 5-Grams in Japanese Text

Method	Nominal Table Size		
	8K	32K	128K
Prime integer division	−1.3 (−0.1%)	−1.1 (−0.2%)	+3.6 (+0.5%)
Power-of-2 integer division	+1.0 (+0.1%)	+0.7 (+0.1%)	+0.9 (+0.1%)
Cyclic polynomials	−0.1 (−0.0%)	+0.8 (+0.1%)	+0.9 (+0.1%)
Polynomial division	+1.6 (+0.1%)	−0.3 (−0.0%)	−0.6 (−0.1%)

The excess work factor is in parentheses. The text sample consisted of 383,685 valid n-grams, of which 117,715 were unique. The table size for prime integer division was equal to the largest prime less than the nominal size.

Some experiments were also conducted with self-annihilating recursive hashing by cyclic polynomials. Two examples are reported here, both involving a word size of 32 bits and a table size of 32KB buckets. The test files were the same used to generate the results in Tables VI through XVI. Nonuniformity is given in Table XVII. As the table shows, this implementation resulted in nonuniformity that was in excess of that shown in Tables VI through XV, but is probably tolerable for most applications.

Table XIV. Nonuniformity Score U and Excess Fractional Work ω for 6-Grams in Japanese Text

Method	Nominal Table Size		
	8K	32K	128K
Prime integer division	−0.4 (−0.0%)	−1.2 (−0.0%)	+0.7 (+0.1%)
Power-of-2 integer division	+0.6 (+0.0%)	+0.7 (+0.1%)	+0.3 (+0.0%)
Cyclic polynomials	−1.7 (−0.1%)	−0.8 (−0.1%)	−0.2 (−0.0%)
Polynomial division	−0.8 (−0.1%)	+0.2 (+0.0%)	+1.1 (+0.1%)

The excess work factor is in parentheses. The text sample consisted of 383,252 valid *n*-grams, of which 163,800 were unique. The table size for prime integer division was equal to the largest prime less than the nominal size.

Table XV. Nonuniformity Score U and Excess Fractional Work ω for 10-Grams in Japanese Text

Method	Nominal Table Size		
	8K	32K	128K
Prime integer division	+0.8 (+0.0%)	+1.0 (+0.1%)	+0.7 (+0.1%)
Power-of-2 integer division	+1.0 (+0.0%)	+1.3 (+0.1%)	+0.1 (+0.0%)
Cyclic polynomials	+0.8 (+0.0%)	−1.0 (−0.1%)	+0.7 (+0.1%)
Polynomial division	+1.1 (+0.0%)	+1.5 (+0.1%)	+1.1 (+0.1%)

The excess work factor is in parentheses. The text sample consisted of 381,530 valid *n*-grams, of which 163,800 were unique. The table size for prime integer division was equal to the largest prime less than the nominal size.

Table XVI. Summary of Nonuniformity Statistics Reported in Tables VI–XV

Statistic	Method			
	Prime Integer Division	Power-of-2 Integer Division	Cyclic Polynomials	Polynomial Division
Mean	−1.2	−2.8	−0.6	0.4
Standard Deviation	11.4	6.6	1.4	1.4
Minimum	−24.2	−24.2	−2.7	−2.5
Maximum	42.0	9.8	1.6	3.4

Each of these numbers was derived from 30 entries in the previous tables.

3.2 Time Tests

Hashing by recursion was motivated by a desire for rapid processing. It is appropriate, then, to show that recursion does result in a speed improvement over direct hashing. Also germane is a speed comparison of the recursive methods.

To study hashing speed, the author timed the period needed to hash every *n*-gram present in the English text sample described in the previous section. For this test, the hash values were not used, nor was any filtering done to ignore *n*-grams containing invalid characters. In this way, the test measured only the speed of hashing. Five methods of hashing were examined: four recursive methods and a nonrecursive standard. The nonrecur-

Table XVII. Nonuniformity Score U and Excess Fractional Work ω for Self-Annihilating Recursive Hashing by Cyclic Polynomials

r	n	Text Sample	$U(\omega)$
$1 + x^8$	4	English	4.6 (1.3%)
$1 + x^8$	4	Japanese	9.3 (1.8%)
$1 + x^4$	8	English	24.1 (1.9%)
$1 + x^4$	8	Japanese	9.3 (0.8%)

The radix, n-gram length, and text sample are as given. This data come from implementing the algorithm pictured in Table V. The excess work factor is in parentheses. The number of table buckets was 32K.

sive method was hashing by integer division, the most often used method of hashing.

