Fast Hashing

Mikkel Thorup

May 15, 2014

1 Hash functions

The concept of truly independent hash functions is extremely useful in the design of randomized algorithms. We have a large universe U of keys, e.g., 64-bit numbers, that we wish to map randomly to a range $[m] = \{0, ..., m-1\}$ of hash values. A *truly random hash function* $h : U \to [m]$ assigns an independent uniformly random variable $h(x)$ to each key in x. The function h is thus a $|U|$-dimensional random variable, picked uniformly at random among all functions from U to $[m]$. When h has been picked, it is no longer random, so $h(x)$ is fixed for all keys $x \in U$.

Unfortunately truly random hash functions are idealized objects that cannot be implemented. More precisely, to represent a truly random hash function, we need to store at least $|U| \log_2 m$ bits, and in most applications of hash functions, the whole point in hashing is that the universe is much too large for such a representation (at least not in fast internal memory).

The idea is to let hash functions contain only a small element or seed of randomness so that the hash function is sufficiently random for the desired application, yet so that the seed is small enough that we can store it when first it is fixed. In these notes we will discuss some basic forms of random hashing that are very efficient to implement, and yet have sufficient randomness for some very important applications.

2 Universal hashing

The concept of universal hashing was introduced by Carter and Wegman in [2]. We wish to generate a random hash function $h : U \to [m]$ from a key universe U to a set of hash values $[m] = \{0, ..., m-1\}$. We think of h as a random variable following some distribution over functions $U \to [m]$. We want h to be *universal* which means that for any given distinct keys $x, y \in U$, when h is picked at random (independently of x and y), we have *low collision probability*:

$$\Pr_h[h(x) = h(y)] \leq 1/m.$$

For many applications, it suffices if for some $c = O(1)$, we have

$$\Pr_h[h(x) = h(y)] \leq c/m.$$

Then h is called *c-universal*.

In this chapter we will first give some concrete applications of universal hashing. Next we will show how to implement universal hashing when the key universe is an integer domain $U = [u] = \{0, ..., u-1\}$ where the integers fit in, say, a machine word, that is, $u \leq 2^w$ where $w = 64$ is the word length. In later chapters we will show how to make efficient universal hashing for large objects such as vectors and variable length strings.

2.1 Applications

One of the most classic application of universal hashing is *simple hash tables with chaining*. We have a set $S \subseteq U$ of keys that we wish to store so that we can find any key from S in expected constant time. Let $n = |S|$ and $m \geq n$. We now pick a universal hash function $h : U \to [m]$, and then create an array L of m lists/chains so that for $i \in [m]$, $L[i]$ is the list of keys that hash to i. Now to find out if a key $x \in U$ is in S, we only have to check if x is in the list $L[h(x)]$. This takes time proportional to $1 + |L[h(x)]|$ (we add 1 because it takes constant time to look up the list even if turns out to be empty).

If $x \notin S$ and h is universal, then the expected number of elements in $L[h(x)]$ is

$$E_h[|L[h(x)]|] = \sum_{y \in S} \Pr_h[h(y) = h(x)] = n/m \leq 1.$$

The first equality uses *linearity of expectation*.

Exercise 2.1 *(a) What is the expected number of elements in $L[h(x)]$ if $x \in S$?*

(b) What bound do you get if h is only 2-universal?

The idea of hash tables goes back to [7], and hash tables were the prime motivation for introduction of universal hashing in [2]. For a text book description, see, e.g., [3, §11.2].

A different applications is that of assigning a unique *signature* $s(x)$ to each key. Thus we want $s(x) \neq s(y)$ for all distinct keys $x, y \in S$. To get this, we pick a universal hash function $s : U \to [n^3]$. The probability of an error (collision) is calculated as

$$\Pr_s[\exists\{x, y\} \subseteq S : s(x) = s(y)] \leq \sum_{\{x,y\} \subseteq S} \Pr_s[s(x) = s(y)] = \binom{n}{2}/n^3 < 1/(2n).$$

The first inequality is a *union bound*: that the probability of that at least one of multiple events happen is at most the sum of their probabilities.

The idea of signatures is particularly relevant when the keys are large, e.g., a key could be a whole text document, which then becomes identified by the small signature. This idea could also be used in connection with hash tables, letting the list $L[i]$ store the signatures $s(x)$ of the keys that hash to i. To check if x is in the table we check if $s(x)$ is in $L[i]$.

Exercise 2.2 *With $s : U \to [n^3]$ and $h : U \to [n]$ independent universal hash functions, for a given $x \in U \setminus S$, what is the probability of a* false positive *when we search x, that is, what is the probability that there is a key $y \in S$ such that $h(y) = h(x)$ and $s(y) = s(x)$?*

2.2 Multiply-mod-prime

Below we now study implementations of the time it takes

Note that if $m \geq u$, we can just let h be the identity (no randomness needed) so we may assume that $m < u$.

The classic universal hash function from [2] is based on a prime number $p \geq u$. We pick a uniformly random $a \in [p]_+ = \{1, ..., p-1\}$ and $b \in [p] = \{0, ..., p-1\}$, and define $h_{a,b} : [u] \to [m]$ by

$$h_{a,b}(x) = ((ax + b) \bmod p) \bmod m) \tag{1}$$

Given any distinct $x, y \in [u] \subseteq [p]$, we want to argue that for random a and b that

$$\Pr_{a \in [p]_+,\, b \in [p]} [h_{a,b}(x) = h_{a,b}(y)] \leq 1/m. \tag{2}$$

In most of our proof, we will consider all $a \in [p]$, including $a = 0$. Ruling out $a = 0$, will only be used in the end to get the tight bound from (2).

