 doi:10.20944/preprints201710.0192.v1

Article

How to Hash a Set

Richard O'Keefe [1,†] *

[1] Computer Science, University of Otago; ok@cs.otago.ac.nz
† Current address: Computer Science, University of Otago

Abstract: Hash tables are widely used. They rely on good quality hash functions. Popular data structure libraries either provide no hash functions or weak hash functions for sets or maps, making it impossible or impractical to use them as keys in other tables. This article presents three algorithms for hashing a set, two of which are simple to implement, practically fast, and can be combined. The quality evaluations follow the method of [1, chapter 2]. The insight that we are looking for commutative semigroups suggests that even better methods than symmetric polynomials may be found.

Keywords: set; hash; commutative; semigroup

1. Introduction

Many programming languages these days have some sort of library offering a range of container data structures. Hash tables, whether for sets or dictionaries or both, are a popular offering. You can expect to find good support for strings as keys; you may hope to find good support for numbers and time-stamps and even for general sequences. What you are not likely to find is good support for sets themselves as elements of hashed sets or keys of hashed dictionaries.

The author has encountered this problem several times: a clause of propositional logic may be represented as a pair of disjoint sets of variables, an itemset in data mining [2] is precisely a set of items, and sets of variables are encountered in data flow analysis [3]. Counting or associating properties with such things requires hashing sets.

This paper describes three techniques for computing hash codes for unordered collections such as sets, bags, and dictionaries. They are (a) using or imposing a canonical order, (b) partitioning, and (c) using a symmetric polynomial to combine element hashes.

2. Current Practice

Major textbooks such as Cormen *et al.* [4], Knuth [5], and Sedgewick [6], have much to say about the theory of hash tables, about hash table data structures, and about techniques for computing hash codes of numbers and strings. Advice about how to construct hash functions for other data structures is rare, and for hashing sets nonexistent.

ANSI Common Lisp [7] simply doesn't offer a set data type to start with. Equality does not look inside hash tables, so the built-in `sxhash` function considers only the identity of a hash table.

Ada 2005 [8,9] provides sets and dictionaries, but does not provide any hash functions for them. If you want one, you must write it all yourself.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 31 October 2017 doi:10.20944/preprints201710.0192.v1

C# version 4 [10,11] has an ISet interface[1] which has nothing to say about the hash code for a set, while the use of exclusive or in all the examples in the GetHashCode page[2] is not encouraging.

Andrés Valloud's wonderful survey of hashing algorithms [1, section 6.2.5] despairingly remarks 'because no particular order of enumeration is guaranteed ... we will nto be able to do much better than to join hash values via a[n exclusive or] funnel".

Java[3], Python[4], GNU Smalltalk [12,13], Squeak Smalltalk [14,15], VisualWorks Smalltalk 7.10 and later [16], and Pharo Smalltalk [17,18] try harder.

All of the algorithms in this paper use the following identifiers:

s the set to be hashed;
h the hash code to be computed from s;
x an element of s;
a an accumulator or array of accumulators;
c computes a commencing value from the cardinality of s;
e a hash function for the elements of s, which is often a scrambled version of the hash function that is used to locate elements in s, but could be the same function or a completely unrelated one;
u the update function;
w a wrap-up function to compute h from a.

When a function identifier appears in a cost formula, it stands for the cost of a call to that function, assumed to be independent of its arguments.

The languages above use variants of the following scheme:

Space: $O(1)$
Time: $O(|s|(e+u)+c+w)$
$a \leftarrow c(|s|)$;
for $x \in s$ **do**
  $a \leftarrow u(a, e(x))$;
end
$h \leftarrow w(a)$;

where u must be commutative and associative. That is, the domain of a (typically 0–2^m-1 for some m) and the function u form a commutative semigroup. There are two popular choices for u: bitwise exclusive or (used by Python and some Smalltalk systems) and sum (used by Java).

These hashes have the potentially useful property that the hash value (more precisely, a rather than h) can be incrementally maintained as elements are added to and removed from s. Incrementally addition places no extra constraints on u; incremental removal needs an inverse where $(\forall x)(\forall y)u^{-1}(u(x,y),y) = x$.