Timing results are given in Table XVIII. Two n-gram lengths were tried. Note that a longer n-gram resulted in slower processing using the nonrecursive method, but did not change the speed of recursive methods. Most striking about the results is the promised speed improvement of recursion over the nonrecursive method, even when hashing by integer division.

Of particular interest is comparison with previously reported recursive hashing methods. The method of Karp and Rabin [1987], recursive prime integer division, is about 2.5 times slower than hashing by recursive cyclic polynomials. The faster method of Gonnet and Baeza-Yates [1990] using division by powers of 2 still takes about 50% longer than hashing by recursive cyclic polynomials.

Each of the tests of self-annihilating recursive hashing by cyclic polynomials reported in the previous section ran at a relative speed of 11.3. The work avoided by self-annihilation resulted in a 50% speed improvement.

4. DISCUSSION AND CONCLUSIONS

The use of recursion for computing each hash value from its predecessor offers great improvement in speed when hashing consecutive n-grams. This speed comes at virtually no expense: experiments on recursive algorithms indicate hashing uniformity on a par with the "ideal" of uniform hashing.

Applications of n-gram counting to language sorting, topic sorting, information storage and retrieval, string searching, and other indexing and comparison operations can benefit greatly from this speed improvement. Particularly affected are real-time interactive uses.

Several methods for recursive hashing are available. Of the four principal methods outlined here, hashing by prime integer division is the slowest; hashing by power-of-2 integer division is the next slowest. While power-of-2 integer division produced the best mean uniformity, the integer division methods are the most erratic in distribution, often giving the flattest distributions, but occasionally producing the poorest ones. Hashing by both cyclic polynomials and general polynomial division result in near-uniform distributions and fastest processing. Hashing by cyclic polynomials holds a slight edge in uniformity and in speed over polynomial division (7%),

Table XVIII. Relative Measured Speed Required to Hash 587,524 *n*-grams of English Text

Method	Relative Speed	
	5-grams	10-grams
Nonrecursive integer division	1.0	0.5
Recursive prime integer division	3.0	3.0
Recursive power-of-2 integer division	4.9	5.0
Recursive polynomial division	7.2	7.2
Recursive cyclic polynomials	7.7	7.7

The standard—nonrecursive integer division applied to 5-grams—took 6.56 seconds on a Macintosh Quadra 950.

though polynomial division is more general and is more suited to longer *n*-grams. For special cases, self-annihilating methods offer even more speed.

Recursive hashing by polynomial division or cyclic polynomials offers the most rapid processing, without sacrificing near-ideal hash value distribution.

In closing, one may note that recursive hashing may also be implemented inexpensively in hardware, especially by either of the polynomial methods.

APPENDIX

A. SOME MATHEMATICAL DEFINITIONS AND OBSERVATIONS

This appendix offers a few preliminaries for the reader who finds the mathematical terms in the body of the article to be unfamiliar.

A.1 Rings and Integral Domains

A ring R is a nonempty set of elements with two operations on those elements, denoted + and $\times$, which obey several properties. Each of the two operations maps any pair of set elements into an element of the set. The operation + is associative and commutative; the operation $\times$ is associative and distributes over +, that is, for any $a, b, c \in R$, $a \times (b + c) = a \times b + a \times c$ and $(b + c) \times a = b \times a + c \times a$. With respect to +, there is an identity element 0 such that $a + 0 = a$ for every $a \in R$. Further, each element $a \in R$ has an inverse $-a$ such that $a + (-a) = 0$. It follows from this definition that $a \times 0 = 0$ for every $a \in R$.