We need only one basic fact about primes:

Fact 2.1 *If p is prime and $\alpha,\ \beta \in [p]_+$ then $\alpha\beta \not\equiv 0 \pmod p$.*

For given pair $(a, b) \in [p]^2$, define $(q, r) \in [p]^2$ by

$$\begin{aligned} ax + b \bmod p &= q & (3)\\ ay + b \bmod p &= r. & (4) \end{aligned}$$

Lemma 2.2 *Equations (3) and (4) define a 1-1 correspondence between pairs $(a, b) \in [p]^2$ and pairs $(r, q) \in [p]^2$.*

Proof For a given pair $(r, q) \in [p]^2$, we will show that there is at most one pair $(a, b) \in [p]^2$ satisfying (3) and (4). Subtracting (3) from (4) modulo p, we get

$$(ay + b) - (ax + b) \equiv a(y - x) \equiv q - r \pmod p, \tag{5}$$

We claim that there is at most one a satisfying (5). Suppose there is another a' satisfying (5). Subtracting the equations with a and a', we get

$$(a - a')(y - x) \equiv 0 \pmod p,$$

but since $a - a'$ and $y - x$ are both non-zero modulo p, this contradicts Fact 2.1. There is thus at most one a satisfying (5) for given (r, q). With this a, we need b to satisfy (3), and this determines b as

$$b = ax - q \bmod p. \tag{6}$$

Thus, satisfying (3) and (4), each pair $(q, r) \in [p]^2$ thus corresponds to at most one pair $(a, b) \in [p]^2$. On the other hand, (3) and (4) define a unique pair $(q, r) \in [p]^2$ for each pair $(a, b) \in [p]^2$. We have p^2 pairs of each kind, so the correspondence must be 1-1. ■

Since $x \neq y$, by Fact 2.1,

$$r = q \iff a = 0. \tag{7}$$

Thus, when we pick $(a, b) \in [p]_+ \times [p]$, we get $r \neq q$.

Returning to the proof of (2), we get a collision $h_{a,b}(x) = h_{a,b}(y)$ if and only if $q \equiv r \pmod{m}$, and we know that $q \neq r$. For given r, there are at most $\lceil p/m \rceil$ values of q with $q \equiv r \pmod{m}$; namely $r, r+m, r+2m, \ldots$. Ruling out $q = r$ leaves us at most $\lceil p/m \rceil - 1$ values of q for each of the p values of r. Noting that $\lceil p/m \rceil - 1 \leq (p+m-1)/m - 1 = (p-1)/m$, we get that the total number of collision pairs (r, q), $r \neq q$, is bounded by most $p(p-1)/m$. Our 1-1 correspondence means that there are at most $p(p-1)/m$ collision pairs $(a, b) \in [p]_+ \times [p]$. Since each of the $p(p-1)$ pairs from $[p]_+ \times [p]$ are equally likely, we conclude that the collision probability is bounded by $1/m$, as required for universality.

Exercise 2.3 *Suppose we for our hash function also consider $a = 0$, that is, for random $(a, b) \in [p]^2$, we define the hash function $h_{a,b} : [p] \to [m]$ by*

$$h_{a,b}(x) = ((ax + b) \bmod p) \bmod m.$$

(a) Show that this function may not be universal.

(b) Prove that it is always 2-universal, that is, for any distinct $x, y \in [p]$,

$$\Pr_{(a,b)\in[p]^2}[h_{a,b}(x) = h_{a,b}(y)] < 2/m.$$

2.2.1 Implementation for 64-bit keys

Let us now consider the implementation of our hashing scheme

$$h(x) = ((ax) \bmod p) \bmod m)$$

for the typical case of 64-bit keys in a standard imperative programming language such as C. Let's say the hash values are 20 bits, so we have $u = 2^{64}$ and $m = 2^{20}$.

Since $p > u = 2^{64}$, we generally need to reserve more than 64 bits for $a \in [p]_+$, so the product ax has more than 128 bits. To compute ax, we now have the issue that multiplication of w-bit numbers automatically discards overflow, returning only the w least significant bits of the product. However, we can get the product of 32-bit numbers, representing them as 64-bit numbers, and getting the full 64-bit result. We need at least 6 such 64-bit multiplications to compute ax.

Next issue is, how do we compute $ax \bmod p$? For 64-bit numbers we have a general mod-operation, though it is rather slow, and here we have more than 128 bits.

An idea from [2] is to let p be a Mersenne prime, that is, a prime of the form $2^q - 1$. We can here use the Mersenne prime $2^{89} - 1$. The point in using a Mersenne prime is that

$$x \equiv x \bmod 2^q + \lfloor x/2^q \rfloor \pmod{p}.$$

Question 2.4 *Assuming a language like C supporting 64-bit multiplication, addition, shifts and bit-wise Boolean operations, but no general mod-operation, sketch the code to compute $(ax \bmod p) \bmod m$. Recall that m is power of two.*

2.3 Multiply-shift

We shall now turn to a truly practical universal hashing scheme proposed by Dietzfelbinger et al. [6], yet ignored by most text books. It generally addresses hashing from w-bit integers to ℓ-bit integers. We pick a uniformly random odd w-bit integer a, and then we compute $h_a : [2^w] \to [2^d]$, as

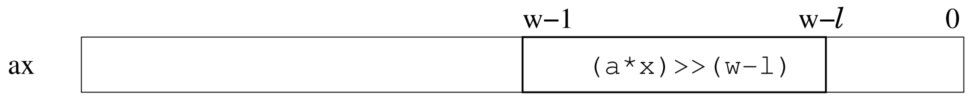
$$h_a(x) = \lfloor (ax \bmod 2^w)/2^{w-\ell} \rfloor \tag{8}$$

This scheme gains an order of magnitude in speed over the scheme from (1), exploiting operations that are fast on standard computers. Numbers are stored as bit strings, with the least significant bit to the right. Integer division by a power of two is thus accomplished by a right shift. For hashing 64-bit integers, we further exploit that 64-bit multiplication automatically discards overflow, which is the same as multiplying modulo 2^{64}. Thus we end up the following C-code:

```
#include <stdint.h> //defines uint64_t as unsigned 64-bit integer.
uint64_t hash(uint64_t x; uint64_t l; uint64_t a) {
// hashes x universally into l bits using the random odd seed a.
  return (a*x) >> (64-l);}
```

This scheme is many times faster and simpler to implement than the standard multiply-mod-prime scheme, but the analysis is a bit more subtle.

