# Universal Classes of Hash Functions

(extended abstract)

J. Lawrence Carter and Mark N. Wegman

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

**Abstract:**

This paper gives an *input independent* average linear time algorithm for storage and retrieval on keys. The algorithm makes a random choice of hash function from a suitable class of hash functions. Given any sequence of inputs the expected time (averaging over all functions in the class) to store and retrieve elements is linear in the length of the sequence. The number of references to the data base required by the algorithm for any input is extremely close to the theoretical minimum for any possible hash function with randomly distributed inputs. We present three suitable classes of hash functions which also may be evaluated rapidly. The ability to analyze the cost of storage and retrieval without worrying about the distribution of the input allows as corollaries improvements on the bounds of several algorithms.

**Introduction:**

One may view different inputs to a program as elements from a class of problems. The answer given by the program is, hopefully, a correct solution to the problem. Ordinarily, when one talks about the average performance of a program, one averages over the class of problems the program can solve. Gill[2], Rabin[7], Strassen and Solovay[9] have used a different approach on some classes of problems. They suggest that the program randomly choose an algorithm from the class of algorithms to solve the problem. They are able to bound the average performance of the class of algorithms for the worst case input. This average on the worst case can be better than the performance of any known single algorithm on its worst case. Some of the problems which this approach overcomes are the following:

1) Classical analysis (averaging over the class of inputs) must make assumptions about the distribution of the inputs. These assumptions may not hold in certain applications.

2) A consequence of (1) is that one cannot classically analyze the average performance of a subroutine independently of the main routine, since the main routine may skew the distribution of data.

3) If the program is presented with a worst-case input, there is no way to avoid the resulting poor performance. However, if one had a class of algorithms to choose from and was able to realize that a particular algorithm was running slowly on a given input, then it might be possible to choose a different algorithm.

In this paper, we apply these notions to hashing for storage and retrieval, and suggest that a class of hash functions be used. We show that if the class of functions is chosen properly, then the average performance of the program on *any* input will be nearly as good as if a single function, chosen with knowledge of the input, were used. We present several classes of hash functions which insure that every sample chosen from the input space will be distributed evenly by enough of the functions to

compensate for the poor performance of the algorithm when an unlucky choice of function is made.

A brief outline of our paper follows. After introducing some notation, we define a property of a class of functions: universal$_2$. We show that any class of functions that is universal$_2$ has the property that given any sample, a randomly chosen member of that class will be expected to distribute the sample evenly. We then exhibit several universal$_2$ classes of functions which can be evaluated easily. Finally we give several examples of the use of these functions.

**Notation:**

If S is a set, $|S|$ will denote the number of elements in S. $\lceil x \rceil$ means the least integer $\geq x$. If x and y are bit strings, then $x \oplus y$ is the exclusive-or of x and y. $Z_n$ will represent the integers mod n. All hash functions map a set A into a set B. We will always assume $|A| > |B|$. A is sometimes called the set of possible keys, and B the set of indices. If f is a hash function and $x,y \in A$, we let $\delta_f(x,y)$ be 1 if $x \neq y$ and $f(x) = f(y)$, and 0 otherwise. Thus, $\delta_f(x,y)$ is 1 if x and y are distinct elements of A which map to the same value under f. If f, x or y is replaced in $\delta_f(x,y)$ by a set, we sum over all the elements in the set. Thus, if H is a collection of hash functions, $x \in A$ and $S \subset A$ then $\delta_H(x,S)$ means

$$\sum_{f \in H} \sum_{y \in S} \delta_f(x,y).$$

Notice that the order of summation does not matter.

**Properties of Universal Classes:**

Let H be a class of functions from A to B. We say that H is *universal$_2$* if for all x,y in A, $\delta_H(x,y) \leq \frac{|H|}{|B|}$. That is, H is universal$_2$ if no pair of distinct keys are mapped into the same index by more than one $|B|^{th}$ of the functions. Proposition 1 shows that this bound on $\delta_H(x,y)$ is tight when $|A|$ is much larger than $|B|$. The second proposition follows almost immediately from the definition of universal$_2$.

