6 Hashing

6.1 Systems of hash functions

Notation:

- The *universe* $\mathcal{U}$. Usually, the universe is a set of integers $\{0, \ldots, U-1\}$, which will be denoted by $[U]$.
- The set of *buckets* $\mathcal{B} = [m]$.
- The set $X \subset \mathcal{U}$ of n items stored in the data structure.
- Hash function $h : \mathcal{U} \to \mathcal{B}$.

Definition: Let $\mathcal{H}$ be a family of functions from $\mathcal{U}$ to $[m]$. We say that the family is *c-universal* for some $c > 0$ if for every pair x, y of distinct elements of $\mathcal{U}$ we have

$$\Pr_{h\in\mathcal{H}}[h(x) = h(y)] \le \frac{c}{m}.$$

In other words, if we pick a hash function h uniformly at random from $\mathcal{H}$, the probability that x and y collide is at most c-times more than for a completely random function h.

Occasionally, we are not interested in the specific value of c, so we simply say that the family is *universal.*

Note: We will assume that a function can be chosen from the family uniformly at random in constant time. Once chosen, it can be evaluated for any x in constant time. Typically, we define a single parametrized function $h_a(x)$ and let the family $\mathcal{H}$ consist of all h_a for all possible choices of the parameter a. Picking a random $h \in \mathcal{H}$ is therefore implemented by picking a random a. Technically, this makes $\mathcal{H}$ a multi-set, since different values of a may lead to the same function h.

Theorem: Let $\mathcal{H}$ be a c-universal family of functions from $\mathcal{U}$ to $[m]$, $X = \{x_1, \ldots, x_n\} \subseteq \mathcal{U}$ a set of items stored in the data structure, and $y \in \mathcal{U} \setminus X$ another item not stored in the data structure. Then we have

$$\mathbb{E}_{h\in\mathcal{H}}[\#i : h(x_i) = h(y)] \le \frac{cn}{m}.$$

That is, the expected number of items that collide with y is at most cn/m.

Proof: Let h be a function picked uniformly at random from $\mathcal{H}$. We introduce indicator random variables

$$A_i = \begin{cases} 1 & \text{if } h(x_i) = h(y), \\ 0 & \text{otherwise.} \end{cases}$$

Expectation of zero-one random variables is easy to calculate: $\mathbb{E}[A_i] = 0 \cdot \Pr[A_i = 0] + 1 \cdot \Pr[A_i = 1] = \Pr[A_i = 1] = \Pr[h(x_i) = h(y)]$. Because $\mathcal{H}$ is universal, this probability is at most c/m.

The theorem asks for the expectation of a random variable A, which counts i such that $h(x_i) = h(y)$. This variable is a sum of the indicators A_i. By linearity of expectation, we have $\mathbb{E}[A] = \mathbb{E}[\sum_i A_i] = \sum_i \mathbb{E}[A_i] \le \sum_i c/m = cn/m$. □

Corollary (Complexity of hashing with chaining): Consider a hash table with chaining which uses m buckets and a hash function h picked at random from a c-universal family. Suppose that the hash table contains items $x_1, \ldots, x_n$.

- Unsuccessful search for an item y distinct from all x_i visits all items in the bucket $h(y)$. By the previous theorem, the expected number of such items is at most cn/m.
- Insertion of a new item y to its bucket takes constant time, but we have to verify that the item was not present yet. If it was not, this is equivalent to unsuccessful search.
- In case of a successful search for x_i, all items visited in x_i's bucket were there when x_i was inserted (this assumes that we always add new items at the end of the chain). So the expected time complexity of the search is bounded by the expected time complexity of the previous insert of x_i.
- Unsuccessful insertion (the item is already there) takes the same time as successful search.
- Finally, deletion takes the same time as search (either successful or unsuccessful).

Hence, if the number of buckets m is $\Omega(n)$, expected time complexity of all operations is constant. If we do not know how many items will be inserted, we can resize the hash table and re-hash all items whenever n grows too much. This is similar to the flexible arrays from section **??** and likewise we can prove that the amortized cost of resizing is constant.

Definition: Let $\mathcal{H}$ be a family of functions from $\mathcal{U}$ to $[m]$. The family is (k,c)-*independent* for integer k ($1 \le k \le |\mathcal{U}|$) and real $c > 0$ iff for every k-tuple $x_1, \ldots, x_k$ of distinct elements of $\mathcal{U}$ and every k-tuple $a_1, \ldots, a_k$ of buckets in $[m]$, we have

$$\Pr_{h\in\mathcal{H}}[h(x_1) = a_1 \wedge \ldots \wedge h(x_k) = a_k] \le \frac{c}{m^k}.$$

That is, if we pick a hash function h uniformly at random from $\mathcal{H}$, the probability that the given items are mapped to the given buckets is at most c-times more than for a completely random function h.

Sometimes, we do not care about the exact value of c, so we simply say that a family is *k-independent,* if it is (k, c)-independent for some c.

Observation:

1. If $\mathcal{H}$ is (k, c)-independent for $k > 1$, then it is also $(k-1, c)$-independent.
2. If $\mathcal{H}$ is $(2, c)$-independent, then it is c-universal.

Constructions from linear congruence

Definition: For any prime p and $m \leq p$, we define the family of linear functions $\mathcal{L} = \{h_{a,b} \mid a, b \in [p]\}$ from $[p]$ to $[m]$, where $h_{a,b}(x) = ((ax + b) \bmod p) \bmod m$.

Theorem: The family $\mathcal{L}$ is 2-universal.

Proof: Let x, y be two distinct numbers in $[p]$. First, we will examine the behavior of linear functions taken modulo p without the final modulo m. For an arbitrary pair $(a, b) \in [p]^2$ of parameters, we define

$$\begin{aligned} r &= (ax + b) \bmod p, \\ s &= (ay + b) \bmod p. \end{aligned}$$

This maps pairs $(a, b) \in [p]^2$ to pairs $(r, s) \in [p]^2$. We can verify that this mapping is bijective: for given (r, s), we can solve the above equations for (a, b). Since integers modulo p form a field, the system of linear equations (which is regular thanks to $x \neq y$) has a unique solution.

So we have a bijection between (a, b) and (r, s). This makes picking (a, b) uniformly at random equivalent to picking (r, s) uniformly at random.