It is convenient to think of the bits of a number as indexed with bit 0 the least significant bit. The scheme is simply extracting bits $w - \ell, ..., w - 1$ from the product ax, as illustrated below.



We will prove that multiply-shift is 2-universal, that is,

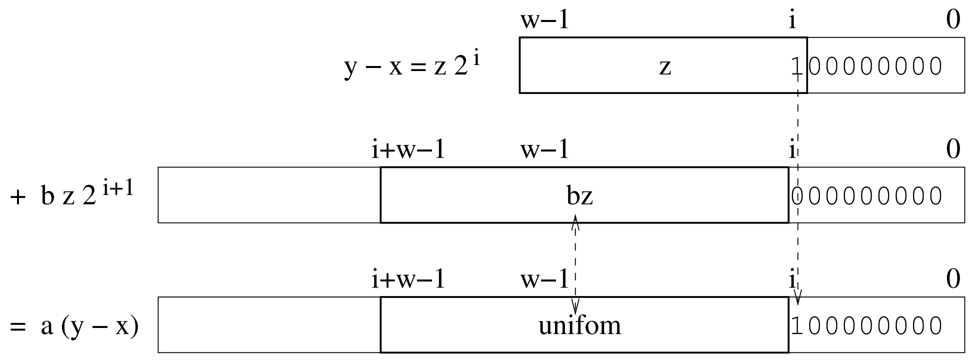
$$\Pr_{\text{odd } a \in [2^w]} [h_a(x) = h_a(y)] \leq 2/2^\ell = O(1/m). \tag{9}$$

We will exploit that any odd number z is relatively prime to any power of two:

Fact 2.3 *If α is odd and $\beta \in [2^q]_+$ then $\alpha\beta \not\equiv 0 \pmod{2^q}$.*

Define b such that $a = 1 + 2b$. Then b is uniformly distributed in $[2^{w-1}]$. Moreover, define z to be the odd number satisfying $(y - x) = z2^i$. Then

$$a(y - x) = z2^i + bz2^{i+1}.$$



Now $bz \bmod 2^{w-1}$ must be uniformly distributed in $[2^{w-1}]$, for if there was another $b' \in [2^{w-1}]$ with $b'z \equiv bz \pmod{2^{w-1}} \iff z(b'-b) \equiv 0 \pmod{2^{w-1}}$, then this would contradict Fact 2.3 since z is odd. There is therefore a 1-1 correspondence between the $b \in [2^{w-1}]$ and the products $bz \bmod 2^{w-1}$. The uniform distribution on b thus implies that $bz \bmod 2^{w-1}$ is uniformly distributed. We conclude that $a(y-x) = z2^i + bz2^{i+1}$ has `0` in bits $0, ..., i-1$, `1` in bit i, and a uniform distribution on bits $i+1, ..., i+w-1$.

We have a collision $h_a(x) = h_a(y)$ if ax and $ay = ax + a(y-x)$ are identical on bits $w-\ell, ..., w-1$. The two are always different in bit i, so if $i \geq w-\ell$, we have $h_a(x) \neq h_a(y)$ regardless of a. However, if $i < w-\ell$, then because of carries, we could have $h_a(x) = h_a(y)$ if bits $w-\ell, ..., w-1$ of $a(y-x)$ are either all `0`s, or all `1`s. Because of the uniform distribution, either event happens with probability $1/2^\ell$, for a combined probability bounded by $2/2^\ell$. This completes the proof of (9).

Question 2.5 *Why is it important that a is odd? Hint: consider the case where x and y differ only in the most significant bit.*

3 Strong universality

We will now consider *strong universality* [11]. For $h : [u] \to [m]$, we consider *pair-wise events* of the form that for given distinct keys $x, y \in [u]$ and possibly non-distinct hash values $q, r \in [m]$, we have $h(x) = q$ and $h(y) = r$. We say a random hash function $h : [u] \to [m]$ is *strongly universal* if the probability of every pair-wise event is $1/m^2$. We note that if h is strongly universal, it is also universal since

$$\Pr[h(x) = h(y)] = \sum_{q \in [m]} \Pr[h(x) = q \wedge h(y) = q] = m/m^2 = 1/m.$$

Observation 3.1 *An equivalent definition of strong universality is that each key is hashed uniformly into $[m]$, and that distinct keys are hashed independently.*

Proof First assuming strong universality and consider distinct keys $x, y \in U$. For any hash value $q \in [m]$, $\Pr[h(x) = q] = \sum_{r \in [m]} \Pr[h(x) = q \wedge h(y) = r] = m/m^2 = 1/m$, so $h(x)$ is uniform in $[m]$, and the same holds for $h(y)$. Moreover, for any hash value $r \in [m]$,

$$\begin{aligned}\Pr[h(x) = q \mid h(y) = r] &= \Pr[h(x) = q \wedge h(y) = r] / \Pr[h(y) = r] \\ &= (1/m^2)/(1/m) = 1/m = \Pr[h(x) = q],\end{aligned}$$

so $h(x)$ is independent of $h(y)$. For the converse direction, when $h(x)$ and $h(y)$ are independent, $\Pr[h(x) = q \wedge h(y) = r] = \Pr[h(x) = q] \cdot \Pr[h(y) = r]$, and when $h(x)$ and $h(y)$ are uniform, $\Pr[h(x) = q] = \Pr[h(y) = r] = 1/m$, so $\Pr[h(x) = q] \cdot \Pr[h(y) = r] = 1/m^2$. ■

Emphasizing the independence, strong universality is also called *2-independence*.

