451: Hashing

G. Miller, K. Sutner
Carnegie Mellon University
2020/10/15

Suppose we have a map $h : \mathcal{U} \to V$ from a large set $\mathcal{U}$, the universe, to a small set V.

We would like h to behave like an injective function.

Of course, this is impossible: there are no injective functions $\mathcal{U} \to V$ unless $|\mathcal{U}| \leq |V|$, and we assume the opposite. There always will be collisions $h(x) = h(y)$ for $x \neq y$.

Fine, but how about a map that behaves like an injective one at least when dealing with reasonably small subsets $S \subset \mathcal{U}$? If S were given ahead of time, then this is straightforward.

But what if we have no prior information about S?

In applications we want to store key-value pairs (k, v) in a location determined by $h(k)$. These details are irrelevant for the following discussion.

So, for simplicity, assume

$$V = \{0, 1, \ldots, m-1\} = (m)$$

so we are dealing with a table of size m. Similarly we may safely assume that

$$\mathcal{U} = \{0, 1, \ldots, N-1\} = (N)$$

In practice, think of keys as being sequences of k bytes, corresponding to a (possibly very large) number written in base 256; so $N = 2^{8k}$. Even for $k = 10$ this is about 1.21×10^{24}.

We are looking for a hash function

$$h : (N) \to (m)$$

h must be easy to compute and should distribute the elements of the universe evenly over the hash table.

Key parameters:

- N, the size of the universe (huge),
- n, the number of elements in S,
- m, the size of the hash table,
- $\alpha = n/m$, the load factor.

Again, $m \ll N$ and typically $n = O(m)$ so that the load factor may be larger than 1, but not hugely larger. Resize the table when α gets too big.

The easiest case is when S is static and we are only interested in supporting lookups once the structure has been built.

For a standard application like memoizing we need dynamic insert and lookup.

More complicated is full support for lookup, insert, delete, resize, . . . And, as always, there are tricks: say, to speed up deletes, just set a flag and wait till the next resize operation to actually remove the dead bodies.

We will focus on the hash function.

There are two basic methods to resolve collisions:

- Chaining: keep values in containers (bins, buckets) hanging off the actual table. Linked lists are popular for this purpose.
- Open Addressing: place values into the table, moving to different slots (in some systematic fashion) if the original one is already taken.

Another possibility is to avoid collisions altogether, provided that S is small enough; more later.

Exercise

Recall all the implementation details of hash tables.

The requirements for a good hash function are somewhat contradictory:

- it should behave just like a random function and distribute all the keys evenly over the table, (aka the mixing property);
- yet it must be easy to compute and perfectly deterministic.

Just to be clear: real randomness won't work: we need to compute h on the fly, not store some huge table. We have to make do with h that "appears random."

It's not entirely clear that there should be any good answers.

Below are some simple standard methods; take a look at Gonnet, Baeza-Yates for more.

Complexity usually is an obstacle to effective algorithms, but sometimes it actually helps.

- If the data is random, we just have to make sure that h preserves enough randomness.
- If the data has a lot of structure, then it is easy to compute. But if the fibres $h^{-1}(v)$ are computationally crazy, then it is very unlikely that, say, $S \subseteq h^{-1}(v)$.