Now, we resurrect the final modulo m. Let us count *bad pairs* (a, b), for which we have $h_{a,b}(x) = h_{a,b}(y)$. These correspond to bad pairs (r, s) such that $r \equiv s$ modulo m. Now we fix r and count s for which (r, s) is bad. If we partition the set $[p]$ to m-tuples, we get $\lceil p/m \rceil$ m-tuples, last of which is incomplete. A complete m-tuple contains exactly one number congruent to r, the final m-tuple one or zero.

Therefore for each r, we have at most $\lceil p/m \rceil \leq (p + m - 1)/m$ bad pairs. Since $m \leq p$, this is at most $2p/m$. Altogether, we have p choices for r, so the number of all bad pairs is at most $2p^2/m$.

As there are p^2 pairs in total, the probability that a pair (r, s) is bad is at most $2/m$. Since there is a bijection between pairs (a, b) and (r, s), the probability of a bad pair (a, b) is the same. The family $\mathcal{L}$ is therefore 2-universal. □

Surprisingly, a slight modification of the family yields even 1-universality.

Definition: For any prime p and $m \leq p$, we define the family of linear functions $\mathcal{L}' = \{h_{a,b} \mid a, b \in [p] \wedge a \neq 0\}$ from $[p]$ to $[m]$, where $h_{a,b}(x) = ((ax + b) \bmod p) \bmod m$.

Theorem: The family $\mathcal{L}'$ is 1-universal.

Proof: We shall modify the previous proof. The bijection between pairs (a, b) and (r, s) stays. The requirement that $a \neq 0$ translates to $r \neq s$. We therefore have $p(p-1)$ pairs (r, s). How many of those are bad?

For a single r, we have one less bad choice of s (in the tuple containing r), so at most $\lceil p/m \rceil - 1 \leq (p + m - 1)/m - 1 = (p - 1)/m$ bad choices. For all r together, we have at most $p(p-1)/m$ bad pairs, so the probability that a random pair is bad is at most $1/m$. □

Now, we turn back to the original linear family $\mathcal{L}$.

Theorem: The family $\mathcal{L}$ is $(2, 4)$-independent.

Proof: We fix $x, y \in [p]$ distinct and $i, j \in [m]$. To verify $(2, 4)$-independence, we need to prove that $\Pr[h_{a,b}(x) = i \wedge h_{a,b}(y) = j] \leq 4/m^2$ for a random choice of parameters (a, b).

As in the proof of universality of $\mathcal{L}$, we consider the bijection between pairs (a, b) and (r, s). Hence the event $h_{a,b}(x) = i$ is equivalent to $r \equiv i$ modulo m and $h_{a,b}(y) = j$ is the same as $s \equiv j$, again modulo m. As the choices of r and s are independent, we can consider each event separately.

Let us count the values of r which are congruent to i modulo m. If we divide the set $[p]$ to m-tuples, each m-tuple contains at most one such r. As we have $\lceil p/m \rceil$ m-tuples and p possible values of r, the probability that $r \equiv i$ is at most $\lceil p/m \rceil / p \leq (p + m - 1)/mp$. As $m \leq p$, we have $p + m - 1 \leq 2p$, so the probability is at most $2p/mp = 2/m$.

Similarly, the probability that $s \equiv j$ is at most $2/m$. Since the events are independent, the probability of their conjunction is at most $4/m^2$. This is what we needed for $(2, 4)$-independence. □

Composition of function families

The proofs of 2-universality and (2,4)-independence of the family $\mathcal{L}$ have a clear structure: they start with a $(2, 1)$-independent family from a field to the same field. Then they reduce the family's functions modulo m and prove that independence/universality degrades only slightly. This approach can be easily generalized.

Lemma M (composition modulo m): Let $\mathcal{H}$ be a $(2, c)$-independent family of functions from $\mathcal{U}$ to $[r]$ and $m < r$. Then the family $\mathcal{H} \bmod m = \{h \bmod m \mid h \in \mathcal{H}\}$ is $2c$-universal and $(2, 4c)$-independent.

Proof: Consider universality first. For two given items $x_1 \neq x_2$, we should show that $\Pr_{h\in\mathcal{H}}[h(x_1) = h(x_2)] \leq 2c/m$. The event $h(x_1) \bmod m = h(x_2) \bmod m$ can be written as a union of disjoint events $h(x_1) = i_1 \wedge h(x_2) = i_2$ over all pairs (i_1, i_2) such that i_1 is congruent to i_2 modulo m. So we have

$$\Pr[h(x_1) = h(x_2)] = \sum_{i_1 \equiv i_2} \Pr[h(x_1) = i_1 \wedge h(x_2) = i_2].$$

By $(2, c)$-independence of $\mathcal{H}$, every term of the sum is at most c/r^2. How many terms are there? We have r choices of i_1, for each of them there are at most $\lceil r/m \rceil \leq (r + m - 1)/m \leq 2r/m$ congruent choices of i_2. Therefore the whole sum is bounded by $r \cdot (2r/m) \cdot (c/r^2) = 2c/m$ as we needed.

For 2-independence, we proceed in a similar way. We are given two items $x_1 \neq x_2$ and two buckets $j_1, j_2 \in [m]$. We are bounding

$$\Pr_h[h(x_1) \bmod m = j_1 \wedge h(x_2) \bmod m = j_2] = \sum_{\substack{i_1 \equiv j_1 \\ i_2 \equiv j_2}} \Pr[h(x_1) = i_1 \wedge h(x_2) = i_2].$$

Again, each term of the sum is at most c/r^2. There are at most $\lceil r/m \rceil \leq (r + m - 1)/m$ choices of i_1 and independently also of i_2. The sum is therefore bounded by

$$\frac{c}{r^2} \cdot \left(\frac{r + m - 1}{m}\right)^2 \leq \frac{c}{r^2} \cdot \frac{(2r)^2}{m^2} = \frac{4c}{m^2},$$

which guarantees $(2, 4c)$-independence of $\mathcal{H} \bmod m$. □

The previous proof can be extended to general k-independence. However, instead of $4c$, we get $2^k c$, which grows steeply with k. This is unavoidable for m close to r, but if we assume $r \gg m$, we can actually improve the constant even beyond the previous lemma.

