Hashing: Universal and Perfect Hashing

Hashing is a great practical tool, with an interesting and subtle theory too. In addition to its use as a dictionary data structure, hashing also comes up in many different areas, including cryptography and complexity theory. In this lecture we describe two important notions: *universal hashing* and *perfect hashing*.

Objectives of this lecture

In this lecture, we want to:

- Review dictionaries and understand the formal definition and general idea of hashing
- Define and analyze *universal hashing* and its properties
- Analyze an algorithm for *static perfect hashing*

Recommended study resources

- CLRS, *Introduction to Algorithms*, Chapter 11, Hash Tables
- DPV, *Algorithms*, Chapter 1.5, Universal Hashing

1 Dictionaries, hashing, and hashtables

1.1 The Dictionary problem

One of the main motivations behind the study and hashing and hash functions is the *Dictionary* problem. A dictionary D stores a set of *items*, each of which has an associated *key*. From an algorithmic point of view, items themselves are not typically important, they can be thought of as just data associated with a key. The key is the important part for us as algorithm designers. The operations we want to support with a dictionary are:

Definition: Dictionary data type

A dictionary supports:

- `insert(item)`: add the given item (associated with its key)
- `lookup(key)`: return the item with the given key (if it exists)
- `delete(key)`: delete the item with the given key

In some cases, we don't care about inserting and deleting keys, we just want fast query times—e.g., if we were storing a literal dictionary, the actual English dictionary does not change (or changes extremely rarely). This is called the *static case*. Another special case is when we only insert new keys but never delete: this is called the *incremental case*. The general case with insertions, lookups and deletes is called the *fully dynamic case*.

For the static problem we could use a sorted array with binary search for lookups. For the dynamic we could use a balanced search tree. However, *hashtables* are an alternative approach that is often the fastest and most convenient way to solve these problems. You should hopefully already be familiar with the main ideas of hashtables from your previous studies.

1.2 Hashing and hashtables

To design and analyze hashing and hashing-based algorithms, we need to formalize the setting that we will work in. We will continue to work in the word RAM model from last lecture, so operations on word-sized integers takes constant time, and we have access to constant time indirect addressing (looking up an element of an array by its index in constant time).

The key space (the universe): The *keys* are assumed to come from some large *universe* U. Most often, when analyzed on the word RAM model, we will assume that $U = 0, \ldots, u-1$, where $u = 2^w$ is the universe size, i.e., the keys are word-sized integers.

The hashtable: There is some set $S \subseteq U$ of keys that we are maintaining (which may be static or dynamic). Let $n = |S|$. Think of n as much smaller than the size of U. We will perform inserts and lookups by having an array A of some size m, and a **hash function** $h : U \to \{0, \ldots, m-1\}$. Given an item with key x, the idea of a hashtable is that we want to store it in $A[h(x)]$. Note that if U was small then you could just store the item in $A[x]$ directly, no need for hashing! Such a

data structure is often called a direct-access array or direct-address table. The problem is that U is assumed to be very big, so this would waste memory. That is why we employ hashing.

Collisions: Recall that hashtables suffer from *collisions*, and we need a method for resolving them. A *collision* is when $h(x) = h(y)$ for two different keys x and y. For this lecture, we will assume that collisions are handled using the strategy of *separate chanining*, by having each entry in A be a list[1] containing all the colliding items. There are a number of other methods (e.g., open addressing aka probing), but for this lecture we are focusing primarily on the *hash function* itself, and separate chaining just happens to be the simplest method. To insert an item, we just add it to the list[2]. If h is a "good" hash function, then our hope is that the lists will be small. This lecture will focus on exploring what "good" hash functions look like.

The question we now turn to is: what properties are needed to achieve good performance?

Desired properties: The main desired properties for a good hashing scheme are:

1. The keys are nicely spread out so that we do not have too many collisions, since collisions affect the time to perform lookups and deletes.
2. $m = O(n)$: in particular, we would like our scheme to achieve property (1) without needing the table size m to be much larger than the number of items n.
3. The function h is fast to compute. In our analysis today we will be viewing the time to compute $h(x)$ as a constant. However, it is worth remembering in the back of our heads that h shouldn't be too complicated, because that affects the overall runtime.