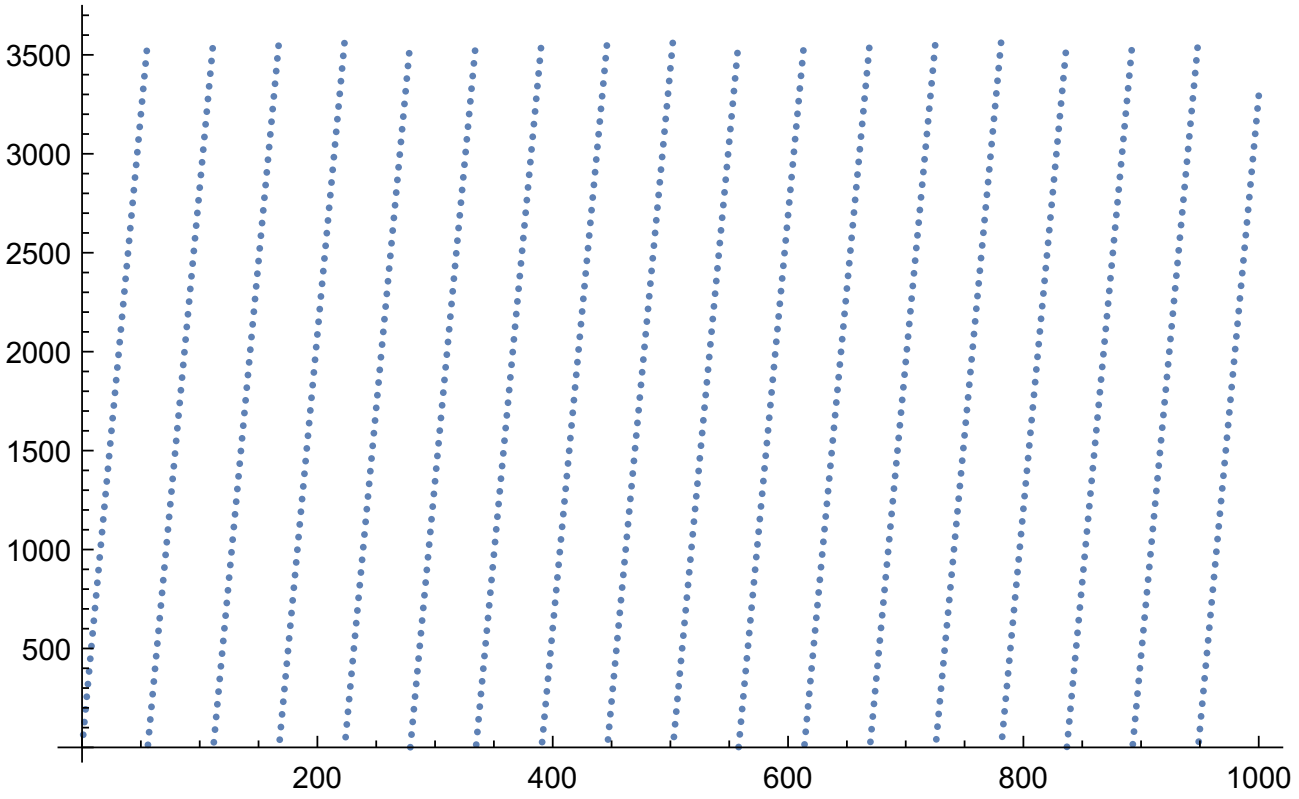
This sounds utterly wishy-washy, but it is known that strictly random objects have no short description (essentially, you just have to write down all the bits; Kolmogorov-Chaitin complexity). But regular data have a short description, so they should not match up with highly complicated fibres.

A particularly simple type of hash function uses modular arithmetic

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

where the modulus m is a sufficiently large prime.

Getting one's hands on a prime is an interesting problem, but not really an issue here: some libraries (e.g. the STL) have a table of primes hard-coded. The primes are chosen to be close to 2^i for $i = 5, \ldots, m$ so that one can essentially double or half the size of the table whenever necessary for resizing.



Primality ensures a reasonable distribution of the hash values, even when the input is highly regular, say, an arithmetic sequence. Compare this to, say, a modulus $m = 2^k$.

Here $m = 3751$, entries are the arithmetic progression $x = i\,64$, $i = 1, \ldots, 1000$.