Lemma K (k-independent composition modulo m): Let $\mathcal{H}$ be a (k, c)-independent family of functions from $\mathcal{U}$ to $[r]$ and m integer such that $r \geq 2km$. Then the family $\mathcal{H} \bmod m = \{h \bmod m \mid h \in \mathcal{H}\}$ is $(k, 2c)$-independent.

Proof: Let us extend the previous proof. Probability that $h(x_j) = i_j$ for all k values of j is bounded by:

$$\frac{c}{r^k} \cdot \left(\frac{r+m-1}{m}\right)^k = \frac{c}{m^k} \cdot \left(\frac{r+m-1}{r}\right)^k \le \frac{c}{m^k} \cdot \left(1 + \frac{m}{r}\right)^k .$$

So the family $\mathcal{H} \bmod m$ is (k, c')-independent for

$$c' = c \cdot \left(1 + \frac{m}{r}\right)^k .$$

Since $\mathrm{e}^x \ge 1 + x$ for all real x, we have $(1 + m/r)^k \le (\mathrm{e}^{m/r})^k = \mathrm{e}^{mk/r}$. By premise of the lemma, $mk/r \le 1/2$, so $\mathrm{e}^{mk/r} \le \mathrm{e}^{1/2} \le 2$. Hence $c' \le 2c$. □

Example: Let us test this machinery on the linear family $\mathcal{L}$. The family of all linear functions in a field is $(2,1)$-independent (by the bijection between (a,b) and (r,s) from the original proofs). Taken modulo m, the family is 2-universal and $(2,4)$-independent by Lemma **M**. Whenever $p \ge 4m$, it is even $(2,2)$-independent by Lemma **K**.

The drawback of the reduction modulo m is that we needed a 2-independent family to start with. If it is merely universal, the modulo can destroy universality (exercise 5). However, composing an universal family with a 2-independent family (typically $\mathcal{L}$) yields a 2-independent family.

Lemma G (general composition): Let $\mathcal{F}$ be a c-universal family of functions from $\mathcal{U}$ to $[r]$. Let $\mathcal{G}$ be a $(2,d)$-independent family of functions from $[r]$ to $[m]$. Then the family $\mathcal{H} = \mathcal{F} \circ \mathcal{G} = \{f \circ g \mid f \in \mathcal{F}, g \in \mathcal{G}\}$ is $(2,c')$-independent for $c' = (cm/r + 1)d$.

Proof: Given distinct $x_1, x_2 \in \mathcal{U}$ and $i_1, i_2 \in [m]$, we should bound

$$\Pr_{h\in\mathcal{H}}[h(x_1) = i_1 \wedge h(x_2) = i_2] = \Pr_{f\in\mathcal{F},g\in\mathcal{G}}\,[g(f(x_1)) = i_1 \wedge g(f(x_2)) = i_2].$$

It is tempting to apply 2-independence of $\mathcal{G}$ on the intermediate results $f(x_1)$ and $f(x_2)$, but unfortunately we cannot be sure that they are distinct. Still, universality of $\mathcal{F}$ should guarantee that they are almost always distinct. We will do it precisely now.

Let M be the *match* event $g(f(x_1)) = i_1 \wedge g(f(x_2)) = i_2$ and C the *collision* event $f(x_1) = f(x_2)$. For a random choice of f and g, we have

$$\begin{aligned}\Pr[M] &= \Pr[M \wedge \neg C] + \Pr[M \wedge C] \\ &= \Pr[M \mid \neg C] \cdot \Pr[\neg C] + \Pr[M \mid C] \cdot \Pr[C].\end{aligned}$$

by elementary probability. (If $\Pr[C]$ or $\Pr[\neg C]$ is 0, the conditional probabilities are not defined, but then the lemma holds trivially.) Each term can be handled separately:

$\Pr[M \mid \neg C] \le d/m^2$	by $(2,d)$-independence of $\mathcal{G}$
$\Pr[\neg C] \le 1$	trivially
$\Pr[M \mid C] \le d/m$	for $i_1 \ne i_2$: the left-hand side is 0, for $i_1 = i_2$: $(2,d)$-independence implies $(1,d)$-independence
$\Pr[C] \le c/r$	by c-universality of $\mathcal{F}$

So $\Pr[M] \le d/m^2 + cd/mr$. We want to write this as c'/m^2, so $c' = d + cdm/r = (1 + cm/r)d$. □

Corollary: If $r \ge m$ (the usual case), $\mathcal{F} \circ \mathcal{G}$ is also $(2,c')$-independent for $c' = (c+1)d$.

Polynomial hashing

So far, we constructed k-independent families only for $k = 2$. Families with higher independence can be obtained from polynomials of degree k over a field.

Definition: For any field $\mathbb{Z}_p$ and any $k \ge 1$, we define the family of polynomial hash functions $\mathcal{P}_k = \{h_{\mathbf{t}} \mid \mathbf{t} \in \mathbb{Z}_p^k\}$ from $\mathbb{Z}_p$ to $\mathbb{Z}_p$, where $h_{\mathbf{t}}(x) = \sum_{i=0}^{k-1} t_i x^i$ and the k-tuple $\mathbf{t}$ is indexed from 0.

Lemma: The family $\mathcal{P}$ is $(k,1)$-independent.

Proof: Let $x_1, \ldots, x_k \in \mathbb{Z}_p$ be distinct items and $a_1, \ldots, a_k \in \mathbb{Z}_p$ buckets. By standard results on polynomials, there is exactly one polynomial h of degree at most k such that $h(x_i) = a_i$ for every i. Hence the probability that a random polynomial of degree at most k has this property is $1/p^k$. □

Of course hashing from a field to the same field is of little use, so we usually consider the family $\mathcal{P}_k \bmod m$ instead. If we set p large enough, Lemma **K** guarantees:

Corollary: If $p \ge 2km$, the family $\mathcal{P}_k \bmod m$ is $(k,2)$-independent.

The downside is that we need time $\Theta(k)$ both to pick a function from the family and to evaluate it for a single x.