	$u(x,y) = x+y$	$u(x,y) = x \oplus y$
add x	$a \leftarrow a + e(x)$	$a \leftarrow a \oplus e(x)$
remove x	$a \leftarrow a - e(x)$	$a \leftarrow a \oplus e(x)$

They also have the quality-related property that if x and y are uniformly distributed over $[0, 2^n)$ then so are $x \oplus y$ and $x + y$.

As the Spartans replied to Phillip II, "if". The main problem with this approach is that the distribution of element hash values is so often *not* uniform. If $c(n) = 0$, as it is in Java and some of the Smalltalks, and if $e(i) = i$ when i is a small integer, as it is in all the listed systems except GNU Smalltalk and Python, we find that $\{1,2\}$ and $\{3\}$ have the same hash value whether u is $+$ or $\oplus$. This can be improved by using a non-trivial c such as $\alpha + \beta n$, but even then $\{1,6\}$ and $\{2,5\}$ have the same

[1] http://msdn.microsoft.com/en-us/library/dd412081.aspx
[2] http://msdn.microsoft.com/en-us/library/system.object.gethashcode.aspx
[3] https://docs.oracle.com/javase/8/docs/api/java/util/AbstractSet.html
[4] setobject.c in https://www.python.org/download/releases/2.7.2/

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 31 October 2017 doi:10.20944/preprints201710.0192.v1

hash value. This is why Python and GNU Smalltalk include a "scrambling" stage in their e calculations. That helps a great deal, but is not a complete solution, hence this paper.

2.1. Do we need something better?

A colleague who read a draft of this paper asked the obvious question: if major programming languages hash sets so badly, and there is no outcry, is this problem really of practical interest?

First, there is a vicious circle. If major languages and libraries hash sets badly or not at all, programmers quickly learn not to do that, and that is then taken as evidence that the status quo is fine.

Second, it is not just sets that are hashed badly. Consider four classes in Java:

- Point2D has the property that (x, y) and $(-x, -y)$ always have the same hash code.
- String uses $(\Sigma_{i=0}^{n-1} \chi[i] \times 31^{n-1-i}) \bmod 2^{32}$. This is very weak. Consider all identifiers of the form `[a-zA-Z][a-zA-Z0-9_]*`. There are 3276 such identifiers. There are 480 triples of such identifiers with the same hash. That is, about 44% of the identifiers are in triples with the same hash. There are a further 790 pairs (1580 identifiers) that have the same hash. Only about 12% of the identifiers get distinct hash codes. Robert Jenkins' hash [19] gets *no* collisions for this data set.
- ArrayList uses the same method as String. As a test case that should be easy for a hash function to discriminate, construct all triples (a, b, c) with $0 \leq a, b, c < 64$ as ArrayLists and tally their hash codes. The collision chains come in three sizes:

62	have length	64
1831	have length	128
124	have length	192

  The constant 31 was chosen for speed, not for quality. A bigger constant would do better.
- HashMap.Node computes the same hash value for the maplets $x \mapsto y$ and $y \mapsto x$ for no apparent reason. This means that if you represent the edges of a graph by a HashMap⟨Node,Node⟩, a graph and its converse will have the same hash value.

As Valloud [1, section 5.1.14] puts it: in Java "it is *expected* that hash functions will be of bad quality, and that instead of developers addressing this situation, what will happen is that applications will spend computing resources calculating bad quality hash value regardless. Then, an *attempt* will be made to fix this issue by mixing the bits of the hash values... No amount of mixing in any amount of dimensions will solve this fundamental collision problem after the collisions are allowed to occur!"

If programmers are willing to tolerate low quality hash functions for simple data structures, it is not surprising that they have needlessly tolerated low quality hash functions for sets and maps.

3. Algorithms

3.1. Canonical order

Some kinds of sets, such as Java's BitSet, EnumSet, and TreeSet store their elements in a canonical order, so that it is possible to hash them as if they were sequences. This is current practice when applicable. Java 1.8 does this for BitSet but not for TreeSet. Any comparison-based data structure such as search trees, jump lists, or skip lists, which allows the elements to be traversed in a canonical order in linear time, can be hashed using sequence hashing techniques, with low overhead.