As for universality, we may accept some relaxed notion of strong universality. We say a random hash function $h : U \to [m]$ is *strongly s-universal* if (1) every key x hash "close to" uniformly in the sense that for every hash value $q \in [m]$, we have $\Pr[h(x) = q] \le c/m$, and (2) every pair of distinct keys hash independently.

Exercise 3.1 *Argue that if $h : U \to [m]$ is strongly c-universal, then h is also c-universal.*

3.1 Applications

One very important application of strongly universal hashing is *coordinated sampling*. In the simplest form, we have a strongly universal hash function $h : U \to [m]$ and then for some threshold $t \in [m]$, we sample x if $h(x) < t$, which happens with probability t/m. For a set $A \subset U$, let $S_{h,t}(A) = \{x \in A \mid h(x) < t\}$ denote the resulting sample from A. Then, by linearity of expectation, $E[|S_{h,t}(A)|] = |A| \cdot t/m$. Conversely, this means that if we have $S_{h,t}(A)$, then we can estimate $|A|$ as $|S_{h,t}(A)| \cdot m/t$.

Suppose we for two different sets B and C have found the samples $S_{h,t}(B)$ and $S_{h,t}(C)$. Based on these we can compute the sample of the union as the union of the samples, that is, $S_{h,t}(B \cup C) = S_{h,t}(B) \cup S_{h,t}(C)$. Likewise, we can compute the sample of the intersection as $S_{h,t}(B \cap C) = S_{h,t}(B) \cap S_{h,t}(C)$. We can then estimate the size of the union and intersection multiplying the corresponding sample sizes by m/t.

The crucial point here is that the sampling from different sets can be done in a distributed fashion as long as a fixed h and t is shared, coordinating the sampling at all locations. This is used, e.g., in machine learning, where we can store the samples of many different large sets. When a new set comes, we sample it, and compare it with the stored samples to estimate which set it has most in common with.

Another application is on the Internet where all routers can store samples of the packets passing through. If a packet is sampled, it is sampled by all routers that it passes, and this means that we can follow the packets route through the network. If the routers did not use coordinated sampling, the chance that the same packet would be sampled at multiple routers would be very small.

The reason that we want strong universality is that we want $|S_{h,t}(A)|$ to be concentrated around its mean $|A| \cdot t/m$ so that we can trust the estimate $|S_{h,t}(A)| \cdot m/t$ of $|A|$.

For $a \in A$, let X_a be the indicator variable for a being sampled, that is $X_a = [h(a) < t]$. Then $|S_{h,t}(A)| = X = \sum_{a \in A} X_a$.

Since h is strongly universal, for any distinct keys $a, b \in A$, $h(a)$ and $h(b)$ are independent, but then X_a and X_b are also independent random variables. Then, by Lemma 3.4 in [8], $\mathsf{Var}[X] = \sum_a \mathsf{Var}[X_a]$ (Lemma 3.4 assumes full independence, but inspecting the proof, it only uses pair-wise independence). Let $p = t/m$ be the common sampling probability for all keys. Also, let $\mu = p|A| = E[X]$ be the expectation of X. For each X_a, we have $E[X_a] = p$, so $\mathsf{Var}[X_a] = p(1-p)$. It follows that $\mathsf{Var}[X] = \mu(1-p) < \mu$. By definition, the standard deviation of X is

$$\sigma_X = \sqrt{\mathsf{Var}[X]} = \sqrt{\mu(1-p)}. \tag{10}$$

Now, by Chebyshev's inequality [8, Theorem 3.3], for any $q > 0$,

$$\Pr[|X - \mu| \ge t\sigma_X] \le 1/q^2. \tag{11}$$

Exercise 3.2 *Suppose that* $|A| = 100,000,000$ *and* $p = t/m = 1/100$. *Then* $E[X] = \mu = 1,000,000$. *Give an upper bound for the probability that* $|X - \mu| \geq 10,000$. *These numbers correspond to a 1% sampling rate and a 1% error.*

3.2 Multiply-mod-prime

The classic strongly universal hashing scheme is a multiply-mod-prime scheme. For some prime p, uniformly at random we pick $(a, b) \in [p]^2$ and define $h_{a,b} : [p] \to [p]$ by

$$h_{a,b}(x) = (ax + b) \bmod p.$$

To see that this is strongly universal, consider distinct keys $x, y \in [p]$ and possibly non-distinct hash values $q, r \in [p]$, $h_{a,b}(x) = q$ and $h_{a,b}(x) = r$. This is exactly as in (3) and (4), and by Lemma 2.2, we have a 1-1 correspondence between pairs $(a, b) \in [p] \times [p]$ and pairs $(q, r) \in [p]^2$. Since (a, b) is uniform in $[p]^2$ it follows that (q, r) is uniform in $[p]^2$, hence that the pair-wise event $h_{a,b}(x) = q$ and $h_{a,b}(x) = r$ happens with probability $1/p^2$.