Let $0 < r < 1$ be irrational. Then $\{ i \cdot r \bmod 1 \mid i < n \}$ is very evenly distributed over the interval $[0, 1]$, giving rise to a hash function

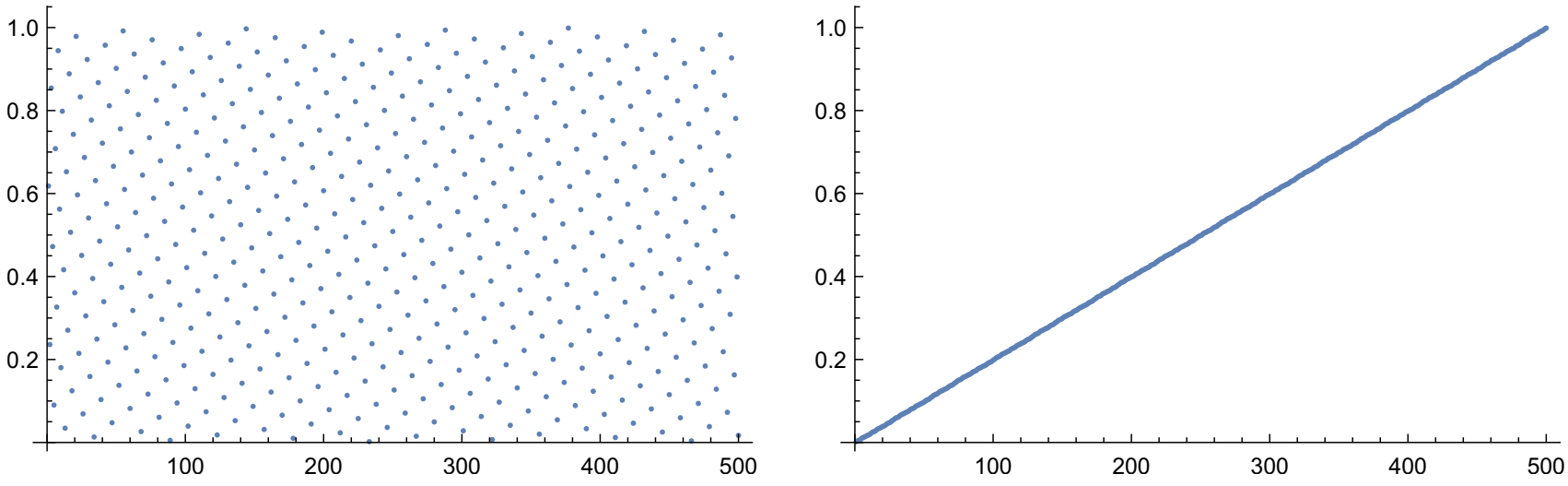
$$h(x) = \lfloor m(xr \bmod 1) \rfloor.$$

A typical choice is (the "other" Golden Ratio)

$$r = (\sqrt{5} - 1)/2 \approx 0.6180339887\,49894813$$

and $m = 2^k$.

BTW, r is not known to be normal (every block of k decimal digits appears with frequency 10^{-k}), though its digits seem to be very evenly distributed. Note that implementation requires a bit of work.



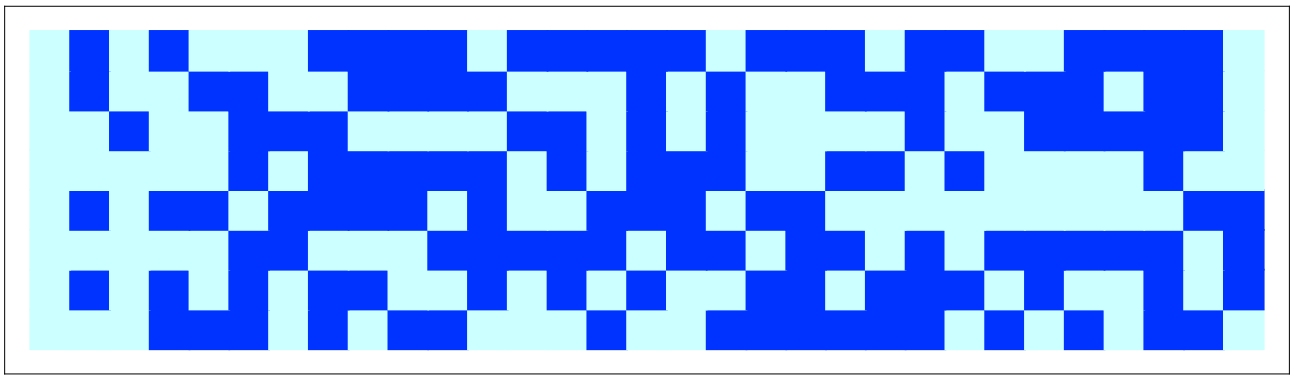
Actual values on the left, sorted on the right. Looks fairly even.