Hash tables do not make such a traversal easy, nor is there any universal total order we could use for generic sets. But we *can* convert the hashes of the elements to a canonical sequence by sorting them.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 31 October 2017 doi:10.20944/preprints201710.0192.v1

Since the element hash values are bounded integers, we can do this using a radix sort [20,21], which has linear worst-case time.

Space: $O(|s|)$
Time: $O(|s|e + w)$
int $a[|s|]$;
int $i \leftarrow 0$;
for $x \in s$ **do**
    $a[i] \leftarrow e(x)$;
    $i \leftarrow i + 1$;
end
sort(a);
$h \leftarrow w(a)$;

Typically, the cost of w will be linear in the size of a, so the overheads are linear in $|s|$. It is unusual for a hashing function to require this much workspace, and the constant factor of the sorting algorithm is not small. So we may take this as a benchmark for *quality*, and look for a related but efficient approach.

3.2. Partitioning

Bucket sort works by partitioning the input into buckets and then sorting the buckets. If the number of buckets is small, and they are not recursively sorted, then we get the following algorithm

Space: $O(B)$
Time: $O(|s|(e + u) + Bc + w)$
int $a[B]$;
for $i \in [0, B)$ **do** $a[i] \leftarrow c_i(|s|)$;
;
for $x \in s$ **do**
    $t \leftarrow e(x)$;
    $i \leftarrow t \bmod B$;
    $a[i] \leftarrow u_i(a[i], \lfloor t/B \rfloor)$;
end
$h \leftarrow w(|s|, a)$;

where each u_i is commutative and associative. The wrap-up function w can be any function of $B + 1$ integers. One thing it should not be is $a[0] \oplus \cdots \oplus a[B-1]$. It will normally take $O(B)$ time.

This method supports incremental revision (of the elements of a) if and only if each u_i does.

3.3. Symmetric Polynomial

We can generalise the Java/Smalltalk approach another way, by looking for another u.

The symmetric functions of two Boolean variables are 0, 1, $x \wedge y$, $\neg x \wedge \neg y$, $x \vee y$, $\neg x \vee \neg y$, $x \oplus y$, and $x \equiv y$. Of these, only $\oplus$ and its near-equivalent $\equiv$ are plausible. To go beyond this requires bit-oriented operations that mix up entire words, such as rotations.

Generalising $+$ is more promising. Instead of looking at just $x + y$, we can look for symmetric polynomials in two variables with integer coefficients satisfying $u(x, u(y, z)) = u(u(x, y), z)$, and use that for u in the Java method.

The simplest family that works is $u(x, y) = p + q(x + y) + rxy$. This is symmetric by construction, and a little algebra shows that it is associative if and only if $pr = q(q - 1)$.[5] Higher degree polynomials do not work.

[5] This was confirmed using SageMath [22].

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 31 October 2017 doi:10.20944/preprints201710.0192.v1

Revising a when an element is added is obvious. To revise a when an element x is removed, let $y = e(x)$. We need to solve $a = u(a', y) = (p + qy) + (q + ry)a'$ or $a' = (q + ry)^{-1}(a - p - qy)$. This has a solution if and only if $q + ry$ has an inverse modulo 2^n, which it does provided $q + ry$ is odd. The inverse can then be found using the Extended Euclidean Algorithm[6]. But $q + ry$ is odd for all y if and only if q is odd and r is even.

This condition is also enough to ensure that if x and y are uniformly distributed over $[0, 2^n - 1)$, so is $p + q(x + y) + rxy \bmod 2^n$. Suppose $u(x, y) = u(x, y')$. Then $(p + qx) + (q + rx)y = (p + qx) + (q + rx)y'$ or $(q + rx)(y - y') = 0$. But with q odd and r even, $q + rx$ has an inverse, so $y = y'$.

This method can be combined with partitioning; each of the partitions may use different values for (p, q, r).