Multiply-shift hashing

Simple and fast hash functions can be obtained from the combination of multiplication with bitwise shifts. On a 32-bit machine, we multiply a 32-bit element x by a random odd number a. We let the machine truncate the result to bottom 32 bits and then we use a bitwise shift by $32 - \ell$ bits to the right, which extracts topmost ℓ bits of the truncated result. Surprisingly, this yields a good hash function from $[2^{32}]$ to $[2^\ell]$.

We provide a more general definition using the bit slice operator: a *bit slice* $x\langle a:b\rangle$ extracts bits at positions a to $b-1$ from the binary expansion of x.⟨1⟩ Therefore, $x\langle a:b\rangle = \lfloor x/2^a \rfloor \bmod 2^{b-a}$.

Definition: For any w (word length of the machine) and ℓ (width of the result), we define the multiply-shift family $\mathcal{M} = \{h_a \mid a \in [2^w], a \text{ odd}\}$ from $[2^w]$ to $[2^\ell]$, where $h_a(x) = (ax)\langle w-\ell : w\rangle$.

Claim: The family $\mathcal{M}$ is 2-universal.

Obviously, $\mathcal{M}$ is not 2-independent, because $h_a(0) = 0$ for all a. However, 2-independence requires only minor changes: increasing precision from w bits to roughly $w+\ell$ and adding a random number to make all buckets equiprobable.

Definition: Let $\mathcal{U} = [2^w]$ be the universe of w-bit words, ℓ the number of bits of result, and $w' \geq w + \ell - 1$. We define the multiply-shift family $\mathcal{M}' = \{h_{a,b} \mid a, b \in [2^{w'}], a \text{ odd}\}$ from $[2^w]$ to $[2^\ell]$, where $h_{a,b}(x) = (ax+b)\langle w'-\ell : w'\rangle$.

This is still easy to implement: for 32-bit keys and $\ell \leq 32$ bits of result, we can set $w' = 64$, compute multiplication and addition with a 64-bit result and then extract the topmost ℓ bits by shifting to the right by $64 - \ell$ bits.

Claim: The family $\mathcal{M}'$ is 2-independent.

Tabulation hashing

To describe a completely random function from $[2^u]$ to $[2^\ell]$, we need a table of 2^u entries of ℓ bits each. This is usually infeasible (and if it were, there would be no reason for hashing), but we can split the u-bit argument to multiple parts, index smaller tables by the parts, and combine the tabulated values to the final result. This leads to a simple construction of hash functions with quite strong properties.

More precisely, let the universe be $\mathcal{U} = [2^{kt}]$ for some $t \geq 1$ (number of parts) and $k \geq 1$ (part size), and $[2^\ell]$ the set of buckets. We generate random tables $T_0, \ldots, T_{t-1}$ of 2^k entries per ℓ bits.

To evaluate the hash function for $x \in \mathcal{U}$, we split x to parts $x\langle ik : (i+1)k\rangle$, where $0 \leq i < t$. We index each table by the corresponding part and we XOR the results:

$$h(x) = \bigoplus_{0 \leq i < t} T_i[\, x\langle ik : (i+1)k\rangle \,].$$

⟨1⟩ The least-significant bit is at position 0, the bit at position i has weight 2^i.

Tabulation hashing can be considered a family of hash functions, parametrized by the contents of the tables. Picking a function from the family takes time $\Theta(t \cdot 2^k)$ to initialize the tables, evaluating it takes $\Theta(t)$.

Claim: Tabulation hashing is 3-independent, but not 4-independent. (Assuming that it uses at least 2 tables.)

Nevertheless, we will see that tabulation hashing possesses some properties, which generally require higher independence.

Hashing of vectors using scalar products

So far, our universe consisted of simple numbers. We now turn our attention to hashing of more complex data, namely d-tuples of integers. We will usually view the tuples as d-dimensional vectors over some field $\mathbb{Z}_p$.

Similarly to the family of linear functions, we can hash vectors by taking scalar products with a random vector.

Definition: For a prime p and vector size $d \geq 1$, we define the family of scalar product hash functions $\mathcal{S} = \{h_{\mathbf{t}} \mid \mathbf{t} \in \mathbb{Z}_p^d\}$ from $\mathbb{Z}_p^d$ to $\mathbb{Z}_p$, where $h_{\mathbf{t}}(\mathbf{x}) = \mathbf{t} \cdot \mathbf{x}$.

Theorem: The family $\mathcal{S}$ is 1-universal. A function can be picked at random from $\mathcal{S}$ in time $\Theta(d)$ and evaluated in the same time.

Proof: Consider two distinct vectors $\mathbf{x}, \mathbf{y} \in \mathbb{Z}_p^d$. Let k be a coordinate for which $x_k \neq y_k$. As the vector product does not depend on ordering of components, we can renumber the components, so that $k = d$.

For a random choice of the parameter $\mathbf{t}$, we have (in $\mathbb{Z}_p$):

$$\Pr[h_{\mathbf{t}}(\mathbf{x}) = h_{\mathbf{t}}(\mathbf{y})] = \Pr[\mathbf{x} \cdot \mathbf{t} = \mathbf{y} \cdot \mathbf{t}] = \Pr[(\mathbf{x} - \mathbf{y}) \cdot \mathbf{t} = 0] =$$
$$= \Pr\left[\sum_{i=1}^{d} (x_i - y_i) t_i = 0\right] = \Pr\left[(x_d - y_d) t_d = -\sum_{i=1}^{d-1} (x_i - y_i) t_i\right].$$

For every choice of $t_1, \ldots, t_{d-1}$, there exists exactly one value of t_d for which the last equality holds. Therefore it holds with probability $1/p$. □

Note: There is clear intuition behind the last step of the proof: in a field, multiplication of a non-zero number by a uniformly random number yields a uniformly random number; similarly, adding a uniformly random number to any number yields a uniformly random result.

What if the field is too large and we want to reduce the result modulo some m? To use Lemma **M**, we need a 2-independent family, but $\mathcal{S}$ is merely universal.

We can use Lemma **G** and compose $\mathcal{S}$ with the family $\mathcal{L}$ of linear functions from $\mathbb{Z}_p$ to $[m]$. As $\mathcal{L}$ is $(2,4)$-independent, $\mathcal{S} \circ \mathcal{L}$ is $(2,8)$-independent. If $p \geq 4m$, $\mathcal{L}$ is $(2,2)$-independent and $\mathcal{S} \circ \mathcal{L}$ $(2,c')$-independent for $c' = (1 + 1/4) \cdot 2 = 5/2$.