Alternatively, one can use a variety of bit-operations (shifting, logical ops, finite fields GF(2^k)) to obtain hash values. These methods tend to have a distinctly hacky flavor.

Suppose $\mathcal{P}$ is a random permutation of the 256 bytes and keys x are sequences of k bytes. So $m = 256$ and $N = m^k$.

defun $hash(x)$
$h = 0$ // `uint8`
forall $i = 1, 2, \ldots, k$
    $h = \mathcal{P}(h \oplus x(i))$
return h

For example, for the highly regular $x = (5, 10, 15, \ldots, 150)$ we get the following sequence of h-values during the execution of the loop. $\mathcal{P}$ here is a pseudo-random permutation generated by a reasonably good RNG.

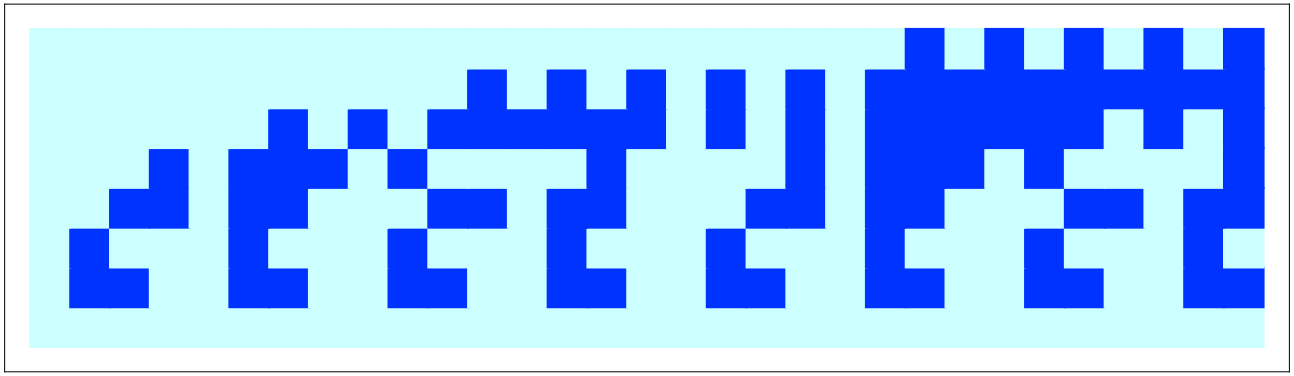


Each column is a byte, the first is 0, the last is 116 (LSD on top).

There are only 4 collisions between the sequences $i\,[40] \bmod 256$ for $i = 1, \ldots, 50$. There are about 2.14×10^{96} byte sequences of length $k = 40$.

Suppose we left off the permutation $\mathcal{P}$. After all, all the xor operations make a mess by themselves, right? Google elementary cellular automaton number 90.

Disaster strikes, for $x = 6\,[30] = 6, 12, \ldots, 180$ the sequence of h values now looks like this:



For the same experiment as on the last slide, the frequencies of collision counts now look like so:

bin size	1	2	3	4	9
bin count	11	3	4	3	1

Again, $\mathcal{P}$ is a permutation of the 256 bytes. This time h is supposed to be a 32-bit unsigned int. As before, x is a sequence of k bytes.

defun $hash(x)$
$h = 0$ // `uint32`
forall $i = 1, 2, \ldots, k$
    $h = (h \ll 8) \oplus \mathcal{P}((h \gg 24) \oplus x(i))$
return h

Of course, it is very difficult to analyze this type of hash map.