4. Results

4.1. Speed

Five methods were written in C and compiled at optimisation level 2 using clang Apple LLVM version 9.0.0 (clang-900.0.38), and run on a mid-2015 MacBook Pro with 2.2 Ghz intel Core i7 processor under the macOS Sierra 1012.6 operating system. An array of 1,000,000 random 32-bit integers was hashed 10,000 times using each method and the times reported. In each case y is the hash code of the new element, which is just that element itself. The times are averages per iteration of the inner loop. To put this in context, the time to hash 146,522 words from a Scrabble dictionary using Robert J. Jenkins Jr.'s 32-bit "newhash" function [19] was 30.51 ns per word.

time	loop body
0.12 ns	`h ^= y`
0.24 ns	`h += y`
0.82 ns	`a[y&3] ^= y>>2`
1.65 ns	`a[y&3] += y>>2`
2.28 ns	`h = 3860031 + (h+y)*2779 + (h*y*2)`

4.2. Quality

Six different set hash functions were implemented:

- Sum(1): the Java method where the element hashes are simply summed.
- Sum(4): the radix sort-inspired method where the bottom 2 bits of each element hash selects an accumulator, the remaining bits are summed into that accumulator, and at the end the accumulators and set size are combined.
- Xor(1): the Smalltalk method where the element hashes are combined using exclusive or.
- Xor(4): the radix sort-inspired method where the bottom 2 bits of each element hash selects an accumulator, the remaining bits are xor-ed into that accumulator, and at the end the accumulators and set size are combined.
- Sort: the element hashes are sorted and then combined as if they were the element hashes of a sequence.
- Fold: the symmetric polynomial method with $(p, q, r) = (3860031, 2779, 2)$. Apart from $pr = q(q - 1)$, p odd, q odd, r even, and $\gcd(p, r)$=1, there was nothing special about them. Perhaps there are additional criteria that could be used to select better parameters. Several different parameter sets were tried, all giving similar results.

The evaluation method follows Valloud [1, chapter 2].

Table 1 considers all 16,384 subsets of the integers $\{1, 2, \ldots, 14\}$. #hash values is the number of distinct hash values. Collision rate is the number of items divided by the number of distinct hash

[6] I am grateful to Professor Mike Atkinson for reminding me of this.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 31 October 2017 doi:10.20944/preprints201710.0192.v1

Table 1. All 16384 subsets of $\{1, 2, \ldots, 14\}$

metric	Sum(1)	Sum(4)	Xor(1)	Xor(4)	Sort	Fold
#hash values	106	6076	16	2176	16384	16384
Collision rate	154.57	2.70	1024.00	7.53	1.00	1.00
Quality	0.65%	37.08%	0.10%	13.28%	100.00%	100.00%
Longest chain	397	20	1024	20	1	1
Mean chain	285.612	4.233	1024.000	11.175	1.000	1.000
χ^2	283.619	2.604	1022.001	9.308	0.000	0.000
Average χ^2	283.619	2.764	1022.001	9.458	0.140	0.142

Table 2. All 16384 subsets of $\{1.0, 2.0, \ldots, 14.0\}$

metric	Sum(1)	Sum(4)	Xor(1)	Xor(4)	Sort	Fold
#hash values	2152	9600	128	3760	16384	16384
Collision rate	7.61	1.71	128.00	4.36	1.00	1.00
Quality	13.13%	58.59%	0.78%	22.95%	100.00%	100.00%
Longest chain	55	8	128	14	1	1
Mean chain	20.271	2.426	128.000	5.914	1.000	1.000
χ^2	18.402	1.012	126.008	4.144	0.000	0.000
Average χ^2	19.163	1.225	126.008	4.311	0.142	0.141

values; it is the mean number of items hashing to the same value. Quality is the inverse of collection rate. Longest chain is the size of the largest group of items with the same hash value. Mean chain is the average size of a group of items with the same hash value, averaged over all the items. This is usually greater than the collision rate, because a group of n items is included n times, one for each item. χ^2 is the usual chi-squared measure for a uniform distribution: if it is small, the items are well spread out; if it is large, they are not.