For the ring examples found in the body of this article, the operations + and $\times$ correspond to common addition and multiplication. The integers $\mathbf{Z}$ are a ring with respect to the usual addition and multiplication.

An integral domain is a commutative ring that possesses no zero divisors; that is, the multiplication operation is commutative, and the multiplication of two nonzero elements of the ring produces a nonzero result.

A.2 Fields

A field F is a special ring which possesses a multiplicative identity 1, is commutative with respect to multiplication, and has a multiplicative inverse for each nonzero element. In other words, for every $a, b \in F$, we have $a \times b = b \times a$ and $a \times 1 = a$; if $a \neq 0$, then there exists $a^{-1} \in F$ with $a \times a^{-1} = 1$. One of the simplest fields is the Galois field of two elements, denoted GF(2). Its elements may be labeled as 0 and 1; the operations are multiplication and addition defined by the tables

+	0	1
0	0	1
1	1	0

and

×	0	1
0	0	0
1	0	1

.

The definition of a field precludes the possibility of zero divisors, that is, it is not possible to find $a, b \in F$, $a \neq 0$, $b \neq 0$ with $a \times b = 0$. Otherwise, one could write $a = a \times 1 = a \times (b \times b^{-1}) = (a \times b) \times b^{-1} = 0 \times b^{-1} = 0$, a contradiction. Thus, a field is also an integral domain.

A.3 Cosets in Rings

Given a ring R and an element $r \in R$, one may form a new ring $R' = R/r$, in which the elements of R' are subsets of R. For each such subset, called a coset, its members differ by multiples of the "modulus" r. Consider, for example, $\mathbf{Z}/5$, the integers modulo 5. The coset containing 2 also contains 7, 102, and -3. The usual notation uses a bar over an element to represent the coset containing that element, so that the coset containing 2 can be indicated by $\bar{2}$. Note that $\bar{2} = \bar{7}$. One may get sloppy and drop the overbars, provided that no confusion results. Further, one may choose to represent a coset by the "smallest" member of that coset, provided that "smallest" makes sense in the ring at hand. In the body of the article, the author took the informal route, leaving out the coset notation and choosing to represent each coset by the smallest element.

A.4 Polynomial Rings

By adjoining the indeterminant x to a ring, one can form a ring of polynomials in x having coefficients in the original ring. (By "adjoining," one means to include all finite sums and products of the new quantity.) For example, $R = \mathrm{GF}(2)[x]$ is a new ring consisting of all polynomials in x whose coefficients are confined to 0 and 1. Elements of this ring are added and multiplied as polynomials normally are, with the computations of coefficients done according to the rules given above for GF(2) (the equivalent result is obtained by treating the coefficients as if they were integers, reducing the results modulo 2), e.g., $(1 + x + x^3) + (1 + x) = x^3$ and $(1 + x + x^3)(1 + x) = 1 + x^2 + x^3 + x^4$.

Given a polynomial $p \in R$, one can form a new ring $R' = R/p$ consisting of cosets of polynomials, with members of the same coset differing by multiples of p. Each coset is usually represented by its member of mini-

mum degree, so that each polynomial is represented by its residue modulo p. For example, the ring $GF(2)[x]/1 + x^2$ consists of four members (cosets): $\bar{0}$, $\bar{1}$, $\bar{x}$, and $\overline{1 + x}$. Note that in this ring $\overline{1 + x^2} = \bar{0}$. The overbars are usually dropped.

A.5 Implications of Reducibility for Hashing

If factorization in the ring R makes sense, one may talk about $r \in R$ being either irreducible or reducible (composite). If r is composite, then $R' = R/r$ cannot be an integral domain. For if r can be factored nontrivially as $r = ab$, then $\bar{a}$, $\bar{b} \in R'$ are such that $\bar{a} \times \bar{b} = 0$, that is, the ring has zero divisors. As examples, note that while $\mathbf{Z}/5$ is an integral domain, $\mathbf{Z}/6$ is not: $\bar{2} \times \bar{3} = \bar{0}$.