Given this, the time to lookup an item x is $O(\text{length of list } A[h(x)])$. The same is true for deletes. Inserts take time $O(1)$ if we don't check for duplicates, or the same time again if we do. So, *our main goal will be to be able to analyze how big these lists get.*

Prehashing non-integer keys: One issue that we sweep under the rug in theory but that matters a lot in practice is dealing with non-integer keys. Hashtables in the real world are frequently used with data such as strings, so we want this to be applicable.

The way that we get around this in theory is to require non-integer key types to come equipped with a *pre-hash* function, i.e., a function that converts the keys reasonably uniformly into integers in the universe U. Then we can proceed as normal assuming integer keys.

Basic intuition: One way to spread items out nicely is to spread them *randomly*. Unfortunately, we can't just use a random number generator to decide where the next item goes because then we would never be able to find it again. So, we want h to be something "pseudorandom" in some formal sense.

We now present some bad news, and then some good news.

[1] Historically, separate chaining was always described by using *linked lists* to store the set of colliding items. For most theoretical purposes however, the kind of list (e.g., linked list vs. dynamic array) is irrelevant so I just say list.

[2] This assumes that the user will never insert a duplicate key. To guard against this we could first scan the list and check for duplicates before inserting.

Claim: Bad news

For any hash function h, if $|U| \geq (n-1)m+1$, there exists a set S of n items that all hash to the same location.

Proof. By the pigeonhole principle. In particular, to consider the contrapositive, if every location had at most $n-1$ items of U hashing to it, then U could have size at most $m(n-1)$. □

So, this is partly why hashing seems so mysterious — how can one claim hashing is good if for any hash function you can come up with ways of foiling it? A common but unsatisfying answer is that there are a lot of simple hash functions that work well in practice for typical sets S. But we are not a practical class, we want theoretical guarantees! What can we do if we want to have good *worst-case* guarantees?

1.3 The key idea: Random hashing

In Lecture One we reviewed one of the holy grails of algorithm design. To foil an adversary from constructing worst-case inputs to your algorithm, *introduce randomness into the algorithm!* Specifically, let's use randomization in our *construction* of h. Importantly, we must remember that the function h itself will be a deterministic function, but we will use randomness to choose *which function* h we end up with.

What we will then show is that for *any* sequence of insert and lookup operations (remember, we won't assume the set S of items inserted is random since that would give us *average-case* complexity and we want *worst-case* bounds), if we pick h in this probabilistic way, the performance of h on this sequence will be good in expectation. We will come up with different kinds of hashing schemes depending on what we mean by "good". Intuitively, the goal is to make the hash appear *as if it was a totally random function*, even though it isn't.

We will first develop the idea of *universal hashing*. Then, we will use it for an especially nice application called "perfect hashing".

2 Universal Hashing

Definition: Universal Hashing

A set of hash functions $\mathscr{H}$ where each $h \in \mathscr{H}$ maps $U \to \{0, \ldots, m-1\}$ is called **universal** (or is called a *universal family*) if for all $x \neq y$ in U, we have

$$\Pr_{h \in \mathscr{H}}[h(x) = h(y)] \leq 1/m. \tag{1}$$

Make sure you understand the definition! This condition must hold for *every pair* of distinct keys $x \neq y$, and the randomness is over the choice of the actual hash function h from the set

$\mathcal{H}$. Here's an equivalent way of looking at this. First, count the number of hash functions in $\mathcal{H}$ that cause x and y to collide. This is

$$|\{h \in \mathcal{H} | h(x) = h(y)\}|.$$

Divide this number by $|H|$, the number of hash functions. This is the probability on the left hand side of (1). So, to show universality you want

$$\frac{|\{h \in \mathcal{H} \mid h(x) = h(y)\}|}{|H|} \leq \frac{1}{m}$$

for every $x \neq y \in U$. Here are some examples to help you become comfortable with the definition.

Example

The following three hash families with hash functions mapping the set $\{a, b\}$ to $\{0, 1\}$ are universal, because at most $1/m$ of the hash functions in them cause a and b to collide, were $m = |\{0, 1\}|$.

	a	b
h_1	0	0
h_2	0	1

	a	b
h_1	0	1
h_2	1	0

	a	b
h_1	0	0
h_2	1	0
h_3	0	1

On the other hand, these next two hash families are not, since a and b collide with probability more than $1/m = 1/2$.

	a	b
h_1	0	0
h_3	1	1

	a	b	c
h_1	0	0	1
h_2	1	1	0
h_3	1	0	1

2.1 Using Universal Hashing

Theorem 1: Universal hashing

If $\mathcal{H}$ is universal, then for any set $S \subseteq U$ of size n, for any key $x \in S$ (e.g., that we might want to lookup), if h is drawn randomly from $\mathcal{H}$, the **expected** number of collisions between x and other keys in S is less than n/m.