All of those metrics assume an infinite table, so that the only collisions are those intrinsic to the hash function. χ^2_p computes the χ^2 metric for the hash values taken modulo a prime p; average χ^2 reports the average χ^2_p for 25 primes greater than or equal to the number of items. This shows how well the hash function would do for finite tables.

Xor(1) is amazingly bad, and Sum(1) little better, despite the element hash values being unique. Sum(4) and even Xor(4) are much better.

Table 2 concerns all 16384 subsets of the floats $\{1.0, 2.0, \ldots, 14.0\}$. The hashing method is limited by its need to deliver 30-bit results that are the same whether a number is single precision, double precision, or extended precision. The hash values for 1.0 to 14.0 are (in hexadecimal): 18006101 18006202 1C007202 18006303 1A006B03 1C007303 1E007B03 18006404 19006004 1A006C04 1B006804 1C007404 1D007004 1E007C04.

For these sets, Xor(1) was very bad, and Sum(1) was not much better. It is not surprising that Fold is good and Sort excellent; what is surprising is how much the multi-accumulator technique improves even Xor(4).

Table 3 concerns 1000 sets, each containing 20 randomly chosen 3-letter strings. No two sets were the same.

Table 3. 1000 sets of three-letter strings

metric	Sum(1)	Sum(4)	Xor(1)	Xor(4)	Sort	Fold
#hash values	1000	1000	1000	1000	1000	1000
Collision rate	1.00	1.00	1.00	1.00	1.00	1.00
Quality	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
Longest chain	1	1	1	1	1	1
Mean chain	1.000	1.000	1.000	1.000	1.000	1.000
χ^2	0.000	0.000	0.000	0.000	0.000	0.000
Average χ^2	0.145	0.137	0.144	0.150	0.145	0.149

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 31 October 2017 doi:10.20944/preprints201710.0192.v1

Table 4. All 16384 subsets of {16-Feb-2000,...,29-Feb-2000}

metric	Sum(1)	Sum(4)	Xor(1)	Xor(4)	Sort	Fold
#hash values	470	14400	32	9728	16384	16384
Collision rate	34.86	1.14	512.00	1.68	1.00	1.00
Quality	2.87%	87.89%	0.20%	59.38%	100.00%	100.00%
Longest chain	169	4	512	4	1	1
Mean chain	88.481	1.266	512.000	2.023	1.000	1.000
χ^2	86.510	0.145	510.002	0.617	0.000	0.000
Average χ^2	86.547	0.287	510.002	0.803	0.134	0.142

Table 5. 121181 words as sets of letters

metric	Sum(1)	Sum(4)	Xor(1)	Xor(4)	Sort	Fold
#hash values	1366	28441	128	12531	121165	121158
Collision rate	88.71	4.26	946.73	9.67	1.00	1.00
Quality	1.13%	23.47%	0.11%	10.34%	99.99%	99.99%
Longest chain	515	49	1620	62	2	2
Mean chain	241.886	10.746	1337.335	24.857	1.000	1.000
χ^2	239.897	8.981	1335.336	22.960	0.000	0.000
Average χ^2	239.897	9.146	1335.336	23.130	0.144	0.143

This shows that the more complex methods at any rate do little or no harm, and that sometimes even Xor(1) gets lucky.

Table 4 concerns all subsets of the 14 dates 16-Feb-2000 to 29-Feb-2000 inclusive. We can expect the hash functions for these dates to be simply related, so we expect Sum(1) and Xor(1) to do badly. They do.

Table 5 is for every line of /usr/share/dict/words on a Mac OS X 10.6.8 system converted to a set of letters, with duplicate sets discarded before the test. Since the characters in these words are mostly lower case letters, this should be not unlike the number and date tests. It is. These are small sets of letters, and so were the sets in table 3, yet although Sum(1) and Xor(1) did well in that test, they do very badly in this.

5. Discussion

Hash functions for sets based on exclusive or and modular addition are demonstrably weak. It is possible to do substantially better with very little coding effort and low overhead.