Alternatively, we can add an additive constant to the scalar product:

Definition: For a prime p, vector size $d \geq 1$, we define the family of scalar product hash functions $\mathcal{S}' = \{h_{\mathbf{t},\beta} \mid \mathbf{t} \in \mathbb{Z}_p^d, \beta \in \mathbb{Z}_p\}$ from $\mathbb{Z}_p^d$ to $\mathbb{Z}_p$, where $h_{\mathbf{t},\beta}(x) = \mathbf{t} \cdot \mathbf{x} + \beta$.

Theorem: If $p \geq 4m$, the family $\mathcal{S}'$ mod m is $(2,2)$-independent. A function can be picked at random from the family in time $\Theta(d)$ and evaluated in the same time.

Proof: By exercise 9, the family $\mathcal{S}'$ is $(2,1)$-independent. By Lemma **K**, $\mathcal{S}'$ mod m is $(2,2)$-independent. □

Rolling hashes from polynomials

Another construction of hashing functions of vectors is based on polynomials, but this time the role of coefficients is played by the components of the vector and the point where the polynomial is evaluated is chosen randomly.

Definition: For a prime p and vector size d, we define the family of polynomial hash functions $\mathcal{R} = \{h_a \mid a \in \mathbb{Z}_p\}$ from $\mathbb{Z}_p^d$ to $\mathbb{Z}_p$, where $h_a(\mathbf{x}) = \sum_{i=0}^{d-1} x_{i+1} \cdot a^i$.

Theorem: The family $\mathcal{R}$ is d-universal. A function can be picked from $\mathcal{R}$ at random in constant time and evaluated for a given vector in $\Theta(d)$ time.

Proof: Consider two vectors $\mathbf{x} \neq \mathbf{y}$ and a hash function h_a chosen at random from $\mathcal{R}$. A collision happens whenever $\sum_i x_{i+1} a^i = \sum_i y_{i+1} a^i$. This is the same condition as $\sum_i (\mathbf{x} - \mathbf{y})_{i+1} a^i = 0$, that is if the number a is a root of the polynomial $\mathbf{x} - \mathbf{y}$. Since a polynomial of degree at most d can have at most d roots (unless it is identically zero), the probability that a is a root is at most d/p. This implies d-universality. □

As usual, we can reduce the range of the function modulo m. However, it is better to compose the family $\mathcal{R}$ with the $(2,4)$-independent family $\mathcal{L}$ of linear functions. Not only we get a 2-independent family as a result, but Lemma **G** guarantees that if p is sufficiently large, the big constant from d-universality disappears. Also, for large p, $\mathcal{L}$ becomes $(2,2)$-independent.

Corollary: Given a prime p and the number of buckets m such that $p \geq 4dm$, the compound family $\mathcal{R} \circ \mathcal{L}$ is $(2, 5/2)$-independent.

Hash functions of this kind play important role in the *Rabin-Karp string search algorithm.* Suppose that we are searching for a d-character substring ν (the needle) in a long text σ (the haystack). We pick a hash function h and calculate the hash $h(\nu)$ of the needle.

Then we slide a window of size d over the haystack and for each position of the window, we hash the contents of the window. Only if the hash of the window is equal to $h(\nu)$, we compare the window with the needle character-by-character.

For this algorithm, we need a hash function which can be recalculated in constant time when the window shifts one position to the right. Such functions are called *rolling hashes* and they are usually chosen from the family $\mathcal{R}$. Compare hashes of the window at positions j and $j+1$ (we read each window from right to left):

$$\begin{aligned} H_j &= h(\sigma_j, \ldots, \sigma_{j+d-1}) = \sigma_j a^{d-1} + \sigma_{j+1} a^{d-2} + \ldots + \sigma_{j+d-1} a^0, \\ H_{j+1} &= h(\sigma_{j+1}, \ldots, \sigma_{j+d}) = \sigma_{j+1} a^{d-1} + \sigma_{j+2} a^{d-2} + \ldots + \sigma_{j+d} a^0, \end{aligned}$$

We can observe that $H_{j+1} = aH_j - \sigma_j a^d + \sigma_{j+d}$, everything calculated in the field $\mathbb{Z}_p$. This is constant-time if we pre-calculate a^d.

Hashing strings

Finally, let us consider hashing of strings of variable length, up to a certain limit L. We will pad the strings to length exactly L by appending blank characters, reducing the problem to hashing of L-tuples.

We have to make sure that a blank does not occur anywhere else in the string — otherwise we get systematic collisions. The obvious choice is to encode the blank as 0, which allows to skip the parts of hash function which deal with blanks altogether.

The L-tuples can be hashed using the family of polynomial-based rolling hash functions. For large L, this is preferred over scalar-product hashing, which requires to generate a random L-component vector of parameters.

Exercises

1. Show that the family of all constant functions from $\mathcal{U}$ to $[m]$ is 1-independent.
2. Show that the family $\mathcal{L}$ is not 1-universal. Find the smallest c such that it is c-universal.
3. What if we modify the definition of $\mathcal{L}$, so that it forces $b = 0$? Is the modified family c-universal for some c? Is it 2-independent?
4. Prove that the family $\mathcal{L}$ is not 3-independent.
5. Show that taking an universal family modulo m sometimes destroys universality. Specifically, show that for each c and $m > 1$, there is a family $\mathcal{H}$ from some $\mathcal{U}$ to the same $\mathcal{U}$ which is 2-universal, but $\mathcal{H}$ mod m is not c-universal.

6. Prove that the family $\mathcal{M}$ is 2-universal.

7.* Prove that the family $\mathcal{M}'$ is $(2, c)$-independent for some c.

8. Prove that tabulation hashing is 3-independent, but not 4-independent (if there are at least two tables).

9. Show that the family $\mathcal{S}'$ is $(2, 1)$-independent.

10. Analyze expected time complexity of the Rabin-Karp algorithm, depending on haystack length, needle length, and the number of buckets.

