Edgar H. Sibley
Panel Editor

Using only a few simple and commonplace instructions, this algorithm efficiently maps variable-length text strings onto small integers.

Fast Hashing of Variable-Length Text Strings

Peter K. Pearson

In the literature on hashing techniques, most authors spend little time discussing any particular hashing function, but make do with an allusion to Knuth [3] in their haste to get to the interesting topics of table organization and collision resolution. The relatively rare articles on hashing functions themselves [2] tend to discuss algorithms that operate on values of predetermined length or that make heavy use of operations (multiplication, division, or shifts of long bit strings) that are absent from the instruction sets of smaller microprocessors.

This article proposes a hashing function specifically tailored to variable-length text strings. This function takes as input a word W consisting of some number n of characters, $C_1, C_2, \ldots, C_n$, each character being represented by one byte, and returns an index in the range 0–255. An auxiliary table T of 256 randomish bytes is used in the process. Here is the proposed algorithm:[1]

```
h[0] := 0 ;
for i in 1..n loop
  h[i] := T[ h[i-1] xor C[i] ] ;
end loop ;
return h[n] ;
```

Notice that the processing of each additional character of text requires only an exclusive-OR operation and an indexed memory read. Also note that it is not necessary to know the length of the string at the beginning of the computation, a property useful when the end of the text string is indicated by a special character rather than by a separately stored length variable.

Two desirable properties of this algorithm for hashing variable-length strings derive from the technique of *cryptographic checksums* or *message authentication codes* [4], from which it is adapted. First, a good cryptographic checksum ensures that small changes to the data result in large and seemingly random changes to the checksum. In the hashing adaptation, this results in good separation of very similar strings. Second, on a good cryptographic checksum the effect of changing one part of the data must *not* be cancelled by an easily computed change to some other part. In hashing, this ensures good separation of anagrams, the downfall of hashing strategies that begin with a length-reducing exclusive-OR of substrings.

The auxiliary table T is obviously crucial to this algorithm, yet I have found very few constraints on its construction. Since the hashing function can only return values that appear in T, each index from 0 to 255 must appear in T exactly once. In other words, T must be a permutation of the values (0 . . . 255). Obviously, if T[i] = i, the corresponding h is merely a longitudinal exclusive-OR checksum, which is a bad hashing function because it does not separate anagrams. I have experimented by filling T with randomly generated permutations of (0 . . . 255) and have found no outstanding good or bad arrangements. (An attempt to promote greater dispersal among very similar short strings by clever choice of T, however, turned out to be a very bad idea.)

For the interested reader who does not want to generate his own random permutations, Table I presents the permutation used in the tests described later in this article.

SEPARATION PERFORMANCE

The purpose of any text hashing function is to take text strings—even very similar text strings—and map them onto integers that are spread as uniformly as possible over the intended range of output values. In the absence of prior knowledge about the strings being hashed, a perfectly uniform output distribution cannot be expected. The best result that one can expect to achieve consistently is a seemingly random mapping of input strings onto output values. To see how well h does its job, one might ask the following questions.

- If h is applied to a string of random bytes, is each of the 256 possible outcomes equally likely? The answer, probably not surprisingly, is yes. From the algorithm given earlier, it is clear that if the last input character, C[n], is *random*—equally likely to take any value, and uncorrelated with any preceding character—then all final values of h are equally likely.
- If two input strings differ by a single bit, will their hash function values collide more often than by

[1] In a practical implementation, the subscripts on h are omitted. They are shown here to clarify later discussion.

chance? The surprising answer here is that they will *never* collide. Two strings of the same length that differ in only one character cannot produce the same function value. The proof begins with this succession of observations:

(1) h xor c = h′ xor c if and only if h = h′.
(2) h xor c = h xor c′ if and only if c = c′.
(3) T[x] = T[x′] if and only if x = x′.
(4) If c ≠ c′, T[h xor c] ≠ T[h xor c′].
(5) If h ≠ h′, T[h xor c] ≠ T[h′ xor c].

If h[i] is the succession of values encountered during the hashing of some string C, and h′[i] that for string C′, the fourth observation above guarantees that the sequences h and h′ begin to differ at the point where C and C′ first differ. The last observation guarantees that they must continue to differ until a second pair of differing input characters is encountered.