In the partitioning technique, the partition count B is a tradeoff between quality and time. $B = 8$ may be a good compromise.

The only guidelines for choosing the coefficients p, q, r as yet are to choose odd q and even r, and to prefer larger values to get better "mixing" of the hash values. A systematic way to choose these coefficients would be useful.

Commutative semigroups are abundant. There is no reason to believe that the symmetric polynomial used here is optimal. Much exploration remains to be done.

1. Valloud, A. *Hashing in Smalltalk: Theory and Practice*; Lulu, 2008.
2. Agrawal, R.; Imielinski, T.; Swami, A. Mining Association Rules between Sets of Items in Large Databases. Proceedings of the 1993 ACM SIGMOD Conference. ACM Press, 1993.
3. Khedker, U.; Sanyal, A.; Karkare, B. *Data Flow Analysis: Theory and Practice*; CRC Press (Taylor and Francis Group), 2009.
4. Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. *Introduction to Algorithms*, third edition ed.; MIT Press: Cambridge, Massachussets, 2009.

5. Knuth, D.E. *The Art of Computer Programming, Volume 3: Sorting and Searching*, second edition ed.; Addison-Wesley Professional: Boston, 1998.
6. Sedgewick, R. *Algorithms in C, Parts 1–5: Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms*, 3rd ed.; Addison-Wesley Professional: Boston, 2001.
7. Graham, P. *ANSI Common Lisp*; Prentice Hall, 1995.
8. ISO, www.iso.org. *ISO/IEC 8652:1995): Information Technology — Programming Languages — Ada, as updated by changes from Technical Corrigendum 1 (ISO/IEC 8652:1995:TC1:2000), and Amendment 1 (ISO/IEC 8526:AMD1:2007)*, 2007.
9. Ada-Europe. *Ada 2005 Language Reference Manual*, 2006. [Online; accessed 6-Dec-2011].
10. Nagle, C.; Evjen, B.; Glynn, J.; Watson, K.; Skinner, M. *Professional C♯ 4.0 and .NET 4*; Wrox Press, 2010.
11. Albahari, J.; Albahari, B. *C♯ 4.0 in a Nutshell: the Definitive Reference*, fourth ed.; O'Reilly Media, 2010.
12. Bonzini, P. GNU Smalltalk 3.2.91. http://smalltalk.gnu.org, 2015. [Online, accessed 11 October 2017].
13. Gökel, C. *Computer Programming using GNU Smalltalk*; Lulu, 2009.
14. Kay, A.; Ingalls, D.; Kaehler, T.; Wallace, S.; Maloney, J.; Raab, A.; Rueger, M. Squeak Smalltalk. http://squeak.org/, 2011. [Online, accessed 12 December 2011].
15. Black, A.P.; Ducasse, S.; Nierstrasz, O.; Pollet, D. *Squeak by Example*; Lulu, 2009.
16. Cincom. VisualWorks public user licence 7.10.1. http://www.cincomsmalltalk.com/main/products/visualworks/, 2014. [Online, accessed 13 October 2014].
17. Denker, M.; Ducasse, S.; Lienhard, A. Pharo Smalltalk. http://www.pharo-project.org/home, 2011. [Online, accessed 12 December 2011].
18. Black, A.P.; Ducasse, S.; Nierstrasz, O.; Pollet, D. *Pharo by Example*; Lulu, 2009.
19. Robert J. Jenkins, J. Hash Functions for Hash Table Lookup. http://burtleburtle.net/bob/hash/evahash.html, 1995–1997. [Online, last accessed 11 October 2017].
20. Hildebrandt, P.; Isbitz, H. Radix Exchange—An Internal Sorting Method for Digital Computers. *Journal of the ACM* **1959**, *6*, 156–163.
21. McIlroy, P.M.; Bostic, K.; McIlroy, M.D. Engineering Radix Sort. *Computing Systems* **1993**, *6*, 5–27.
22. Developers, T.S. *SageMath, the Sage Mathematics Software System (Version 7.5.1)*, 2017. http://www.sagemath.org.