11. Consider tabulation hashing for d-tuples. How good it is?

6.2 Cuckoo hashing

Sketch

We have two functions f, g mapping the universe $\mathcal{U}$ to $[m]$. Each bucket contains at most one item. An item $x \in \mathcal{U}$ can be located only in buckets $f(x)$ and $g(x)$. Lookup checks only these two buckets, so it takes constant time in the worst case.

Insert looks inside both buckets. If one is empty, we put the new item there. Otherwise we „kick out“ an item from one of these buckets and we replace it by the new item. We place the kicked-out item using the other hash function, which may lead to kicking out another item, and so on. After roughly $\log n$ attempts (this is called insertion timeout), we give up and rehash everything with new choice of f and g.

Theorem: Let $\varepsilon > 0$ be a fixed constant. Suppose that $m \geq (2 + \varepsilon)n$, insertion timeout is set to $\lceil 6 \log n \rceil$, and f, g chosen at random from a $\lceil 6 \log n \rceil$-independent family. Then the expected time complexity of INSERT is $\mathcal{O}(1)$, where the constant in $\mathcal{O}$ depends on ε.

Note: Setting the timeout to $\lceil 6 \log m \rceil$ also works.

Note: It is also known that a 6-independent family is not sufficient to guarantee expected constant insertion time, while tabulation hashing (even though it is not 4-independent) is sufficient.

6.3 Linear probing

Open addressing is a form of hashing where each bucket contains at most one item — the items are stored in an array and the hash function produces the index of an array cell where the particular item should be stored. Of course, multiple items can collide in the same cell. So we have to consider alternative locations of items.

In general, we can define a *probe sequence* for each element of the universe. This is a sequence of possible locations of the element in the array in decreasing order of preference. The cells will be numbered from 0 to $m-1$ and indexed modulo m.

INSERT follows the probe sequence until it finds an empty cell. FIND follows the probe sequence until it either finds a matching item, or it finds an empty cell. DELETE is more complicated, since we generally cannot remove items from their cells — a probe sequence for a different item could have been interrupted. Instead, we will replace deleted items with "tombstones" and rebuild the whole table occasionally.

We will study a particularly simple probe sequence called *linear probing*. We fix a hash function h from the universe to $[m]$. The probe sequence for x will be $h(x)$, $h(x)+1$, $h(x)+2$, …, taken modulo m.

Historically, linear probing was considered inferior to other probing methods, because it tends to produce *runs* (a.k.a. *clusters*) — continuous intervals of cells occupied by items. Once a large run forms, it is probable that the next item inserted will be hashed to a cell inside the cluster, extending the cluster even further. On the other hand, linear probing is quite friendly to caches, since it accesses the array sequentially. We will prove that if we use a strong hash function and we keep the table sparse enough (let us say at most half full), the expected size of runs will be constant.

Claim (basic properties of linear probing): Suppose that $m \ge (1+\varepsilon)\cdot n$. Then the expected number of probes during an operation is:

- $\mathcal{O}(1/\varepsilon^2)$ for a completely random hash functions
- $\mathcal{O}(1/\varepsilon^{13/6})$ for hash function chosen from a 5-independent family
- $\Omega(\log n)$ for at least one 4-independent family
- $\Omega(\sqrt{n})$ for at least one 2-independent family
- $\Omega(\log n)$ for multiply-shift hashing
- $\mathcal{O}(1/\varepsilon^2)$ for tabulation hashing

We will prove a slightly weaker version of the first bound. (All restrictions can be removed at the expense of making the technical details more cumbersome.)

Theorem: Let m (table size) be a power of two, $n \le m/3$ (the number of items), h a completely random hash function, and x an item. Then the expected number of probes when searching for x is bounded by a constant independent of n, m, h, x, and the universe.

The rest of this section is dedicated to the proof of this theorem.

Without loss of generality, we can assume that $n = m/3$, since it clearly maximizes the expected number of probes. We need not worry that m is actually not divisible by 3 — errors introduced by rouding are going to be negligible.

We build a complete binary tree over the table cells. A node at height t corresponds to an interval of size 2^t, whose start is aligned on a multiple of the size. We will call such intervals *blocks.* A block is *critical* if more than $2/3 \cdot 2^t$ items are hashed to it. (We have to distinguish carefully between items *hashed* to a block by the hash function and those really *stored* in it.)

We will calculate probability (over random choice of the hash function) that a given block is critical. We will use the standard tool for estimating tail probabilities — the Chernoff inequality (the proof can be found in most probability theory textbooks).

Theorem (Chernoff bound for the right tail): Let $X_1, \ldots, X_k$ be independent random variables taking values in $\{0, 1\}$. Let further $X = X_1 + \ldots + X_k$, $\mu = \mathbb{E}[X]$, and $c > 1$. Then

$$\Pr[X \geq c\mu] \leq \left(\frac{\mathrm{e}^{c-1}}{c^c} \right)^{\mu}.$$

Lemma: Let B be a block of size 2^t. The probability that B is critical is at most q^{2^t}, where $q = (\mathrm{e}/4)^{1/3} \doteq 0.879$.

Proof: For each item x_i, we define an indicator random variable X_i, which will be 1 if x_i is hashed to the block B. The expectation of an indicator equals the probability of the indicated event, that is $2^t/m$.

$X = X_1 + \ldots + X_n$ counts the number of items hashed to B. By linearity of expectation, $\mu = \mathbb{E}[X] = \sum_i \mathbb{E}[X_i] = n \cdot 2^t/m$. As we assume that $n = m/3$, we have $\mu = 2^t/3$. So if a block is critical, we must have $X \geq 2\mu$. By the Chernoff bound with $c = 2$, this happens with probability at most $(\mathrm{e}/4)^{\mu} = (\mathrm{e}/4)^{2^t/3} = q^{2^t}$. □

Now, we are going to analyze *runs* — maximal intervals of consecutive non-empty cells (indexed modulo m). There is an empty cell before and after each run. Therefore, whenever an item is stored in a run, it must be also hashed there. Our ultimate goal is to estimate size of runs. In order to do this, we prove that a long run must contain at least one large critical block.

Lemma: Let R be a run of size at least $2^{\ell+2}$ and $B_0, \ldots, B_3$ the first 4 blocks of size 2^ℓ intersecting R. Then at least one of these blocks is critical.