Exercise 3.3 *Let* $m \leq p$. *For random* $(a, b) \in [p]^2$, *define the hash function* $h_{a,b} : [p] \to [m]$ *by*

$$h_{a,b}(x) = ((ax + b) \bmod p) \bmod m.$$

(a) Argue $h_{a,b}$ *is strongly 2-universal.*

(b) In the universal multiply-mod-prime hashing from Section 2, we insisted on $a \neq 0$, *but now we consider all* $a \in [p]$. *Why this difference?*

3.3 Multiply-shift

We now present a simple generalization from [4] of the universal multiply-shift scheme from Section 2 that yields strong universality. As a convenient notation, for any bit-string z and integers $j > i \geq 0$, $z[i, j) = z[i, j - 1]$ denotes the number represented by bits $i, ..., j - 1$ (bit 0 is the least significant bit, which confusingly, happens to be rightmost in the standard representation), so

$$z[i, j) = \lfloor (z \bmod 2^j)/2^i \rfloor.$$

To hash $[2^w] \to [2^\ell]$ we may pick any $\overline{w} \geq w + \ell - 1$. For any pair $(a, b) \in [\overline{w}]^2$, we define $h_{a,b} : [2^w] \to [2^\ell]$ by

$$h_{a,b}(x) = (ax + b)[\overline{w} - \ell, \overline{w}). \tag{12}$$

As for the universal multiply shift, we note that the scheme of (12) is easy to implement with convenient parameter choices, e.g., with $\overline{w} = 64$, $w = 32$ and $\ell = 20$, we get the C-code:

```
#include <stdint.h>
// defines uint32/64_t as unsigned 32/64-bit integer.
uint64_t hash(uint32_t x; uint32_t l; uint64_t a; uint64_t b; ) {
// hashes x strongly universally into l bits
// using the random seeds a and b.
  return (a*x+b) >> (64-l);}
```

We will prove that the scheme from (12) is strongly universal.In the proof we will reason a lot about uniformly distributed variables, e.g., if $X \in [m]$ is uniformly distributed and β is a constant integer, then $(X + \beta) \bmod m$ is also uniformly distributed in $[m]$. More interestingly, we have

Fact 3.2 *Consider two positive integers α and m that are relatively prime, that is, α and m have no common prime factor. If X is uniform in $[m]$, then $(\alpha X) \bmod m$ is also uniformly distributed in $[m]$. Important cases are (a) if $\alpha < m$ and m is prime, and (b) if α is odd and m is a power of two.*

Proof We want to show that for every $y \in [m]$ there is at most one $x \in [m]$ such that $(\alpha x) \bmod m = y$, for then there must be exactly one $x \in [m]$ for each $y \in [m]$, and vice versa. Suppose we had distinct $x_1, x_2 \in [m]$ such that $(\alpha x_1) \bmod m = y = (\alpha x_2) \bmod m$. Then $\alpha(x_2 - x_1) \bmod m = 0$, so m is a divisor of $\alpha(x_2 - x_1)$. By the fundamental theorem of arithmetic, every positive integer has a unique prime factorization, so all prime factors of m have to be factors of $\alpha(x_2 - x_1)$ in same or higher powers. Since m and α are relatively prime, no prime factor of m is factor of α, so the prime factors of m must all be factors of $x_2 - x_1$ in same or higher powers. Therefore m must divide $x_2 - x_1$, contradicting that $x_1 \not\equiv x_2 \pmod m$. Thus, as desired, for any $y \in [m]$, there is at most one $x \in [m]$ such that $(\alpha x) \bmod m = y$. ■

Theorem 3.3 *When $a, b \in [\overline{w}]$ are uniform and independent, then the multiply-shift scheme from (12) is strongly universal.*

Proof Consider any distinct keys $x, y \in [2^w]$. We want to show that $h_{a,b}(x)$ and $h_{a,b}(y)$ are independent uniformly distributed variables in $[2^\ell]$.

Let s be the index of the least significant `1`-bit in $(y - x)$ and let z be the odd number such that $(y - x) = z2^s$. Since z is odd and a is uniform in $[2^{\overline{w}}]$, by Fact 3.2 (b), we have that az is uniform in $[2^{\overline{w}}]$. Now $a(y - x) = az2^s$ has all `0`s in bits $0, .., s - 1$ and a uniform distribution on bits $s, .., s + \overline{w} - 1$. The latter implies that $a(y - x)[s, .., \overline{w} - 1]$ is uniformly distributed in $[2^{\overline{w}-s}]$.

Consider now any fixed value of a. Since b is still uniform in $[2^{\overline{w}}]$, we get that $(ax + b)[0, \overline{w})$ is uniformly distributed, implying that $(ax + b)[s, \overline{w})$ is uniformly distributed. This holds for any fixed value of a, so we conclude that $(ax + b)[s, \overline{w})$ and $a(y - x)[s, \overline{w})$ are independent random variables, each uniformly distributed in $[2^{\overline{w}-s}]$.