Proof. Each $y \in S$ ($y \neq x$) has at most a $1/m$ chance of colliding with x by the definition of universal. So, let the random variable $C_{xy} = 1$ if x and y collide and 0 otherwise. Let C_x be the random variable denoting the total number of collisions for x. So,

$$C_x = \sum_{\substack{y \in S \\ y \neq x}} C_{xy}.$$

We know $\mathbb{E}[C_{xy}] = \Pr(x \text{ and } y \text{ collide}) \leq 1/m$. Therefore, by linearity of expectation,

$$\mathbb{E}[C_x] = \sum_{\substack{y \in S \\ y \neq x}} \mathbb{E}[C_{xy}] \leq \frac{|S|-1}{m} = \frac{n-1}{m},$$

which is less than n/m. □

When a table is storing n items in m slots, this quantity n/m represents how "full" the table is and shows up frequently in the anaylsis of hashing, so it gets a name. $\alpha = n/m$ is called the *load factor* of the hashtable. We now immediately get the following theorem.

Theorem

Insert, lookup, and delete, on a hashtable using universal hashing with separate chaining cost $\Theta(1+\alpha)$ time in expectation.

Proof. The runtime for insert, lookup, and delete, for a key x is proportional to the number of items in the list at slot $A[h(x)]$ (plus a constant cost to compute the hash function). Suppose x is not currently in the hashtable, then by Theorem 1 the expected number of items in $A[h(x)]$ is the number of items in the hashtable that collide with x, which is less than $(n+1)/m = \Theta(1+\alpha)$. Similarly if x is currently in the table then then the number of items in $A[h(x)]$ is the the number of items in the hashtable that collide with x plus one (itself, since x is definitely at location $A[h(x)]$), so by Theorem 1 the cost is again at most $\Theta(1+n/m) = \Theta(1+\alpha)$. □

Corollary

For any sequence of L insert, lookup, and delete operations in which there are at most m keys in the hashtable at any one time, using separate chaining with universal hashing, the expected total cost of the L operations is only $O(L)$.

Proof. Since there are at most m keys in the table at any time, $\alpha \leq 1$, so every operation costs $\Theta(1+\alpha) = \Theta(1)$ in expectation. By linearity of expectation, the expected total cost is $O(L)$. □

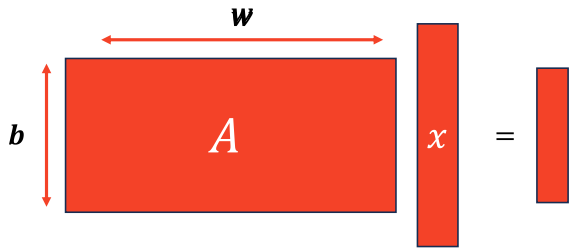
Can we construct a universal $\mathscr{H}$? If not, this this is all useless. Luckily, the answer is yes.

2.2 Constructing a universal hash family: the matrix method

Definition: The matrix method for universal hashing

Assume that keys are w-bits long, so $U = 0, \ldots, 2^w - 1$. We require that the table size m is a power of 2, so an index is b-bits long with $m = 2^b$. We pick a random b-by-w 0/1 matrix A, and define $h(x) = Ax$, where we do addition mod 2. x is interpreted as a 0/1 vector of length w, and $h(x)$ is a 0/1 vector of length b, denoting the bits of the result.

These matrices are short and fat. For instance, in picture form:



Claim: The matrix method is universal

Let $\mathscr{H}$ be the hash family generated by the matrix method. For all $x \neq y$ from U, we have

$$\Pr_{h \in \mathscr{H}}[h(x) = h(y)] = \frac{1}{2^b} = \frac{1}{m}$$

Proof. First of all, what does it mean to multiply A by x? We can think of it as adding some of the columns of A (doing vector addition mod 2) where the 1 bits in x indicate which ones to add. E.g., if $x = (1010\cdots)^\intercal$, Ax is the sum of the 1st and 3rd columns of A.