Exercise

Suppose $\mathcal{P}$ is known to an adversary. Is that enough to break such a hash table?

Here is a less hacky method, mostly used for error detection in storage devices and networks: check whether a bit sequence got corrupted (and request a retransmit).

Think of a (long) bit sequence $\boldsymbol{a} \in \mathbf{2}^n$ as the coefficient list of a polynomial over GF_2, the two element field.

$$p(x) = a_{n-1}x^{n-1} + \ldots a_1 x + a_0$$

In coding theory, this is called a code polynomial. Now fix another polynomial $g(x)$ of degree d (something like 8, 12, 16, 32 ...), the generator polynomial.

This may sound weird, but is justified by the fact that g generates an ideal in $\mathsf{GF}_2[x]$ (a principal ideal domain), the ideal of all code words.

We can construct a hash function based on polynomial arithmetic: compute the remainder of the code polynomial upon division by the generator polynomial.

$$h(p(x)) = p(x) \pmod{g(x)}$$

Polynomial g has degree d, so we get a d-bit CRC, the coefficient list of the remainder polynomial.

In applications, we transmit the coefficient list of

$$q(x) = x^d p(x) + h(x^d p(x))$$

If there is no transmission error, then q is divisible by g at the other end (we are in characteristic 2).

How would we actually compute h?

Since we are in characteristic 2, $g(x) = 0$ means

$$x^d = p_{d-1}x^{d-1} + p_{d-2}x^{d-2} + \ldots + p_1 x^1 + p_0$$

This can be translated into code that runs very fast (should be written in assembly).

$h = 0$
$q = 1$
for $i = 0, \ldots, n-1$ **do**
    if $a_i == 1$
    then $h = h + q$
    $q = x \cdot q \bmod p$ // cheap shift and bitmasking
od
return h

The question is: what should g be? E.g., $g(x) = x + 1$ is just plain parity.

How about $g(x) = x^d + 1$? For simplicity, assume $n = md$ and write

$$\boldsymbol{a}(x) = \sum_{j<m} x^{jd} \sum_{i<d} a_{jd+i} x^i$$

So we can detect a 1-bit error in each of the d coefficient blocks. But a 2-bit error in the same block is undetected.

To get more interesting results we need to rely on algebra.

Exercise

Make sure you understand how the algorithm works on these simple generators.

A better choice is an irreducible polynomial, so that $\mathbf{2}[x]/(g(x))$ is (isomorphic to) the finite field of size 2^d.

In fact, we should pick a primitive polynomial: the critical instruction $q = x \cdot q \bmod p$ then generates the whole multiplicative subgroup of GF_{2^d}. These are not so easy to produce, but the area is classical algebra, and one can find suitable polynomials in the literature.

But note that reducible polynomials are also in use, some times of the form $(x+1)\,g(x)$ where p is irreducible.

Exercise

Why would it be useful to have the whole multiplicative subgroup?

Here is a primitive polynomial of degree 8.

$$g(x) = x^8 + x^6 + x^5 + x^2 + 1$$

Hashing 256 polynomials of degree 511 produces the following collision frequencies:

bin size	1	2	3	4	5
bin count	89	45	12	9	1

So where does randomness come from in these methods?

division method	number theory, primality
multiplication method	number theory, irrationality
bit operations	random permutations, combinatorics
CRC	algebra, finite fields

For the bit operations, the space of possible hash functions is enormous:

$$256! \approx 8.58 \times 10^{506} \approx \infty$$

The others are built on solid axiomatics, but the theories a are complicated.

So even a clever and computationally powerful adversary will have great difficulties foiling such hash functions. We will exploit this idea in the next section.