In each case presented in this article, keys are represented by elements of the ring R and are hashed by "reducing" them to elements of $R' = R/r$. Suppose that the keys are biased with respect to multiples of some ring element a; that is, suppose that there exists $a \in R$ such that the keys are distributed unequally between the sets $S_\Delta = \{am + \Delta \mid m \in R\}$. (For an integer example, suppose that there are very few keys of odd value. Then significantly more of the keys will be found in the set $S_0 = \{2m + 0 \mid m \in \mathbf{Z}\}$ than in $S_1 = \{2m + 1 \mid m \in \mathbf{Z}\}$.)

If it happens that a divides r, then for each Δ, all of the elements in S_Δ will be mapped to the same element in R'. Hence, a poor distribution of keys among the sets $\{S_\Delta\}$ can easily result in a bad distribution of hash values. (It will not do so always, since multiple sets hashing to each value may wash out such biases.)

To avoid such key biases being reflected in the hash value distribution, it is sufficient to insist that r be irreducible, that is, that R' be an integral domain.

In hashing using cyclic polynomials, it was pointed out that computation is done over a ring that is not an integral domain. It turns out, however, that this is not likely to be a problem for most implementations. In most cases, $w = 2^m$ for some m, so that $x^w + 1 = (x + 1)^w$, and so the only risk is that polynomials divisible by $x + 1$ will not be balanced by polynomials that are not divisible by $x + 1$. Each of these categories is half of the 2^w elements of $R' = \mathrm{GF}(2)[x]/(x^w + 1)$: the even density polynomials and the odd density polynomials, respectively. But in practice one will only use the lower $t < w$ bits (coefficients) for the hash value, and, over these bits, the two sets are indistinguishable.

ACKNOWLEDGMENTS

The idea of recursive hashing was suggested to the author by Jeff Kubina, as was a variant of cyclic polynomial hashing. The author would like to thank Jeff M. Kubina, Marc Damashek, Steve Huffman, Claudia Pearce, and Suzanne Banghart for reviewing the article.

REFERENCES

ANGELL, R. C., FREUND, G. E., AND WILLET, P. 1983. Automatic spelling correction using trigram similarity measure, *Inf. Process. Manage. 19,* 4, 255–261.

BARTON, I. J., CREASY, S. E., LYNCH, M. F., AND SNELL, M. J. 1974. An information-theoretic approach to text searching in direct access systems. *Commun. ACM 17,* 6 (June), 345–350.

BERLEKAMP, E. R. 1984. *Algebraic Coding Theory.* Aegean Park Press, Laguna Hills, Calif.

BICKEL, P. J. AND DOKSUM, K. A. 1977. *Mathematical Statistics.* Holden-Day, San Francisco, Calif.

BISHOP, Y. M. M., FIENBERG, S. E., AND HOLLAND, P. W. 1975. *Discrete Multivariate Analysis: Theory and Practice.* MIT Press, Cambridge, Mass.

BURNETT, J. E., COOPER, D., LYNCH, M. F., WILLETT, P., AND WYCHERLEY, M. 1979. Document retrieval experiments using indexing vocabularies of varying size. I. Variety generation symbols assigned to the fronts of index terms. *J. Doc. 35,* 3 (Sept.), 197–206.

CAVNAR, W. B. 1993. N-gram-based text filtering for TREC-2. In *Proceedings of TREC-2: Text Retrieval Conference 2,* D. Harman, Ed. National Bureau of Standards, Gaithersburg, Md.

COHEN, J. D. 1995. Highlights: Language- and domain-independent automatic indexing terms for abstracting. *J. Am. Soc. Inf. Sci. 46* (Apr.), 162–174. See vol. 47, issue 3, p. 260 for erratum.

CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms.* MIT Press, Cambridge, Mass.

CORNEW, R. W. 1968. A statistical method of spelling correction. *Inf. Control 12,* 79–93.

DAMASHEK, M. 1995. Gauging similarity with n-grams: Language-independent categorization of text. *Science 267,* 10 (Feb.), 843–848.