Now, take an arbitrary pair of keys x, y such that $x \neq y$. They must differ someplace, so say they differ in the i^{th} coordinate and for concreteness say $x_i = 0$ and $y_i = 1$. Imagine we first choose all the entries of A but those in the i^{th} column. Over the remaining choices of i^{th} column, $h(x) = Ax$ is fixed, since $x_i = 0$ and so Ax does not depend on the i^{th} column of A. However, each of the 2^b different settings of the i^{th} column gives a different value of $h(y)$ (in particular, every time we flip a bit in that column, we flip the corresponding bit in $h(y)$). So there is exactly a $1/2^b$ chance that $h(x) = h(y)$.

More verbosely, let $y' = y$ but with the i^{th} entry set to zero. So $Ay = Ay' +$ the i^{th} column of A. Now Ay' is also fixed now since $y'_i = 0$. Now if we choose the entries of the i^{th} column of A, we get $Ax = Ay$ exactly when the i^{th} column of A equal $A(x - y')$, which has been fixed by the choices of all-but-the-i^{th}-column. Each of the b random bits in this i^{th} column must come out right, which happens with probability $(1/2)$ each. These are independent choices, so we get probability $(1/2)^b = 1/m$. □

Efficiency Okay great, its universal! But how efficient is it? If we manually compute the matrix product, since it is an $b \times w$ matrix, this will take $O(b\,w) = O(w \lg m)$ time, which is not great, since this is actually worse than using a BST. However, this is assuming that we compute the result bit-by-bit. If we take advantage of the word RAM and use the fact that the key and rows are w-bit integers, we can compute each row-vector product in constant time with a single multiplication and improve the performance to $O(\lg m)$ time, which is about the same as a balanced BST since we assume $m = O(n)$. That means that the matrix method is not particularly practical as a hash family since we prefer hash functions that are constant time. It is mostly intended to serve as a proof that nontrivial universal families actually exist and a good first example of how to prove universality. There does exist universal families that contain hash functions that can be evaluated in constant time, but their proofs of universality are more complicated.

2.3 Another universal family: the dot-product method

Definition: The dot-product method for universal hashing

We will first require m to be a prime number. In the matrix method, we viewed the key as a vector of bits. In this method, we will instead view the key x as a vector of integers $[x_1, x_2, \ldots, x_k]$ with the requirement being that each x_i is in the range $\{0, 1, \ldots, m-1\}$ and hence $k = \log_m u$. There is a very natural interpretation of this. Just think of the key being written in base m.

To select a hash function, we choose k random numbers $r_1, r_2, \ldots, r_k$ in $\{0, 1, \ldots, m-1\}$ and define:

$$h(x) = r_1 x_1 + r_2 x_2 + \ldots + r_k x_k \mod m.$$

Note that choosing $[r_1, r_2, \ldots, r_k]$ here is equivalent to just picking a single value r in the universe and then writing it in base m as well. The proof that this method is universal follows the exact same lines as the proof for the matrix method.

Claim: The dot-product method is universal

Let $\mathcal{H}$ be the hash family generated by the dot-product method. For all $x \neq y$ from U, we have

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] = \frac{1}{m}$$

Proof. Let x and y be two distinct keys. We want to show that $\Pr_h(h(x) = h(y)) \leq 1/m$. Since $x \neq y$, it must be the case that there exists some index i such that $x_i \neq y_i$. Now imagine choosing all the random numbers r_j for $j \neq i$ first. Let $h'(x) = \sum_{j \neq i} r_j x_j$. So, once we pick r_i we will have $h(x) = h'(x) + r_i x_i$. This means that we have a collision between x and y exactly when $h'(x) + r_i x_i = h'(y) + r_i y_i \mod m$, or equivalently when

$$r_i(x_i - y_i) = h'(y) - h'(x) \mod m.$$

Since m is prime, division by a non-zero value mod m is legal (every integer between 1 and $m-1$ has a multiplicative inverse modulo m), which means there is exactly one value of r_i modulo m for which the above equation holds true, namely $r_i = (h'(y) - h'(x))/(x_i - y_i) \mod m$. So, the probability of this occurring is exactly $1/m$. □

Efficiency Is this more efficient than the matrix method? Well, our keys consist of k pieces/digits, where $k = \log_m u$, so computing the hash function takes $O(k) = O(\log_m u)$ time, which is faster than the matrix method when m is large, but slower when m is small. If we pick $m = \Theta(n)$ like usual, then this is $O(\log_n u)$, which is $O(1)$ time if $u = O(n^c)$ (same as the analysis of Radix Sort!), i.e., if the universe size is polynomial in n. So this hash family is constant time for reasonable universe sizes, but not for all universe sizes.