Back to the bit operations. What about the (pseudo-)random numbers used to generate $\mathcal{P}$?

> Anyone attempting to produce random numbers by purely arithmetic means is, of course, in a state of sin.
>
> John von Neumann

True as a matter of principle, but in the RealWorld$^{\mathrm{TM}}$ there are fairly good methods available. Mersenne Twister.

And then there is Quantis. Now if only we had a good axiomatization of physics ...

Let $\mathcal{H}$ be a collection of hash functions (over the same domain and codomain) and define the collision set on $x \neq y \in \mathcal{U}$ to be

$$\mathcal{H}_{xy} = \{\, h \in \mathcal{H} \mid h(x) = h(y) \,\}$$

Call $\mathcal{H}$ universal iff

$$\forall\, x \neq y \in \mathcal{U}\ (\, |\mathcal{H}_{xy}| \leq |\mathcal{H}|/m \,)\,.$$

Equivalently, we have for all $x \neq y$

$$\mathsf{Pr}_{h \in \mathcal{H}}[\, h \text{ collision between } x \text{ and } y \,] \leq 1/m.$$

A priori, it is not clear how we would go about constructing a universal family (of easily computable functions).

Example

As a toy example, consider $m = 2$, $N = 7$ and $|\mathcal{H}| = 4$. So we need to avoid more than two collisions for any pair $0 \leq x < y < 7$.

	0	1	2	3	4	5	6
h_1	0	0	0	0	1	1	1
h_2	0	0	1	1	1	1	0
h_3	0	1	0	1	1	0	1
h_4	1	0	0	1	0	1	1

It might be tempting to lower the bound $1/m$ in the definition of universality, thus further reducing the probability of collisions. Alas, for large N there is not much one can do.

Exercise

Show that for any $\mathcal{H}$ and bound β such that $|\mathcal{H}_{xy}|/|\mathcal{H}| \leq \beta$ for all $x \neq y$, we must have $\beta \geq 1/m - 1/N$.

Lemma

Suppose $\mathcal{H}$ is universal and we hash $n \leq m$ elements were h is chosen at random from $\mathcal{H}$. Then for any key x

$$\mathsf{E}[\ \textit{collisions on } x\ ] < 1.$$

Proof. We need to compute

$$e = \sum_h \Pr[h \text{ chosen }] \cdot \#\text{ collisions on } x \text{ using } h.$$

Since $\Pr[h \text{ chosen }] = 1/|\mathcal{H}|$ the claim follows from universality: sum over all $y \neq x$ to get $e \leq (n-1)/m < 1$. □

Hence the expected cost of a sequence of s operations is $O(s)$ as long as the load factor never goes above 1.

Example (All Functions)

Note that $\mathcal{H} = \mathcal{U} \to m$ is a universal family: the number of collisions on $x \neq y$ is m^{N-1}, so $|\mathcal{H}_{xy}|/|\mathcal{H}| = 1/m$. But this collection is useless computationally: it is tedious to generate its members at random, and we do not wish to store large tables (most of these functions can only be represented by a table of size about $N \log m$).

Example (Prime Table Size)

Assume that the table size $m = p$ is prime. Think of key x as a number being written in base p: $x = x_0, \ldots, x_{k-1}$. Now consider a vector $\boldsymbol{a} = a_0, \ldots, a_{k-1}$ of numbers $0 \leq a_i < p$. Each such vector gives rise to a linear function

$$h_{\boldsymbol{a}}(x) = \sum_{i<k} a_i x_i \bmod p.$$

Lemma

$\mathcal{H} = \{ h_{\boldsymbol{a}} \mid \boldsymbol{a} \in p^k \}$ is a universal collection of hash functions.

Proof.