- On a more practical level, if h is applied to a large number of real English words, are the 256 possible outcomes reasonably equally represented? Yes, h was applied to a spelling-checker dictionary with 26,662 words (many differing from others in appended "-s" or "-ed"), and the number of occurrences of each of the 256 output values was tallied. (All dictionary entries were in lowercase and appeared in alphabetical order.) Naturally, the resulting 256 counts were not all equal, but they are not alarmingly uneven. The traditional chi-square goodness-of-fit test asks how often a truly random function would be expected to produce a distribution at least as uneven as this, and the answer is, "About half the time" ($\chi^2 = 255.64$, 255 degrees of freedom, $p = 0.477$). To test for correlation between the hash values of adjacent words, a second test was run in which successive hash values were exclusive-ORed, and the resulting 26,661 values were tallied. The distribution of the 256 resulting tallies was notably uniform ($\chi^2 = 212.47$, 255 d.f., $p = 0.976$).

COMPARISON WITH A SIMPLE ALTERNATIVE

Although it seems to receive little attention in the literature, the following hashing algorithm is suspected of being widely used due to its speed and ease of implementation on small processors. (This is a special case of the addition option mentioned in [2, p.268]).

```
h[0] := 0 ;
for i in 1 .. n loop
  h[i] := (h[i − 1] + C[i]
  mod table_size ;
end loop ;
return h[n]
```

As before, the subscripts on h would be omitted in any practical implementation. If `table_size` is a power of two, the remainder can be extracted without a division operation. Also, if the processor can accommodate integers larger than any plausible sum of characters, the remainder can be taken just once, after all of the totals, instead of after each step.

Addition being commutative, this function will not separate anagrams, perhaps an inauspicious sign. The distribution of values returned by this function for the 26,662 dictionary entries mentioned demonstrated a significant deviation from uniformity ($\chi^2 = 468.9$, 255 d.f., $p < 0.001$).

On the other hand, in a test involving a smaller number of words, this additive hashing function performed about as well as h. In this test, 128 words were randomly selected from the 26,662-word dictionary, and collisions were counted among the resulting hash values. Over ten such trials, the mean number of collisions for the additive hashing function (28.2) was only slightly higher than for h (26.8). So the nonuniform distribution of the additive hashing function does not necessarily confer a large performance penalty.

VARIANTS

The following variations and extensions of this hashing scheme have been explored.

TABLE I. Pseudorandom Permutation of the Integers 0 through 255

1	87	49	12	176	178	102	166	121	193	6	84	249	230	44	163
14	197	213	181	161	85	218	80	64	239	24	226	236	142	38	200
110	177	104	103	141	253	255	50	77	101	81	18	45	96	31	222
25	107	190	70	86	237	240	34	72	242	20	214	244	227	149	235
97	234	57	22	60	250	82	175	208	5	127	199	111	62	135	248
174	169	211	58	66	154	106	195	245	171	17	187	182	179	0	243
132	56	148	75	128	133	158	100	130	126	91	13	153	246	216	219
119	68	223	78	83	88	201	99	122	11	92	32	136	114	52	10
138	30	48	183	156	35	61	26	143	74	251	94	129	162	63	152
170	7	115	167	241	206	3	150	55	59	151	220	90	53	23	131
125	173	15	238	79	95	89	16	105	137	225	224	217	160	37	123
118	73	2	157	46	116	9	145	134	228	207	212	202	215	69	229
27	188	67	124	168	252	42	4	29	108	21	247	19	205	39	203
233	40	186	147	198	192	155	33	164	191	98	204	165	180	117	76
140	36	210	172	41	54	159	8	185	232	113	196	231	47	146	120
51	65	28	144	254	221	93	189	194	139	112	43	71	109	184	209

This permutation gave a good hashing behavior.

Smaller Character Range

If the range of the input characters can be limited, a smaller auxiliary table T may be used, and the range of h can be limited accordingly. For example, if the input string can be limited to digits and uppercase letters, then each character can be mapped into the range [0, 63] as it is processed; T can be a table of 64 values in the range [0, 63], and h will then return a value in that range. For example, the 26,544 spelling-checker entries consisting entirely of letters and digits were hashed with a function that mapped the digits onto [0, 9] and both uppercase and lowercase alphabets onto [10, 35]. A 64-element T was built by eliminating all entries exceeding 63 from the table presented earlier. Distribution over the 64 output values was as even as would be expected from a random function ($\chi^2 = 59.17$, 63 degrees of freedom, $p = 0.614$), while the exclusive-ORs of successive values were insignificantly less uniform ($\chi^2 = 81.69$, 63 d.f., $p = 0.057$).