3 More powerful hash families

Recall that our overarching goal with universal hashing was to produce a hash function that behaved *as if it was totally random*. We can try to be more specific about what we mean. In the case of universal hashing, if we took any two distinct keys x, y from our universe, and then hashed them using our hash function from a universal family, then the probability of collision was at most $1/m$, which is the probability that we would get if the hash function was totally random! We can therefore think of universal hashing as hashing that appears to behave totally randomly if all we care about is pairwise collisions.

In some cases (for some algorithms), though, this is not good enough. Although universal hashing looks good if all we care about are collisions, there are scenarios where universal hashes appear totally not random. Lets consider an example. Suppose we are maintaining a hash table of size $m = 2$, and an evil adversary would like to cause a collision by inserting just two items. If our hash was totally random, then the adversary would have a 50/50 chance of success just by pure chance. Suppose that we use the following universal family for our hash table.

	a	b	c
h_1	0	0	1
h_2	1	0	1

In this case, the evil adversary can just first insert a, and now we are in trouble. If a goes into slot 0, then the adversary knows we have h_1 and can hence select b to insert next, causing a guaranteed collision. Otherwise, if a goes into slot 1, then the adversary can select c and cause a guaranteed collision. So, even though we used a universal hash family, it wasn't as good as a totally random hash, because the adversary was able to figure out which hash function had been selected by just knowing the hash of one key. The problem at a high level was that although this family makes collisions unlikely, it doesn't do anything to prevent the hashes of different keys from correlating. In this family, the adversary can deduce the hash values of b and c by just knowing the hash of a.

To fix this, there is a closely-related concept called pairwise independence.

Definition: Pairwise independence

A hash family $\mathcal{H}$ is *pairwise independent* if for all pairs of distinct keys $x_1, x_2 \in U$ and every pair of values $v_1, v_2 \in \{0, \ldots, m-1\}$, we have

$$\Pr_{h \in \mathcal{H}}[h(x_1) = v_1 \text{ and } h(x_2) = v_2] = \frac{1}{m^2}$$

Intuitively, pairwise independence guarantees that if we only ever look at pairs of keys in our universe, then their hash values appear to behave totally randomly! In other words, if the adversary ever learns the hash value of one key, it can not deduce any information about the hash values of the other keys, they appear totally random. Of course, it is possible that by learning the hash values of *two* keys, the adversary may be able to deduce information about other keys. To improve this, we can generalize the definition of pairwise independence to arbitrary-size sets

of keys.

Definition: k-wise independence

A hash family $\mathcal{H}$ is *k-wise independent* if for all k distinct keys $x_1, x_2, \ldots, x_k$ and every set of k values $v_1, v_2, \ldots, v_k \in \{0, \ldots, m-1\}$, we have

$$\Pr_{h \in \mathcal{H}}[h(x_1) = v_1 \text{ and } h(x_2) = v_2 \text{ and } \ldots \text{ and } h(x_k) = v_k] = \frac{1}{m^k}$$

Intuitively, if a hash family is k-wise independent, then the hash values of sets of k keys appear totally random, or, if an adversary learns the hash values of $k-1$ keys, it can not deduce any information about the hash values of any one other key.

4 Perfect Hashing

The next question we consider is: if we fix the set S (the dictionary), can we find a hash function h such that *all* lookups are constant-time? The answer is *yes*, and this leads to the topic of *perfect hashing*. We say a hash function is **perfect** for S if all lookups involve $O(1)$ deterministic work-case cost (though lookup must be deterministic, randomization is still needed to actually construct the hash function). Here are now two methods for constructing perfect hash functions for a given set S.

4.1 Method 1: an $O(n^2)$-space solution

Say we are willing to have a table whose size is quadratic in the size n of our dictionary S. Then, here is an easy method for constructing a perfect hash function. Let $\mathcal{H}$ be universal and $m = n^2$. Then just pick a random h from $\mathcal{H}$ and try it out! The claim is there is at least a 50% chance it will have no collisions.