D'AMORE, R. J. AND MAH, C. P. 1985. One-time complete indexing of text: Theory and practice. 1985. In *Proceedings of the 8th International ACM Conference on Research and Development in Information Retrieval.* ACM, New York, 155–164.

DARRAGH, J. J., WITTEN, I. H., AND JAMES, M. L. 1990. The reactive keyboard: A predictive typing aid. *Computer 23,* 11 (Nov.), 41–49.

DE HEER, T. 1982. The application of the concept of homeosemy to natural language information retrieval. *Inf. Process. Manage. 18,* 5, 229–236.

DUMEY, A. I. 1956. Indexing for rapid random access memory systems. *Comput. Autom. 5,* 12 (Dec.), 6–9.

GONNET, G. H. AND BAEZA-YATES, R. A. 1990. An analysis of the Karp-Rabin string matching algorithm. *Inf. Process. Lett. 34,* 271–274.

GONNET, G. H. AND BAEZA-YATES, R. 1991. *Handbook of Algorithms and Data Structures in C and Pascal.* Addison-Wesley, Wokingham, U.K.

HANAN, M. AND PALERMO, F. P. 1963. An application of coding theory to a file address problem. *IBM J. Res. Devel. 7* (Apr.), 127–129.

HANSON, A. R., RISEMAN, E. M., AND FISHER, E. 1976. Context in word recognition. *Pattern Recog. 8,* 35–45.

HARRISON, M. C. 1971. Implementation of the substring test for hashing. *Commun. ACM 14,* 12 (Dec.), 777–779.

HULL, J. J. AND SRIHARI, S. N. 1982. Experiments in text recognition with binary *n*-gram and Viterbi algorithms. *IEEE Trans. Pattern Anal. Mach. Intell. PAMI-4,* 5 (Sept.), 520–530.

HUSSAIN, A. B. S. AND DONALDSON, R. W. 1974. Suboptimal sequential decision schemes with on-line feature ordering. *IEEE Trans. Comput. C-23,* 6, 582–590.

KARP, R. M. AND RABIN, M. O. 1987. Efficient randomized pattern-matching algorithms. *IBM J. Res. Devel. 31,* 2 (Mar.), 249–260.

KIM, J. Y. AND SHAWE-TAYLOR, J. 1992. An approximate string-matching algorithm. *Theoret. Comput. Sci. 92,* 107–117.

KNUTH, D. E. 1973. *The Art of Computer Programming.* Vol. 3, *Sorting and Searching.* Addison-Wesley, Reading, Mass.

KOTAMARTI, U. AND THARP, A. L. 1990. Accelerated text searching through signature trees. *J. Am. Soc. Inf. Sci. 41,* 2 (Mar.), 79–86.

KUKICH, K. 1992. Techniques for automatically correcting words in text. *ACM Comput. Surv. 24,* 4 (Dec.), 377–439.

LUM, V. Y., YUEN, P. S. T., AND DODD, M. 1971. Key-to-address transform techniques: A fundamental performance study on large existing formatted files. *Commun. ACM 14,* 4 (Apr.), 228–239.

LYNCH, M. F. 1977. Variety generation—A reinterpretation of Shannon's mathematical theory of communication, and its implications for information science. *J. Am. Soc. Inf. Sci. 28,* 1 (Jan.), 19–25.

MCELWAIN, C. K. AND EVENS, M. B. 1962. The degarbler—A program for correcting machine-read morse code. *Inf. Control 5,* 368–384.

MCKENZIE, B. J., HARRIES, R., AND BELL, T. 1990. Selecting a hashing algorithm. *Softw. Pract. Exper. 20,* 2 (Feb.), 209–224.

MORRIS, R. AND CHERRY, L. L. 1975. Computer detection of typographical errors. *IEEE Trans. Prof. Commun. PC-18* (Mar.), 54–64.

NAKAMURA, M. AND SHIKANO, K. 1989. A study of English word category prediction based on neural networks. In *ICASSP-89: International Conference on Acoustics, Speech, and Signal Processing*. Vol. 2. IEEE, New York, 731–734.