Larger Range of Hash Values

In some applications, a range of hash indices larger than 256 is needed. Here is a simple way to get 16 bits of hash index from the function h:

(1) Apply h to the string, calling the result H1.
(2) Add 1 (modulo 256) to the first character of the string.
(3) Apply h to the modified string to get H2.
(4) Concatenate H1 and H2 to get a 16-bit index.

When this algorithm was applied to the 26,662-word spelling-checker dictionary, 4,721 collisions occurred. Since perfectly random hashing would produce, on the average, 4,757 collisions, we conclude that this 16-bit extension of h performs essentially as well as random hashing.

In a second test, the 65,536 possible 16-bit index values were grouped and tallied in 533 bins. (The number of bins was chosen so that the average bin would catch about 50 of the 26,662 tallies.) The resulting distribution of tallies was consistent with the hypothesis of a uniform distribution ($\chi^2 = 558.6$, 532 d.f., $p = 0.205$).

Permuted Index Space

Some users of hashing functions who are concerned with collision handling prefer to think of the hashing function as producing a *permutation* of the index space, thereby specifying not just a single hash index, but a succession of hash indices to be tried in case of collisions [6]. The function h is well suited to this sort of application. By repeatedly incrementing the first character of the input string, modulo 256, one causes the hash index returned by h to pass through all 256 possible index values in a very irregular manner. This is derived from the assertion that strings of equal length differing in only one character cannot produce the same hashing function value.

Perfect Hashing

A hashing function is *perfect*, with respect to some list of words, if it maps the words in the list onto distinct values, that is, with no collisions. A perfect hashing function is *minimal* if the integers onto which that particular list of words is mapped form a contiguous set, that is, a set with no holes. (See, for example, [1], [5], and [3, pp. 506–507].) Minimal perfect hashing func-

TABLE II. A Permutation Demonstrating Perfect Hashing

39	159	180	252	71	6	13	164	232	35	226	155	98	120	154	69
157	24	137	29	147	78	121	85	112	8	248	130	55	117	190	160
176	131	228	64	211	106	38	27	140	30	88	210	227	104	84	77
75	107	169	138	195	184	70	90	61	166	7	244	165	108	219	51
9	139	209	40	31	202	58	179	116	33	207	146	76	60	242	124
254	197	80	167	153	145	129	233	132	48	246	86	156	177	36	187
45	1	96	18	19	62	185	234	99	16	218	95	128	224	123	253
42	109	4	247	72	5	151	136	0	152	148	127	204	133	17	14
182	217	54	199	119	174	82	57	215	41	114	208	206	110	239	23
189	15	3	22	188	79	113	172	28	2	222	21	251	225	237	105
102	32	56	181	126	83	230	53	158	52	59	213	118	100	67	142
220	170	144	115	205	26	125	168	249	66	175	97	255	92	229	91
214	236	178	243	46	44	201	250	135	186	150	221	163	216	162	43
11	101	34	37	194	25	50	12	87	198	173	240	193	171	143	231
111	141	191	103	74	245	223	20	161	235	122	63	89	149	73	238
134	68	93	183	241	81	196	49	192	65	212	94	203	10	200	47

1	a	9	for	17	in	25	the
2	and	10	from	18	is	26	this
3	are	11	had	19	it	27	to
4	as	12	have	20	not	28	was
5	at	13	he	21	of	29	which
6	be	14	her	22	on	30	with
7	but	15	his	23	or	31	you
8	by	16	i	24	that		

With this table, a minimal, perfect hashing function is constructed that produces the values shown for 31 common English words.

tions are useful in applications where a predetermined set of high-frequency words is expected and the hash value is to be used to index an array relating to those words. If the hashing function is minimal, no elements in the array are wasted (unused).