Since $\mathbb{Z}_p$ is a field and our hash maps are linear over $\mathbb{Z}_p$, $h_{\boldsymbol{a}}(x) = h_{\boldsymbol{a}}(y)$ for $x_i \neq y_i$ implies that

$$a_i = (y_i - x_i)^{-1} \sum_{j \neq i} a_j (x_j - y_j) \pmod{p}$$

Hence there are p^{k-1} possible values of $\boldsymbol{a}$ which produce a collision.

But the total number of choices for $\boldsymbol{a}$ is p^k.

□

To remove the prime table size condition, choose a prime $p > m$ larger than the size of the universe. Define

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

In other words

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

where $0 < a < p$, $0 \leq b < p$. Thus, we have a family $\mathcal{H}$ of size $p(p-1)$.

The functions $f_{a,b}$ are all affine bijections, so there are no collisions at the first level. So how about the second level?

Observe that, for any pair of source points $x_0 \neq x_1 \in \mathbb{Z}_p$ and target points $y_0 \neq y_1 \in \mathbb{Z}_p$, there is precisely one choice of parameters a and b such that $f_{a,b}(x_i) = y_i$:

$$a = \frac{y_0 - y_1}{x_0 - x_1} \qquad b = \frac{x_0 y_1 - y_0 x_1}{x_0 - x_1}.$$

So fix $x_0 \neq x_1$. Thus, if we choose $(a, b) \in \mathbb{Z}_p^\times \times \mathbb{Z}_p$ uniformly at random, then $(h_{a,b}(x_0), h_{a,b}(x_1))$ is also uniformly distributed.

So the probability of a collision between x_0 and x_1 is the same as the probability that $x_0 = x_1 \pmod m$.

Write $0 \leq x < p$ as

$$x = im + j \qquad 0 \leq i < q, 0 \leq j < m$$
$$x = qm + j \qquad 0 \leq j < r$$

In either case, $x \bmod m = j$. Hence, there are r equivalence classes of size $q + 1$, and $m - r$ classes of size q. Let's ignore the case $r = 0$. Then there are $q \leq (p - 1)/m$ collision producing choices for x_1. Hence the probability is at most $1/m$.

Exercise

Deal with the $r = 0$ case.

Exercise

Redo the argument with counting, no probability.

Choose parameters

$$m = 256 \qquad p = 541 \qquad a = 473 \qquad b = 178$$

Inserting 256 random numbers produces collision frequencies

bin size	1	2	3	4
bin count	97	51	15	3

Inserting 256 numbers $20\,i$, $i = 1, \ldots, 256$ produces collision frequencies

bin size	1	2	3
bin count	37	96	9

A method that does not require the use of primes is based on matrix-vector multiplications over $\mathbb{Z}_2$. Suppose $m = 2^k$ so that the hash values are length k bitvectors. The universe consists of all length r bitvectors: $\mathcal{U} = \mathbf{2}^r$.

For any rectangular matrix $M \in \mathbf{2}^{k \times r}$ we can set

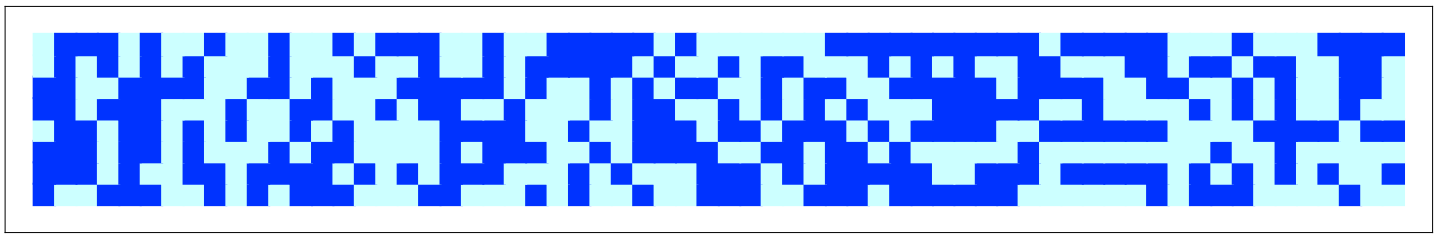
$$h_M(x) = M \cdot x$$

where all the arithmetic is in $\mathbb{Z}_2$. This produces a family of hash functions of size 2^{kr}.