NEUHOFF, D. L. 1975. The Viterbi algorithm as an aid in text recognition. *IEEE Trans. Inf. Theory IT-21* (Mar.), 222–226.

PAESELER, A. AND NEY, H. 1989. Continuous-speech recognition using a stochastic language model. In *ICASSP-89: International Conference on Acoustics, Speech, and Signal Processing 2*. IEEE, New York, 719–722.

PEARCE, C. E. 1994. A dynamic hypertext environment through n-gram analysis. Ph.D. dissertation, Univ. of Maryland Baltimore County, Baltimore, Md.

PETERSON, W. W. 1957. Addressing for random-access storage. *IBM J. Res. Devel. 1* (Apr.), 130–146.

PETERSON, W. W. 1961. *Error-Correcting Codes.* John Wiley & Sons, New York.

PIETRA, S. D., PIETRA, V. D., MERCER, R. L., AND ROUKOS, S. 1992. Adaptive language modeling using minimum discriminant estimation. In *ICASSP-92: International Conference on Acoustics, Speech, and Signal Processing.* Vol. 1. IEEE, New York, 633–636.

PRYOR, D. V., THISTLE, M. R., AND SHIRAZI, N. 1993. Text searching on Splash 2. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines.* IEEE Computer Society Press, Los Alamitos, Calif., 172–177.

SCHAY, G. AND RAVER, N. 1963. A method for key-to-address transformation. *IBM J. Res. Devel. 7* (Apr.), 121–126.

SCHMITT, J. C. 1990. Trigram-based method of language identification. U.S. Patent No. 5,062,143. U.S. Patent Office, Washington, D.C.

SCHUEGRAF, E. J. AND HEAPS, H. S. 1973. Selection of equifrequent word fragments for information retrieval. *Inf. Storage Retrieval. 9,* 697–711.

SHANNON, C. E. 1951. Prediction and entropy of printed english. *Bell System Tech. J. 30* (Jan.), 50–64.

SHINGHAL, R., ROSENBERG, D., AND TOUSSAINT, G. T. 1978. A simplified heuristic version of a recursive Bayes algorithm for using context in text recognition. *IEEE Trans. Syst. Man Cyber. SMC-8,* 5 (May), 412–414.

SUEN, C. Y. 1979. n-gram statistics for natural language understanding and text processing. *IEEE Trans. Patt. Anal. Mach. Intell. PAMI-1,* 2 (Apr.), 164–172.

THOMAS, R. B. AND KASSLER, M. 1967. Character recognition in context. *Inf. Control 10,* 43–64.

UKKONEN, E. 1992. Approximate string-matching with q-grams and maximal matches. *Theor. Comput. Sci. 92,* 191–211.

VOSSLER, C. M. AND BRANSTON, N. M. 1964. The use of context for correcting garbled English text. In *Proceedings of the 19th National Conference of the ACM.* ACM, New York, D2.4-1–D2.4-13.

WILLET, P. 1979. Document retrieval experiments using indexing vocabularies of varying size. II. Hashing, truncation, digram and trigram encoding of index terms. *J. Doc. 35,* 4 (Dec.), 296–305.

WISNIEWSKI, J. L. 1987. Effective text compression with simultaneous digram and trigram encoding. *J. Inf. Sci. 13,* 159–164.

WRIGHT, J. H., JONES, G. J. F., AND WRIGLEY, E. N. 1992. Hybrid grammar-bigram speech recognition system with first-order dependence model. In *ICASSP-92: International Conference on Acoustics, Speech, and Signal Processing.* Vol. 1. IEEE, New York, 169–172.

YANNAKOUDAKIS, E. J. AND HUTTON, P. J. 1992. An assessment of n-phoneme statistics in phoneme guessing algorithms which aim to incorporate phonotactic constraints. *Speech Commun. 11,* 6 (Dec.), 581–602.

ZAMORA, E. M., POLLOCK, J. J., AND ZAMORA, A. 1981. The use of trigram analysis for spelling error detection. *Inf. Process. Manage. 17,* 6, 305–316.

Received April 1995; revised December 1995; accepted November 1996