Now, since $a(y - x)[0, s) = 0$, we get that

$$(ay + b)[s, \infty) = ((ax + b) + a(y - x))[s, \infty) = ((ax + b)[s, \infty) + (a(y - x))[s, \infty).$$

The fact that $a(y-x)[s, \overline{w})$ is uniformly distributed independently of $(ax+b)[s, \overline{w})$ now implies that $(ay+b)[s, \overline{w})$ is uniformly distributed independently of $(ax+b)[s, \overline{w})$. However, $\overline{w} \geq w+\ell-1$ and $s < w$ so $s \leq w-1 \leq \overline{w}-\ell$. Therefore $h_{a,b}(x) = (ax+b)[\overline{w}-\ell, \overline{w})$ and $h_{a,b}(y) = (ay+b)[\overline{w}-\ell, \overline{w})$ are independent uniformly distributed variables in $[2^\ell]$. ■

In order to reuse the above proof in more complicated settings, we crystallize a technical lemma from the last part:

Lemma 3.4 *Let $\overline{w} \geq w + \ell - 1$. Consider a random function $g : U \to [2^{\overline{w}}]$ (in the proof of Theorem 3.3, we would have $U = [2^w]$ and $g(x) = ax[0, \overline{w})$) with the property that there for any distinct $x, y \in U$ exists a positive $s < w$, determined by x and y (and not by g, e.g., in the proof of Theorem 3.3, s was the least significant set bit in $y - x$), such that $(g(y) - g(x))[0, s) = 0$ while $(g(y) - g(x))[s, \overline{w})$ is uniformly distributed in $[2^{\overline{w}-s}]$. For b uniform in $[2^{\overline{w}}]$ and independent of g, define $h_{g,b} : U \to [2^\ell]$ by*

$$h_{g,b}(x) = (g(x) + b)[\overline{w} - \ell, \overline{w}).$$

Then $h_{g,b}(x)$ is strongly universal.

3.4 Vector multiply-shift

Our strongly universal multiply shift scheme generalizes nicely to vector hashing. The goal is to get strongly universal hashing from $[2^w]^d$ to 2^ℓ. With $\overline{w} \geq w + \ell - 1$, we pick independent uniform $a_0, ...a_{d-1}, b \in [2^{\overline{w}}]$ and define $h_{a_0,...a_{d-1},b} : [2^w]^d \to [2^\ell]$ by

$$h_{a_0,...a_{d-1},b}(x_0, ..., x_{d-1}) = \left(\left(\sum_{i \in [d]} a_i x_i \right) + b \right) [\overline{w} - \ell, \overline{w}). \tag{13}$$

Theorem 3.5 *The vector multiply-shift scheme from (13) is strongly universal.*

Proof We will use Lemma 3.4 to prove that this scheme is strongly universal. We define $g : [2^w]^d \to [2^{\overline{w}}]$ by

$$g(x_0, ..., x_{d-1}) = \left(\sum_{i \in [d]} a_i x_i \right) [0, \overline{w}).$$

Consider two distinct keys $x = (x_0, ..., x_{d-1})$ and $y = (y_0, ..., y_{d-1})$. Let j be an index such that $x_j \neq y_j$ and such that the index s of the least significant set bit is as small as possible. Thus $y_j - x_j$ has 1 in bit s, and all $i \in [d]$ have $(y_j - x_j)[0, s) = 0$. As required by Lemma 3.4, s is determined from the keys only, as required by Lemma 3.4. Then

$$(g(y) - g(x))[0, s) = \left(\sum_{i \in [d]} a_i (y_i - x_i) \right) [0, s) = 0$$

regardless of $a_0, ...a_{d-1}$. Next we need to show that $(g(y) - g(x))[s, \overline{w})$ is uniformly distributed in $[2^{\overline{w}-s}]$. The trick is to first fix all a_i, $i \neq j$, arbitrarily, and then argue that $(g(y) - g(x))[s, \overline{w})$ is uniform when a_i is uniform in $[2^{\overline{w}}]$. Let z be the odd number such that $z2^s = y_j - x_j$. Also, let Δ be the constant defined by

$$\Delta 2^s = \sum_{i \in [d], i \neq j} a_i (y_i - x_j).$$

Now

$$g(y) - g(x) = (a_j z + \Delta) 2^s.$$

With z odd and Δ a fixed constant, the uniform distribution on $a_j \in [2^{\overline{w}}]$ implies that $(a_j z + \Delta) \bmod 2^{\overline{w}}$ is uniform in $[2^{\overline{w}}]$ but then $(a_j z + \Delta) \bmod 2^{\overline{w}-s} = (g(y) - g(x))[s, \overline{w})$ is also uniform in $[2^{\overline{w}-s}]$. Now Lemma 3.4 implies that the vector multiply-shift scheme from (13) is strongly universal. ∎

Exercise 3.4 *Corresponding to the universal hashing from Section 2, suppose we tried with $\overline{w} = w$ and just used random odd $a_0, ..., a_{d-1} \in [2^w]$ and a random $b \in [2^w]$, and defined*

$$h_{a_0,...a_{d-1},b}(x_0, ..., x_{d-1}) = \left(\left(\sum_{i\in[d]} a_i x_i\right) + b\right)[w - \ell, w).$$

Give an instance showing that this simplified vector hashing scheme is not remotely universal.

Our vector hashing can also be used for universality, where it gives collision probability $1/2^\ell$. As a small tuning, we could skip adding b, but then we would only get the same $2/2^\ell$ bound as we had in Section 2.

3.5 Pair-multiply-shift

A cute trick from [1] allows us roughly double the speed of vector hashing, the point being that multiplication is by far the slowest operation involved. We will use exactly the same parameters and seeds as for (13). However, assuming that the dimension d is even, we replace (13) by

$$h_{a_0,...a_{d-1},b}(x_0, ..., x_{d-1}) = \left(\left(\sum_{i\in[d/2]} (a_{2i} + x_{2i+1})(a_{2i+1} + x_{2i})\right) + b\right)[\overline{w} - \ell, \overline{w}). \tag{14}$$

This scheme handles pairs of coordinates $(2i, 2i + 1)$ with a single multiplication. Thus, with $\overline{w} = 64$ and $w = 32$, we handle each pair of 32-bit keys with a single 64-bit multiplication.