The table T at the heart of this new hashing function can sometimes be modified to produce a minimal, perfect hashing function over a modest list of words. In fact, one can usually choose the exact value of the function for a particular word. For example, Knuth [3] illustrates perfect hashing with an algorithm that maps a list of 31 common English words onto unique integers between -10 and 30. The table T presented in Table II maps these same 31 words onto the integers from 1 to 31, *in alphabetical order.*

Although the procedure for constructing the table in Table II is too involved to be detailed here, the following highlights will enable the interested reader to repeat the process.

(1) A table T was constructed by pseudorandom permutation of the integers (0 . . . 255).
(2) One by one, the desired values were assigned to the words in the list. Each assignment was effected by exchanging two elements in the table.
(3) For each word, the first candidate considered for exchange was T[h[n-1] xor C[n]]), the last table element referenced in the computation of the hash function for that word.
(4) A table element could not be exchanged if it was referenced during the hashing of a previously assigned word or if it was referenced earlier in the hashing of the same word.
(5) If the necessary exchange was forbidden by Rule 4, attention was shifted to the previously referenced table element, T[h[n-2] xor C[n-1]]).

This procedure is not always successful. For example, using the ASCII character codes, if the word "a" hashes to 0 and the word "i" hashes to 15, it turns out that the word "in" must hash to 0. Initial attempts to map Knuth's 31 words onto the integers (0 . . . 30) failed for exactly this reason. The shift to the range (1 . . . 31) was an ad hoc tactic to circumvent this problem.

Does this tampering with T damage the statistical behavior of the hashing function? Not seriously. When the 26,662 dictionary entries are hashed into 256 bins, the resulting distribution is still not significantly different from uniform ($\chi^2 = 266.03$, 255 d.f., $p = 0.30$). Hashing the 128 randomly selected dictionary words resulted in an average of 27.5 collisions versus 26.8 with the unmodified T. When this function is extended as described above to produce 16-bit hash indices, the same test produces a substantially greater number of collisions (4,870 versus 4,721 with the unmodified T), although the distribution still is not significantly different from uniform ($\chi^2 = 565.2$, 532 d.f., $p = 0.154$).

CONCLUSION

The main advantages of the hashing function presented here are:

(1) No restriction is placed on the length of the text string.
(2) The length of the text string does not need to be known beforehand.
(3) Very little arithmetic is performed on each character being hashed.
(4) Similar strings are not likely to collide.
(5) Minimal, perfect hashing functions can be built in this form.

Its principal disadvantages are:

(1) Output value ranges that are not powers of 2 are somewhat more complicated to provide.
(2) More auxiliary memory (the 256-byte table T) is required by this hashing function than by many traditional functions.

This hashing function is expected to be particularly useful in situations where good separation of similar words is needed, very limited instruction sets are available, or perfect hashing is desired.

REFERENCES

1. Cichelli, R. J. Minimal perfect hash functions made simple. *Commun. ACM 23*, 1 (Jan. 1980), 17.
2. Knott, G. D. Hashing functions. *Comput. J. 18*, 3 (1974). 265–278.
3. Knuth, D. E. *The Art of Computer Programming.* Vol. III, *Searching and Sorting.* Addison-Wesley, Reading, Mass., 1973.
4. Meyer, C., and Matyas, S. *Cryptography.* John Wiley & Sons, New York, 1982.
5. Sprugnoli, R. Perfect hashing functions: A single probe retrieving method for static sets. *Commun. ACM 20*, 11 (Nov. 1977), 841.
6. Ullman, J. D. A note on the efficiency of hashing functions. *J. ACM 19*, 3 (July 1972), 569–575.

CR Categories and Subject Descriptors: E.2 **[Data]**: Data Storage Representations—*hash-table representations*; H.3.1 **[Information Storage and Retrieval]**: Content Analysis and Indexing—*indexing methods*; H.3.3 **[Information Storage and Retrieval]**: Information Search and Retrieval—*search process.*

General Terms: Algorithms.

Additional Key Words and Phrases: Hashing, scatter storage.

ABOUT THE AUTHOR:

PETER PEARSON is a computer scientist at Lawrence Livermore National Laboratory, where his work tends to emphasize microcomputers, statistics, cryptology, and physics. Author's Present Address: 5624 Victoria Lane, Livermore, CA 94550