Proof: The size of R is at least 4 times 2^ℓ, but R is generally not aligned on a start of a block. So R intersects between 4 and 5 blocks of size 2^ℓ. The first block B_0 contains at least one cell of R, blocks B_1 to B_3 contain 2^ℓ cells of R each. So the interval $L = R \cap (B_0 \cup \ldots \cup B_3)$ must contain at least $1 + 3 \cdot 2^\ell$ cells, all of them non-empty. Since there is an empty cell before L, each item stored in L is also hashed there.

We will show that if no block B_i is critical, there cannot be so many items hashed to L. Each non-critical block of size 2^ℓ has at most $2/3 \cdot 2^\ell$ items hashed to it, so our 4 blocks can receive at most $8/3 \cdot 2^\ell < 3 \cdot 2^\ell$ items. □

We consider the situation when we search for an item x. We start probing at position $h(x)$, so the number of probes is at most the size of the run containing the cell $h(x)$.

Lemma: Let R be a run containing $h(x)$ and $|R| \in [2^{\ell+2}, 2^{\ell+3})$. Then at least one of the following 12 blocks of size 2^ℓ is critical: the block containing $h(x)$, 8 blocks preceding it, and 3 blocks following it.

Proof: The run R intersects between 4 and 9 blocks of the given size, so it begins at most 8 blocks before the block containing $h(x)$. By the previous lemma, one of the first 4 blocks of R is critical. □

Hence, the probability that run length lies between $2^{\ell+2}$ and $2^{\ell+3}$ is bounded by the probability that one of the 12 blocks (chosen independently of the actual length of the run) is critical. By union bound and our estimate on the probability that a block is critical, we get:

Corollary: Let R be a run containing $h(x)$. The probability that $|R| \in [2^{\ell+2}, 2^{\ell+3})$ is at most $12 \cdot (\mathrm{e}/4)^{2^\ell/3} = 12 \cdot q^{2^\ell}$, where q is the constant introduced above.

Finally, we will use this to prove that the expected size of R is at most a constant. We cover all possible sizes by intervals $[2^{\ell+2}, 2^{\ell+3})$ together with an exceptional interval $[0, 3]$. We replace each size by the upper bound of its interval, which only increases the expectation. So we get:

$$\mathbb{E}[|R|] \le 3 \cdot \Pr[|R| \le 3] + \sum_{\ell \ge 0} 2^{\ell+3} \cdot \Pr[|R| \in [2^{\ell+2}, 2^{\ell+3}]].$$

We bound the former probability simply by 1 and use the previous corrolary to bound the latter probability:

$$\mathbb{E}[|R|] \le 3 + \sum_{\ell \ge 0} 2^{\ell+3} \cdot 12 \cdot q^{2^\ell} = 3 + 8 \cdot 12 \cdot \sum_{\ell \ge 0} 2^\ell \cdot q^{2^\ell} \le 3 + 96 \cdot \sum_{i \ge 1} i \cdot q^i.$$

Since the last sum converges for an arbitrary $q \in (0, 1)$, the expectation of $|R|$ is at most a constant. This concludes the proof of the theorem. □

Exercises

1. Show how to implement Delete without tombstones for linear probing.

6.4 Bloom filters

Bloom filters are a family of data structures for approximate representation of sets in a small amount of memory. A Bloom filter starts with an empty set. Then it supports insertion of new elements and membership queries. Sometimes, the filter gives a *false positive* answer: it answers YES even though the element is not in the set. We will calculate the probability of false positives and decrease it at the expense of making the structure slightly larger. False negatives will never occur.

A trivial example

We start with a very simple filter. Let h be a hash function from a universe $\mathcal{U}$ to $[m]$, picked at random from a c-universal family. For simplicity, we will assume that $c = 1$. The output of the hash function will serve as an index to an array $B[0 \ldots m-1]$ of bits.

At the beginning, all bits of the array are zero. When we insert an element x, we simply set the bit $B[h(x)]$ to 1. A query for x tests the bit $B[h(x)]$ and answers YES iff the bit is set to 1. (We can imagine that we are hashing items to m buckets, but we store only which buckets are non-empty.)

Suppose that we have already inserted items $x_1, \ldots, x_n$. If we query the filter for any x_i, it always answers YES. But if we ask for a y different from all x_i's, we can get a false positive answer if x falls to the same bucket as one of the x_i's.

Let us calculate the probability of a false positive answer. For a specific i, we have $\Pr_h[h(y) = h(x_i)] \le 1/m$ by 1-universality. By union bound, the probability that $h(y) = h(x_i)$ for some i is at most n/m.

We can ask an inverse question, too: how large filter do we need to push error probability under some $\varepsilon > 0$? By our calculation, $\lceil n/\varepsilon \rceil$ bits suffice. It is interesting that this size does not depend on the size of the universe — all previous data structures required at least $\log |\mathcal{U}|$ bits per item. On the other hand, the size scales badly with error probability: for example, a filter for 10^6 items with $\varepsilon = 0.01$ requires 100 Mb.

Multi-band filters

To achieve the same error probability in smaller space, we can simply run multiple filters in parallel. We choose k hash functions $h_1, \ldots, h_k$, where h_i maps the universe to a separate array B_i of m bits. Each pair (B_i, h_i) is called a *band* of the filter.

Insertion adds the new item to all bands. A query asks all bands and it answers YES only if each band answered YES.

We shall calculate error probability of the k-band filter. Suppose that we set $m = 2n$, so that each band gives a false positive with probability at most $1/2$. The whole filter gives a false positive only if all bands did, which happens with probability at most 2^{-k} if the functons $h_1, \ldots, h_k$ where chosen independently. This proves the following theorem.

Theorem: Let $\varepsilon > 0$ be the desired error probability and n the maximum number of items in the set. The k-band Bloom filter with $m = 2n$ and $k = \lceil \log(1/\varepsilon) \rceil$ gives false positives with probability at most ε. It requires $2n\lceil \log(1/\varepsilon) \rceil$ bits of memory and both INSERT and LOOKUP run in time $\mathcal{O}(k)$.

In the example with $n = 10^6$ and $\varepsilon = 0.01$, we get $m = 2 \cdot 10^6$ and $k = 7$, so the whole filter requires 14 Mb. If we decrease ε to 0.001, we have to increase k only to 10, so the memory consumption reaches only 20 Mb.