Proposition 1: Given any collection H of hash functions (not necessarily universal$_2$), there exists $x,y \in A$ such that

$$\delta_H(x,y) > \frac{|H|}{|B|} - \frac{|H|}{|A|}.$$

Proof (sketch): Let $a = |A|$ and $b = |B|$. A counting argument shows that for each $f \in H$,
$\delta_f(A,A) \geq \frac{a^2}{b} - a.$
Thus, $\delta_H(A,A) \geq a^2|H|(1/b - 1/a)$. Therefore, by the pigeon hole principle, there exists $x,y \in A$ such that $\delta_H(x,y) > |H|(1/b - 1/a)$. □

Proposition 2: Let x be any element of A and S any subset of A. Let f be a function chosen randomly from a universal$_2$ class of functions (with equal probabilities on the functions.) Then the expected number of elements of S that x collides with, i.e. $\delta_f(x,S)$, is $\leq \frac{|S|}{|B|}$.
Proof:
Mean value of $\delta_f(x,S)$

$$= \frac{1}{|H|} \sum_{f \in H} \delta_f(x,S)$$
$$= \frac{1}{|H|} \sum_{y \in S} \delta_H(x,y) \text{ (by notation)}$$
$$\leq \frac{1}{|H|} \sum_{y \in S} \frac{|H|}{|B|} \text{ (by def. of universal}_2\text{)}$$
$$= \frac{|S|}{|B|}. \quad \square$$

In addition to being useful later, Proposition 2 has some direct applications. For instance, an optical character reader postprocessing system is described in [8]. This system is designed to check if a word x is a member of a set of valid words S. The set $\{ f(y) \mid y \in S \}$ is stored in memory. To test whether x is in S, a check is made to see if f(x) is in the stored set. Since f(y) is generally shorter than y, a considerable amount of space was saved. However, there is a chance of error; if $f(x)=f(y)$ and $y \in S$, then x may erroneously be accepted as valid. Proposition 2 gives a bound on the probability of error when f is chosen from a class of universal$_2$ functions.

We are interested in the cost of using these functions in storage and retrieval operations. Given a sequence R of requests (insertions or retrievals) to some data base, and a hash function f, we define the cost of R with respect to f, C(f,R), to be the sum of the costs of the individual requests. The cost of an individual request referring to an element x is one plus the number of distinct previously inserted y's for which f(x) = f(y).

This cost function reflects the worst case cost of inserting or finding elements in a storage and retrieval scheme in which each element of B is associated with a linked list, and an element x is stored in the list associated with f(x) (see [1], page 111 - 113.) Other collision resolution schemes would have other cost functions associated with them. For example, if the keys with the same index were stored in a balanced tree, the corresponding cost function would be smaller.

The following theorem gives a nice bound on the expected cost of using a universal$_2$ class of hash functions with the linked list method for resolving collisions.

Proposition 3: Let R be a sequence of r requests which includes k insertions. Suppose H is a universal$_2$ class of hash functions. Then if we choose f at random from H, the expected cost C(f,R) is $\leq r(1 + \frac{k}{|B|})$.

Proof: The expected cost of R is the sum of the expected costs of the individual requests. Proposition 2 and the definition of cost tell us that an individual request has expected cost no greater than $1 + \frac{k}{|B|}$. □

A special case of this proposition is that if k is roughly the size of B then the expected cost is 2r. Notice that this linear bound holds for any sequence of requests, not just for the "average" request. For many applications, there is an upper bound on the number of elements to be stored and hence, B can be chosen appropriately. If there is no known upper bound it is possible to dynamically choose a size, and rehash when that choice proves to be too small. Rehashing can be done in linear time, and even in real-time.

We can show that the expected cost (averaging over the hash functions) of any request is virtually the same as the expected cost (averaging over the possible requests) of *any* single hash function when applied to a random request after random insertions have been made. The reason is as follows: Let a = |A| and b = |B|. The counting argument cited in proposition 1 implies that if f is any hash function and x and y are chosen at random from A, then the expectation of $\delta_f(x,y)$ is $\geq (1/b - 1/a)$. It follows that the expectation of $\delta_f(x,S)$ is $\geq |S|\cdot(1/b - 1/a)$, where S is the random subset of A which has been previously stored. Thus, the cost of the request is at least $1 + |S|\cdot(1/b - 1/a)$. When A is much larger than B (which will be the case in most applications of hashing), this is virtually the same as the cost of a request cited in the proof of Proposition 3.

It is also possible to bound the probability that given a sequence of requests R, the performance of a randomly chosen function will be worse than tolerable on R. Since we know that C(f,R) must be at least r, we can conclude that when k is roughly the size of |B|, the probability that C(f,R) > t·r is less than 1/(t-1). For some classes of hash functions (such as the last class mentioned in this paper), it is possible to derive a bound on the standard deviation or higher moments of the cost of a randomly chosen function on a particular R. This allows us to get a much better estimate of the probability that C(f,R) will be large.