Exercise 3.5 (a bit more challenging) *Prove that the scheme defined by (14) is strongly universal. One option is to prove a tricky generalization of Lemma 3.4 where $(g(y) - g(x))[0, s)$ may not be 0 but can be any deterministic function of x and y. With this generalization, you can make a proof similar to that for Theorem 3.5 with the same definition of j and s.*

Above we have assumed that d is even. In particular this is a case, if we want to hash an array of 64-bit integers, but cast it as an array of 32-bit numbers. If d is odd, we can use the pair-multiplication for the first $\lfloor d/2 \rfloor$ pairs, and then just add $a_d x_d$ to the sum.

4 String hashing

4.1 Hashing vector prefixes

Sometimes what we really want is to hash vectors of length up to D but perhaps smaller. As in the multiply-shift hashing schemes, we assume that each coordinate is from $[2^w]$. The simple point is that we only want to spend time proportional to the actual length $d \leq D$. With $\overline{w} \geq w + \ell - 1$, we pick independent uniform $a_0, ...a_{D-1} \in [2^{\overline{w}}]$. For even d, we define $h : \bigcup_{\text{even } d \leq D}[2^w]^d \rightarrow [2^\ell]$ by

$$h_{a_0,...a_D}(x_0, ..., x_{d-1}) = \left(\left(\sum_{i\in[d/2]} (a_{2i} + x_{2i+1})(a_{2i+1} + x_{2i})\right) + a_d\right)[\overline{w} - \ell, \overline{w}). \tag{15}$$

Exercise 4.1 *Prove that the above even prefix version of pair-multiply-shift is strongly universal. In the proof you may assume that the original pair-multiply-shift from (14) is strongly universal, as you may have proved in Exercise 3.5. Thus we are considering two vectors $x = (x_0, ...x_{d-1})$ and $y = (y_0, ...y_{d'-1})$. You should consider both the case $d' = d$ and $d' \neq d$.*

4.2 Hashing bounded length strings

Suppose now that we want to hash strings of 8-bit characters, e.g., these could be the words in a book. Then the nil-character is not used in any of the strings. Suppose that we only want to handle strings up to some maximal length, say, 256.

With the prefix-pair-multiply-shift scheme from (15), we have a very fast way of hashing strings of d 64-bit integers, casting them as $2d$ 32-bit integers. A simple trick now is to allocate a single array x of $256/8 = 32$ 64-bit integers. When we want to hash a string s with c characters, we first set $d = \lceil c/8 \rceil$ (done fast by `d=(c+7)>>3`). Next we set $x_{d-1} = 0$, and finally we do a memory copy of s into x (using a statement like `memcpy(x,s,c)`). Finally, we apply (15) to x.

Note that we use the same variable array x every time we want to hash a string s. Let s^* be the image of s created as a $c^* = \lceil c/8 \rceil$ length prefix of x.

Exercise 4.2 *Prove that if s and t are two strings of length at most 256, neither containing the nil-character, then their images s^* and t^* are different. Conclude that we now have a strongly universal hash functions for such string.*

Exercise 4.3 *Implement the above hash function for strings. Use it in a chaining based hash table, and apply it to count the number of distinct words in a text (take any pdf-file and convert it to ascii, e.g., using `pdf2txt`).*

To get the random numbers defining your hash functions, you can go to `random.org`).

One issue to consider when you implement a hash table is that you want the number m of entries in the hash array to be as big as a the number of elements (distinct words), which in our case is not known in advance. Using a hash table of some start size m, you can maintain a count of the distinct words seen so far, and then double the size when the count reaches, say, $m/2$.

Many ideas can be explored for optimization, e.g., if we are willing to accept a small false-positive probability, we can replace each word with a 32- or 64-bit hash value, saying that a word is new only if it has a new hash value.

Experiments with some different texts: different languages, and different lengths. What happens with the vocabulary.

The idea now is to check how much time is spent on the actual hashing, as compared with the real code that both does the hashing and follows the chains in the hash array. However, if we just *compute the hash values, and don't use them, then some optimizing compilers, will notice, and just do nothing. You should therefore add up all the hash values, and output the result, just to force the compiler to do the computation.*

Linear Probing As an alternative in Exercise 4.3, you could implement the hash table using linear probing. However, with linear probing, if you want the hashing to be safe in the sense of getting expected constant time for any input, then this does require some more powerful hash functions (which we have not covered yet). Currently, the most efficient schemes combine methods from [9, 10]. Referring to [9, §1, §6] for the definition of tabulation hashing, you can apply it directly on words with up to 4 characters, that is, 32-bits. If the words are longer, you can first hash them, as described above, down to 32-bits, and then rehash them using simple tabulation. The reason that this works is explained in [10]. In [10, §6] you will also see that it is possible to store the distinct strings directly in the hash array.

4.3 Hashing variable length strings

We now consider the hashing of a variable length strings $x_0x_1\cdots x_d$ where all characters belong to some domain $[u]$.

We use a the method from [5], which first picks a prime $p \geq u$. The idea is view $x_0, ..., x_d$ as coefficients of a degree d polynomial

$$P_{x_0,\ldots,x_d}(\alpha) = \sum_{i=0}^{d} x_i \alpha^i \bmod p$$

over $\mathbb{Z}_p$. As seed for our hash function, we pick an argument $a \in [p]$, and compute the hash function

$$h_a(x_0\cdots x_d) = P_{x_0,\ldots,x_d}(a).$$

Consider some other string $y = y_0y_1\cdots y_{d'}$, $d' \leq d$. We claim that

$$\Pr_{a\in[p]}[h_a(x_0\cdots x_d) = h_a(y_0\cdots y_{d'})] \leq d/p$$

The proof is very simple. By definition, the collision happens only if a is root in the polynomial $P_{y_0,\ldots,y_{d'}} - P_{x_0,\ldots,x_d}$. Since the strings are different, this polynomial is not the constant zero. Moreover, its degree is at most d. Since the degree is at most d, the fundamental theorem of algebra tells us that it has at most d distinct roots, and the probability that a random $a \in [p]$ is among these roots is at most d/p.