Claim

If $\mathcal{H}$ is universal and $m = n^2$, then

$$\Pr_{h \in \mathcal{H}}(\text{no collisions in } S) \geq 1/2.$$

Proof. How many pairs (x, y) in S are there? **Answer:** $\binom{n}{2}$. For each pair, the chance they collide is $\leq 1/m$ by definition of universal. Therefore,

$$\Pr(\text{exists a collision}) \leq \frac{\binom{n}{2}}{m} = \frac{n(n-1)}{2m} \leq \frac{n^2}{2n^2} = \frac{1}{2}$$

□

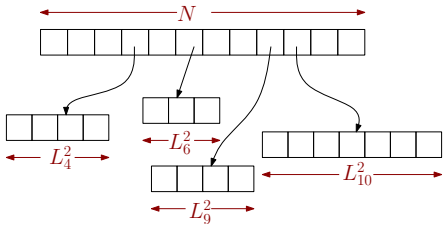
This is like the other side to the "birthday paradox". If the number of days is a lot *more* than the number of people squared, then there is a reasonable chance *no* pair has the same birthday.

So, we just try a random h from $\mathscr{H}$, and if we got any collisions, we just pick a new h. On average, we will only need to do this twice. Now, what if we want to use just $O(n)$ space?

4.2 Method 2: an $O(n)$-space solution

The question of whether one could achieve perfect hashing in $O(n)$ space was a big open question for some time, posed as "should tables be sorted?" That is, for a fixed set, can you get constant lookup time with only linear space? There was a series of more and more complicated attempts, until finally it was solved using the nice idea of universal hash functions in a 2-level scheme.

The method is as follows. We will first hash into a table of size n using universal hashing. This will produce some collisions (unless we are extraordinarily lucky). However, we will then rehash each bin using Method 1, squaring the size of the bin to get zero collisions. So, the way to think of this scheme is that we have a first-level hash function h and first-level table A, and then n second-level hash functions $h_1, \ldots, h_n$ and n second-level tables $A_1, \ldots, A_n$. To lookup a key x, we first compute $i = h(x)$ and then find the item in $A_i[h_i(x)]$. (If you were doing this in practice, you might set a flag so that you only do the second step if there actually were collisions at index i, and otherwise just put x itself into $A[i]$, but let's not worry about that here.)



Say hash function h hashes L_i keys of S to location i. We already argued (in analyzing Method 1) that we can find $h_1, \ldots, h_n$ so that the total space used in the secondary tables is $\sum_i (L_i)^2$. What remains is to show that we can find a first-level function h such that $\sum_i (L_i)^2 = O(n)$. In fact, we will show the following:

Theorem

If we pick the initial h from a universal family $\mathscr{H}$, then

$$\Pr\left[\sum_i (L_i)^2 > 4n\right] < \frac{1}{2}.$$

Proof. We will prove this by showing that $\mathbb{E}\left[\sum_i (L_i)^2\right] < 2n$. This implies what we want by Markov's inequality. (If there was even a $1/2$ chance that the sum could be larger than $4n$ then that fact by itself would imply that the expectation had to be larger than $2n$. So, if the expectation is less than $2n$, the failure probability must be less than $1/2$.)

Now, the neat trick is that one way to count this quantity is to count the number of ordered pairs that collide, including a key colliding with itself. E.g, if a bucket has `{d,e,f}`, then `d` collides

with each of `{d,e,f}`, `e` collides with each of `{d,e,f}`, and `f` collides with each of `{d,e,f}`, so we get 9. So, we have:

$$\begin{aligned}
\mathbb{E}\left[\sum_i (L_i)^2\right] &= \mathbb{E}\left[\sum_x \sum_y C_{xy}\right] && (C_{xy} = 1 \text{ if } x \text{ and } y \text{ collide, else } C_{xy} = 0)\\
&= n + \sum_x \sum_{y \neq x} \mathbb{E}[C_{xy}] \\
&\leq n + \frac{n(n-1)}{m} && (\text{where the } 1/m \text{ comes from the definition of universal})\\
&< 2n. && (\text{since } m = n)
\end{aligned}$$

□

So, we simply try random h from $\mathscr{H}$ until we find one such that $\sum_i L_i^2 < 4n$, and then fixing that function h we find the n secondary hash functions $h_1, \ldots, h_n$ as in Method 1.

Exercises: Hashing

Problem 1. Show that any pairwise independent hash family is also a universal hash family.

Problem 2. Show that the matrix method as defined above, which was universal, is **not** pairwise independent.