**Some universal$_2$ classes:**

The first class of universal$_2$ hash functions we present is suitable for applications where the bit

strings which represent the keys can conveniently be multiplied by the computer.

Suppose $A=\{0,1,...,a-1\}$ and $B=\{0,1,...,b-1\}$. Let p be a prime with $p \geq a$. Let g be any function from $Z_p$ to B which, as closely as possible, maps the same number of elements of $Z_p$ into each element of B. Formally, we require $| \{y \epsilon Z_p \mid g(y)=z\} | \leq \lceil \frac{p}{b} \rceil$ for all $z \epsilon B$. A natural choice for g is the residue modulo b. When $b = 2^k$ for some k, this amounts to taking the last k bits in the binary representation of y.

Let m and n be elements of $Z_p$ with $m \neq 0$. We define $h_{m,n}: A \rightarrow Z_p$ by $h_{m,n}(x) = (mx+n) \bmod p$. Now define $f_{m,n}(x) = g(h_{m,n}(x))$. The class H is the set $\{f_{m,n} \mid m,n \epsilon Z_p \quad m \neq 0\}$. If desired, p can be chosen so the mod p operation can be calculated without a division.

The following lemma is useful in proving that this class is universal$_2$.

Lemma: When H is defined as above, then for any $x,y \epsilon A$ with $x \neq y$, $\delta_H(x,y)$ equals the number of ordered pairs (r,s) with $r,s \epsilon Z_p$, $r \neq s$ and $g(r)=g(s)$.

Proof: There is a natural correspondence between the functions $h_{m,n}$ and the ordered pairs (r,s) where $r,s \epsilon Z_p$ and $r \neq s$. Specifically, we identify the function $h_{m,n}$ by the ordered pair $(h_{m,n}(x),h_{m,n}(y))$. Since $m \neq 0$, $h_{m,n}(x) \neq h_{m,n}(y)$. This correspondence is one-to-one and onto since the linear equations $xm+n \equiv r \pmod p$ and $ym+n \equiv s \pmod p$ have a unique solution for m and n in the field $Z_p$.

If (r,s) is the pair $(h_{m,n}(x),h_{m,n}(y))$, then $f_{m,n}(x)=f_{m,n}(y)$ if and only if $g(r)=g(s)$. Thus, $\delta_H(x,y)$ is the number of such pairs. □

Proposition 4: The class H defined above is universal$_2$.

Proof: Let $n_i$ be the number of elements in $\{ t \epsilon Z_p \mid g(t)=i \}$. The restriction on g is that for each i, $n_i \leq \lceil \frac{p}{b} \rceil$. Since p and b are integers, this implies that $n_i \leq \frac{p-1}{b}+1$. Thus, for a given r, the number of choices for s such that $r \neq s$ but $g(r)=g(s)$ is $\leq \frac{p-1}{b}$. Since there are p choices for r, $p \cdot \frac{p-1}{b} \geq$ the number of ordered pairs (r,s) satisfying the condition in the lemma $= \delta_H(x,y)$. Recalling that for $x=y$, $\delta_H(x,y)=0$, this shows that H is universal$_2$. □

Frequently, algorithms are analyzed making the assumption that multiplication takes unit time. The number of multiplications is said to be the cost of the algorithm. This model is appropriate when there are no operations which can be done an unbounded number of times for each multiplication. When a universal$_2$ class of hash functions is used, then the number of memory references per request can be bounded when averaged over all functions in the class as in Proposition 3 (assuming k is not unbounded with respect to $|B|$.) Therefore the model is appropriate, and under it the hash functions in the class given above may be applied in constant time. Thus in this model, for any sequence of requests, the algorithm takes average time linear in the number of requests.

It may seem that the addition of n in the class of functions given above plays an unimportant role. This is only partly true. Suppose for $m \epsilon Z_p$ we define $h_m(x) = (mx) \bmod p$, and as before define $f_m(x)$ as $g(h_m(x))$. Let $H = \{f_m \mid m \epsilon Z_p \quad m \neq 0\}$. It can be shown that this class of functions comes within a factor of two of being universal$_2$, that is, for any x and y, $\delta_H(x,y) \leq 2 \cdot \frac{|H|}{|B|}$. On the other hand, this bound cannot be improved significantly. For instance, let $b = |B|$, and choose k so that $p = kb+k+1$ is prime (there will be infinitely many such k's.) Let g(x) be the function x (mod b). Let $x = 1$ and $y = b+1$. Then the 2k functions $f_1$, $f_2$, ..., $f_k$, $f_{p-k}$, $f_{p-k+1}$, ..., $f_{p-1}$ each map x and y to the same bin. Thus, $\delta_H(x,y) = 2k$ while $\frac{|H|}{|B|} = \frac{p-1}{b} = \frac{kb+k}{b} = (1+\frac{1}{b})k$.