Now, for a fast implementation using Horner's rule, it is better to reverse the order of the coefficients, and instead use the polynomial

$$P_{x_0,\ldots,x_d}(a) = \sum_{i=0}^{d} x_{d-i} a^i \mod p$$

Then we compute $P_{x_0,\ldots,x_d}(a)$ using the recurrence

- $H_a^0 = x_0$
- $H_a^i = (a(H^{i-1} + x_i)) \mod p$
- $P_{x_0,\ldots,x_d}(a) = H_a^d$.

With this recurrence, we can easily update the hash value if new character x_{d+1} is added to the end of the string x_{d+1}. It only takes an addition and a multiplication modulo p. For speed, we would let p be a Mersenne prime, e.g. $2^{89} - 1$.

Recall from the discussed in Section 2.2.1 that the multiplication modulo a prime like $2^{89} - 1$ is a bit complicated to implement.

The collision probability d/p may seem fairly large, but assume that we only want hash values in the range $m \leq p/d$, e.g, for $m = 2^{32}$ and $p = 2^{89} - 1$, this would allow for strings of length up to 2^{57}, which is big enough for most practical purposes. Then it suffices to compose the string hashing with a universal hash function from $[p]$ to $[m]$. Composing with the previous multiply-mod-prime scheme, we end up using two seeds $a, b \in [p]$, and then compute the hash function as

$$h_{a,b}(x_0, ..., x_d) = \left(b \left(\sum_{i=0}^{d} x_i a^i \right) \mod p \right) \mod m$$

Exercise 4.4 *Assuming that strings have length at most p/m, argue that the collision probability is at most $2/m$.*

Above we can let u be any value bounded by p. With $p = 2^{89} - 1$, we could use $u = 2^{64}$ thus dividing the string into 64-bit characters.

Exercise 4.5 *Implement the above scheme and run it to get a 32-bit signature of a book.*

Major speed-up The above code is slow because of the multiplications modulo Mersenne primes, one for every 64 bits in the string.

An idea for a major speed up is to divide you string into chunks $X_0, ..., X_j$ of 32 integers of 64 bits, the last chunk possibly being shorter. We want a single universal hash function $r : \bigcup_{d \leq 32} [2^{64}]^d \to [2^{64}]$. A good choice would be to use our strongly universal pair-multiply-shift scheme from (15). It only outputs 32-bit numbers, but if we use two different such functions, we can concatenate their hash values in a single 64-bit number.

Exercise 4.6 *Prove that if r has collision probability P, and if $(X_0, ..., X_j) \neq (Y_0, ..., Y_{j'})$, then*

$$\Pr[(r(X_0), ..., r(X_j) = (r(Y_0), ..., r(Y_{j'}))] \leq P.$$

The point above is that in the above is that $r(X_0), ..., r(X_j)$ is 32 times shorter than $X_0, ..., X_j$. We can now apply our slow variable length hashing based on Mersenne primes to the reduced string $r(X_0), ..., r(X_j)$. This only adds P to the overall collision probability.

Exercise 4.7 *Implement the above tuning. How much faster is your hashing now?*

Speed-up for short strings When devising a generic string hashing algorithm, we do not know if it is going to be applied on short or long strings, e.g., as in Section 4.2, we may just use it to hash words in a book, most of which are fairly short, and then we would like to just use the hash function from 4.2, and not do any polynomial with Mersenne primes. As a generic trick, suppose we have one hash function h_0 that is really good for hashing strings of length up to ℓ, and another h_1 that is good for longer strings. Assuming that the hash values are bit-strings from some range R, we can then pick a random value $a \in R$, and use the combined hash function

$$h(x) = \begin{cases} h_0(x) & \text{if } |x| \leq \ell \\ a \oplus h_1(x) & \text{otherwise.} \end{cases}$$

Exercise 4.8 *Show that if h_0 and h_1 are (strongly) universal, then so is h.*

A nice aspect of the above combination is that if the majority of the strings are short, then loop prediction should imply that we pay very little for the test $|x| \leq \ell$.

Major open problem Can we get something simple and fast like multiply-shift to work directly for strings, so that we do not need to compute polynomials over prime fields?

References

[1] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: fast and secure message authentication. In *Proc. 19th CRYPTO*, pages 216–233, 1999.

[2] J. Carter and M. Wegman. Universal classes of hash functions. *J. Comp. Syst. Sci.*, 18:143–154, 1979. Announced at STOC'77.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, McGraw-Hill, 3 edition, 2009.

[4] M. Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In *Proc. 13th STACS, LNCS 1046*, pages 569–580, 1996.

[5] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable (extended abstract). In *Proc. 19th ICALP, LNCS 623*, pages 235–246, 1992.

[6] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25:19–51, 1997.

[7] A. I. Dumey. Indexing for rapid random access memory systems. *Computers and Automation*, 5(12):6–9, 1956.

[8] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[9] M. Pătraşcu and M. Thorup. The power of simple tabulation hashing. *J. ACM*, 59(3):Article 14, 2012. Announced at STOC'11.

[10] M. Thorup. String hashing for linear probing. In *Proceedings of the 20th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 655–664, 2009.

[11] M. Wegman and J. Carter. New hash functions and their use in authentication and set equality. *J. Comp. Syst. Sci.*, 22:265–279, 1981.