Suppose we have a collision at $x \neq y \in \mathbf{2}^r$. Wlog $x_1 = 0$, $y_1 = 1$. Let M_c be the version of M where the first column is changed to $c \in \mathbf{2}^k$. Then $Mx = M_c x$ but $M_c y$ ranges over all of $\mathbf{2}^k$.

It follows that $2^{k(r-1)} = |\mathcal{H}|/m$ matrices produce a collision on x and y, and $\mathcal{H}$ is duly universal.

Here is a 8×64 random matrix.



Inserting 256 random vectors produces collision frequencies

bin size	1	2	3	4
bin count	92	47	18	4

Inserting 256 numbers $20\,i$, $i = 1, \ldots, 256$ produces collision frequencies

bin size	1	2	3	4
bin count	100	49	18	1

A hash function is perfect if it produces no collisions whatsoever on a given set $S \subseteq \mathcal{U}$.

You have to take this with a grain of salt, one of our methods uses a two-level scheme, and there are collisions on the top level. Of course, we must have $m \geq n = |S|$ in this case. The question arises if there is some reasonable way to construct a perfect hash function. We will only deal with the static case where S is fixed.

The first attempt simply uses a lot of space to insure the absence of collisions with high probability: $m = n^2$.

Lemma

Let $m = n^2$ and pick a hash function randomly from a universal class. Then the probability of having collisions is less than $1/2$.

Proof. The expected number of collisions is

$$\mathsf{E}[C] = \binom{n}{2}\frac{1}{m} = \frac{n-1}{2n} < 1/2.$$

Done by Markov's inequality. □

Lemma (Markov's Inequality)

Let X be a random variable assuming non-negative integer values. Then $\Pr[X \geq k\,\mathsf{E}[X]] \leq 1/k$.

Of course, a quadratic size table is quite unrealistic. However, we can get away with many smaller tables, each of which is still quadratic in the number of elements stored in that particular table.

To be more precise, let $m = n$ and write n_i, $i = 1, \ldots, m$ for the number of elements that hash to slot i under h. Thus, n_i is the size of one bin (a fibre of h).

Now consider the Boolean matrix representation A of that equivalence relation. Clearly the total number of 1's in this matrix is none other than $\sum n_i^2$, the total number of collisions including the phony type $h(x) = h(x)$.

But since h is universal the expected number of these collisions is $n + 2\binom{n}{2}\frac{1}{m} = 2n - 1$.

Hence we have shown

Lemma

$\mathsf{E}[\sum n_i^2] < 2n.$

We can now design a perfect hashing scheme using at most $4n$ space as follows.

Step 1.

- Pick a universal hash function at random for table size $n = m$.
- If $\sum n_i^2 \leq 4n$ go to step 2, otherwise repeat.

Step 2.
Do the following for each of the (non-empty) buckets obtained from Step 1.

- Pick a universal hash function at random for table of size $m = n_i^2$.
- If there are no collisions on the ith bucket, stop; otherwise repeat.

There are loops in this algorithm, and we have to worry about how often we need to repeat an attempt. As it turns out, we can find the requisite hash functions in expected linear time.

Lemma

$$\Pr\left[\sum_i n_i^2 > 4n\right] < 1/2$$

Proof. By Markov, it suffices to show that $\mathsf{E}[\sum n_i^2 < 2n]$.

$$\begin{aligned}
\mathsf{E}[\sum n_i^2] &= \mathsf{E}[\sum_x \sum_y \delta_{h(x)h(y)}] \\
&= n + \mathsf{E}[\sum_x \sum_{y \neq x} \delta_{h(x)h(y)}] \\
&\leq n + n(n-1)/m \quad < 2n
\end{aligned}$$

□