The universal$_2$ class of functions given above

may not be convenient when the keys are too long to be multiplied using a single machine instruction. However, the next proposition gives a method of extending a class of functions for long keys.

Proposition 4: Suppose $|B|$ is a power of two and H is a class of functions from A to B with the property that for each $i \varepsilon B$,
$|\{f \varepsilon H \mid f(x) \oplus f(y) = i\}| = \frac{|H|}{|B|}$. Recall that $\oplus$ is exclusive-or. Then we can define a universal$_2$ class of hash function from $A \times A$ to B as follows. For $f,g \varepsilon H$, define $h_{f,g}((x,y)) = f(x) \oplus g(y)$. Then this new collection of hash functions $J = \{h_{f,g} \mid f,g \varepsilon H\}$ is universal$_2$ and also satisfies the condition of this proposition.

The proof is quite similar to the proof of Proposition 6, and therefore omitted. Proposition 5 can be applied repeatedly to extend the functions to arbitrarily long keys. If the functions in H can be applied in constant time, then the time required to compute an extended function is proportional to the length of the key.

The proposition does not quite apply to the universal$_2$ class of functions given earlier both because H is not a power of 2 (so $|H|/|B|$ cannot be an integer) and because the number of functions for which $f(x) \oplus f(y) = 0$, i.e. $\delta_H(x,y)$, is actually less than $|H|/|B|$. Both of these differences add small factors to $\delta_J(x,y)$ which barely prevent J from being universal$_2$. The percentage contributed by these small factors decrease asymptotically to 0 as p is increased. More details will be given in a future paper.

The following is a class of functions which do not require multiplication, which may be better for many applications. Suppose A can be viewed as the set of i-digit numbers written in base $\alpha$, and B as the set of binary numbers of length j. Then $|A| = \alpha^i$ and $|B| = 2^j$. Let M be the class of arrays of length $i\alpha$, each of whose elements are bit strings of length j. For $m \varepsilon M$, let m(k) be the bit string which is the $k^{th}$ element of m, and for $x \varepsilon A$, let $x_k$ be the $k^{th}$ digit of x written in base $\alpha$. We define
$f_m(x) = m(x_1+1) \oplus m(x_1+x_2+2) \oplus \ldots \oplus m(\sum_{k=1}^{i} x_k+k)$.
The class H is the set $\{f_m \mid m \varepsilon M\}$.

Another way of defining this class is to give a program for applying a function to an input x.

```
dcl m(iα) bit(j) init(random);
dcl x(i) digits base α;
dcl value bit(j);
disp := 0;
value := 0;
for k := 1 to i do begin
    disp := disp + x(k) + 1;
    value := value ⊕ m(disp);
    end;
return (value);
```

Proposition 6: The class H defined above is universal$_2$.

Proof: For x and y in A, suppose $f_m(x)$ is the exclusive-or of the rows $r_1, r_2, \ldots, r_s$ of m, and $f_m(y)$ is $r_{s+1} \oplus \ldots \oplus r_t$. Notice that $f_m(x) = f_m(y)$ if and only if $r_1 \oplus \ldots \oplus r_t = 0$. Assuming $x \neq y$, there will be some k such that $r_k$ is involved in the calculation of only one of $f_m(x)$ or $f_m(y)$. Then $f_m(x)$ will equal $f_m(y)$ if and only if $r_k$ is the exclusive-or of the other $r_i$'s. Since there are $2^j = B$ possibilities for that row, x and y will collide for one $B^{th}$ of the possible functions $f_m$. Thus, the class of all $f_m$'s is universal$_2$. □

For a given B, each function in H takes time linear in the length of the key. In addition, we can more accurately describe the distribution of costs of a particular sequence of requests under the different functions. For instance, Markowsky [6] has shown that for any sequence R of r requests with k insertions, and any positive t, the probability that $C(f,R) - r(1+\frac{k}{|B|}) > r \cdot t$ is less than $\frac{k}{t^2|B|}$ and also less than $\frac{7k}{t^3|B|}$.

**Importance:**

Programmers sometimes spend a considerable amount of time refining hash functions for applications where it is critical that a uniform distribution be achieved ([5],p.508-513). This may be difficult because it is necessary that the expected input set not be biased in such a way as to make the hash function perform poorly. One of the practical values of a class of $universal_2$ functions is that we know that there are many acceptable functions in the class. Simply choosing a single hash function randomly from such a class gives a high expectation that a uniform distribution will be achieved. Furthermore, if the function is changed each time the program is run, then we can be sure of good performance averaged over all runs.