Optimizing parameters

The multi-band filter works well, but it turns out that we can fine-tune its parameters to improve memory consumption by a constant factor. We can view it as an optimization problem: given a memory budget of M bits, set the parameters m and k such that the filter fits in memory ($mk \leq M$) and the error probability is minimized. We will assume that all hash functions are perfectly random.

Let us focus on a single band first. If we select its size m, we can easily calculate probability that a given bit is zero. We have n items, each of them hashed to this bit with probability $1/m$. So the bit remains zero with probability $(1 - 1/m)^n$. This is approximately $p = \mathrm{e}^{-n/m}$.

We will show that if we set p, all other parameters are uniquely determined and so is the probability of false positives. We will find p such that this probability is minimized.

If we set p, it follows that $m \approx -n/\ln p$. Since all bands must fit in M bits of memory, we want to use $k = \lfloor M/m \rfloor \approx -M/n \cdot \ln p$ bands. False positives occur if we find 1 in all bands, which has probability

$$(1-p)^k = \mathrm{e}^{k\ln(1-p)} \approx \mathrm{e}^{-M/n \cdot \ln p \cdot \ln(1-p)}.$$

As e^{-x} is a decreasing function, it suffices to maximize $\ln p \cdot \ln(1-p)$ for $p \in (0,1)$. By elementary calculus, the maximum is attained for $p = 1/2$. This leads to false positive probability $(1/2)^k = 2^{-k}$.

Let us return back to the original question: given n and ε, how large a filter do we need? We have to push 2^{-k} under ε, so we set $k = \lceil \log(1/\varepsilon) \rceil$, and $M = kn/\ln 2 \approx$

$n \cdot \log(1/\varepsilon) \cdot (1/\ln 2) \doteq n \cdot \log(1/\varepsilon) \cdot 1.44$. This improves the constant from the previous theorem from 2 to circa 1.44.

Note: It is known that any approximate membership data structure with false positive probability ε and no false negatives must use at least $n \log(1/\varepsilon)$ bits of memory. The optimized Bloom filter is therefore within a factor of 1.44 from the optimum.

Single-table filters

It is also possible to construct a Bloom filter, where multiple hash functions point to bits in a shared table. (In fact, this was the original construction by Bloom.) Consider k hash functions $h_1, \ldots, h_k$ mapping the universe to $[m]$ and a bit array $B[0, \ldots, m-1]$. $\textsc{Insert}(x)$ sets the bits $B[h_1(x)], \ldots, B[h_k(x)]$ to 1. $\textsc{Lookup}(x)$ returns YES, if all these bits are set.

This filter can be analysed similarly to the k-band version. We will assume that all hash functions are perfectly random and mutually independent.

Insertion of n elements sets kn bits (not necessarily distinct), so the probability that a fixed bit $B[i]$ is unset is $(1 - 1/m)^{nk}$, which is approximately $p = \mathrm{e}^{-nk/m}$. We will find the optimum value of p, for which the probability of false positives is minimized. For fixed m, we get $k = -m/n \cdot \ln p$.

We get a false positive if all bits $B[h_i(x)]$ are set. This happens with probability approximately[2] $(1-p)^k = (1-p)^{-m/n \cdot \ln p} = \exp(-m/n \cdot \ln p \cdot \ln(1-p))$. Again, this is minimized for $p = 1/2$. So for a fixed error probability ε, we get $k = \lceil \log(1/\varepsilon) \rceil$ and $m = kn/\ln 2 \doteq 1.44 \cdot n \cdot \lceil \log(1/\varepsilon) \rceil$.

We see that as far as our approximation can tell, single-table Bloom filters achieve the same performance as the k-band version.

Counting filters

An ordinary Bloom filter does not support deletion: when we delete an item, we do not know if some of its bits are shared with other items. There is an easy solution: instead of bits, keep b-bit counters $C[0 \ldots m-1]$. $\textsc{Insert}$ increments the counters, $\textsc{Delete}$ decrements them, and $\textsc{Lookup}$ returns YES if all counters are non-zero.

However, since the counters have limited range, they can overflow. We will handle overflows by keeping the counter at the maximum allowed value $2^b - 1$, which will not be

[2] We are cheating a little bit here: the events $B[i] = 1$ for different i are not mutually independent. However, further analysis shows that they are very little correlated, so our approximation holds.

changed by subsequent insertions nor deletions. We say that the counter is *stuck.* Obviously, too many stuck counters will degrade the data structure. We will show that this happens with small probability only.

We will assume a single-band filter with one fully random hash function and m counters after insertion of n items. We will assume no deletions, which is the worst case. For a fixed counter value t, we have

$$\Pr[C[i] = t] = \binom{n}{t} \cdot \left(\frac{1}{m}\right)^t \cdot \left(1 - \frac{1}{m}\right)^{n-t},$$

because for each of $\binom{n}{t}$ t-tuples we have probability $(1/m)^t$ that the tuple is hashed to i and probability $(1 - 1/m)^{n-t}$ that all other items are hashed elsewhere. If $C[i] \geq t$, there must exist a t-tuple hashed to i and the remaining items can be hashed anywhere. Therefore:

$$\Pr[C[i] \geq t] \leq \binom{n}{t} \cdot \left(\frac{1}{m}\right)^t.$$

Since $\binom{n}{t} \leq (n\mathrm{e}/t)^t$, we have

$$\Pr[C[i] \geq t] \leq \left(\frac{n\mathrm{e}}{t}\right)^t \cdot \left(\frac{1}{m}\right)^t = \left(\frac{n\mathrm{e}}{mt}\right)^t.$$

As we already know that the optimum m is approximately $n/\ln 2$, the probability is at most $(\mathrm{e}\ln 2/t)^t$. By union bound, the probability that there exists a stuck counter is at most m-times more.

Example: A 4-bit counter is stuck when it reaches $t = 15$, which by our bound happens with probability at most $3.06\cdot 10^{-14}$. If we have $m = 10^9$ counters, the probability that any is stuck is at most $3.06\cdot 10^{-5}$. So for any reasonably large table, 4-bit counters are sufficient and they seldom get stuck. Of course, for a very long sequence of operations, stuck counters eventually accumulate, so we should preferably rebuild the structure occasionally.