The theoretical importance is that it allows one to get a good bound on the average performance of an algorithm which uses hashing. The problem with an ordinary hashing scheme is that the algorithm might bias the information being stored and retrieved towards those cases that are distributed unevenly by the particular hash function being used.

Rabin has developed an algorithm which finds the nearest neighbors of a collection of points in a plane, given the coordinates of the points [7]. This algorithm involves making a random choice of points, and it uses hashing. If one also randomly chooses the hash function from a $universal_2$ class, then the expected running time of the algorithm will always be linear the number of points.

In [3] and [4] an algorithm is suggested for multiplying sparse polynomials, using hashing. We can strengthen the results of these papers. Let two polynomials, P and Q, have n and m non-zero terms respectively. If multiplies and adds are viewed as taking constant time, then given any two polynomials P and Q, we can multiply them in average time $O(n \cdot m)$. Let $CP_1, CP_2, \ldots, CP_n$ be the coefficients of the n terms of P. Let $EP_1, EP_2, \ldots, EP_n$ be the exponents of those terms. Let $CQ_i$ and $EQ_i$ stand for the same quantities of Q. The following algorithm will have the performance we suggested, assuming Store and Retrieve are implemented using a $universal_2$ class of hash functions. Their first argument is a key, and the second is the value stored or retrieved. If a value has not been stored previously for a given key, a retrieve will have a zero value.

```
Begin
   Choose a hash function;
   For i := 1 to n do
      For j := 1 to m do
            Begin
            Coefficient := CP_i * CQ_j;
            Retrieve (EP_i + EQ_j,k);
            Store (EP_i + EQ_j,Coefficient + k);
            End;
Print all keys and values which have been stored;
End;
```

Since addition and multiplication are viewed as taking constant time, the first class of functions we presented would seem appropriate for this analysis.

**Future research:**

There are a number of areas which can be investigated, such as:

1) Improve the bounds cited here on the probability that a particular function from the table look-up class will perform poorly on a particular input.

2) Suppose the store and retrieve algorithm is changed so that the last element retrieved is moved to the first position on the list. Can a smaller bound on the probability that the cost function exceeds a specified value be derived?

3) Extend the analysis to other storage and retrieval algorithms which involve hashing, such as double hashing and open addressing.

4) Generalize the definition of $universal_2$ to $universal_n$ to consider the action of the class of functions on any collection of n elements of A. Determine if one can obtain improved results with such

a stronger assumption. (One definition of universal$_k$ implies that the expected number of keys from a k-element set mapping into a given element of B would be binomially distributed.)

5) When should one decide that a particular function is a poor choice and it would be worth the effort to choose a new function and rehash?

**Acknowledgements:**

We would like to thank Ashok Chandra for helping suggest and formulate the problem; Dave Glickman for suggesting we examine the table look-up technique; Hania Gajewska and David Smith for suggesting revisions in an earlier manuscript; Walter Rosenbaum for discussions about a practical use of this work; and George Markowsky for help in understanding the distribution of performance of the class obtained by table look-up.

**References:**

[1] Aho, A.V., Hopcroft, J.E., and Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass. (1974).

[2] Gill, J.T., III, Computational Complexity of Probabilistic Turing Machines, *Proceedings of the Sixth ACM Symposium on the Theory of Computing*, May, 1976, Seattle, Washington, p. 91-95.

[3] Goto, E., and Kanada, Y., Hashing Lemmas on Time Complexities with Applications to Formula Manipulation, *Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation*, August, 1976, Yorktown Heights, New York, p.149-153.

[4] Gustavson, F., and Yun, D.Y.Y., Arithmetic Complexity of Unordered or Sparse Polynomials *Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation*, August, 1976, Yorktown Heights, New York, p.154-159.

[5] Knuth, D.E., *Sorting and Searching*, Addison-Wesley, Reading, Mass. (1973).

[6] Markowsky, G., private communication.

[7] Rabin, M.O., Probabilistic algorithms, *Proceedings of Symposium on New Directions and Recent Results in Algorithms and Complexity* (1976 - to appear).

[8] Rosenbaum, W.S., and Hilliard, J.J., Multifont OCR postprocessing system. *IBM Journal of Research and Development* (July 1975).

[9] Strassen, V., and Solovay, R., A fast Monté-Carlo test for primality, *SIAM Journal on Computing* (to appear).