# Hashing
# Statistics 650/750
# Week 9 Tuesday

Christopher Genovese and Alex Reinhart

23 Oct 2018

## Announcements

- New Challenge Schedule
- Today's notes online

## Hashing: The Key Idea

The key idea of hashing is to summarize an arbitrary object with a deterministic, fixed-length "signature" with specified properties.

```
Two signatures of the previous paragraph:

    d0c62a9a66f2b9752a32867d922016bf
    e65ff981ccb4255700f88dfc4e9268c2d51358b0
```

The function that computes this signature from data is called a **hash function**; it's value is called a **hash value**, or variously, a *hash code*, *digest*, *digital signature*, or simply *hash.* A hash function must be a pure function, so the same input object is always assigned the same hash value.

**Hashing** refers to a class of algorithms that make essential use of hash functions. We will find all manner of interesting uses for hashing.

The properties that we require a hash function to satisfy vary with the application. Common criteria include:

- Two unequal objects are *unlikely* to have the same hash value.
- It is computationally hard to find two objects with the same hash value.
- Given the hash value, it is computationally hard to reconstruct the object.
- Two objects that are close together are more likely to have the same hash value than two objects that are far apart.

### Hash Functions

A hash function maps a specified "universe" of objects to a value (a *hash value*) that can be used as a representative of the object. The hash value is usually an arbitrary bit string (often viewed as an integer), though other types of values may be useful for particular applications.

What makes a good hash function? While it in depends in part on the application, but for *most* common uses, an ideal hash function would have four properties:

1. Uniformity of hash values across the output range
2. Every subset of output bits depends on every non-empty subset of input bits.
3. Avalanche Effect

   ```
   P(Output bit j changes | Input bit i changes) = 1/2  for all i,j.
   ```

4. Fast calculation

In practice, criteria 1-3 are only partially realizable and trade-off with 4.

### Basic structure of typical hash functions

A schematic for a general hash function:

```
hash(initial_state, state_bits, input):
   state = initial_state
   bits_hashed = 0
   while input is available:
       chunk = next state_bits bits of input (pad with zeroes)
       optionally pre-mix chunk
       optionally mix bits of state
       state = update(state xor chunk)  # more generally update(state,chunk)
       bits_hashed += state_bits

   return finalize(state, bits_hashed)
```

Here, update() is a specialized hash function that takes and returns `state_bits` bits, and finalize is a specialized hash function that maps the state to a bit string of the desired length.

This reduces the problem of hashing general input to the problem of hashing fixed-width integers. Let's say that `state_bits` is 64 or 32 for the purposes of discussion.

A common approach to update() is to chain together a sequence of simple *reversible* operations like

- Adding or xoring a constant
- Permuting bits
- xoring state with shifted versions of state (s xor (s » k))
- replacing chunk with a CRC code (or similar)

Example: The update component of Jenkin's Hash. It takes a 96-bit input in three 32-bit parameters a, b, c

```
a -= b; a -= c; a ^= (c>>13);
b -= c; b -= a; b ^= (a<<8);
c -= a; c -= b; c ^= (b>>13);
a -= b; a -= c; a ^= (c>>12);
b -= c; b -= a; b ^= (a<<16);
c -= a; c -= b; c ^= (b>>5);
a -= b; a -= c; a ^= (c>>3);
b -= c; b -= a; b ^= (a<<10);
c -= a; c -= b; c ^= (b>>15);
return concat(a,b,c); // string as 96-bit integer
```

## Some good hash functions for general use

The choice of hash function depends strongly on the application and is *mostly* heuristic. But the choice can have an important impact on performance. We will consider cases where one or more than one hash function is used at a time.

### Classical Methods

We can start by thinking about how to hash integers. We can represent any object as one large or a sequence of smaller integers.

- Division Method

  $h(k) = k \bmod M$ (M typically prime)

- Multiplication Method

  $h(k) = \mathrm{floor}(M\ (A\ k \bmod 1))$

  where $0 < A < 1$ (e.g., $A = (\mathrm{sqrt}(5) - 1)/2 = 0.61803\ldots$).

- Multiply-Shift Method

  Let $M = 2^m$ be a power of 2 and let W be the number of bits in a machine word. If $a < 2^W$ is an odd integer, define

  $h_a(k) = (a\ k \bmod 2^W)\ \mathrm{div}\ 2^{W-m}$

  (reduce a k modulu $2^W$ and then keep the higher order bits).

  In C-like languages this is easily expressed as

```
h_a(k) = (unsigned)(a * k) >> (W - m).
```

- Multiply-Shift-Add Method

  Improve on Multiply-Shift

  $h_{ab}(k) = ((a\ k + b) \bmod 2^W)\ \mathrm{div}\ 2^{W-M}$

  where everything is as before except $0 <= b < 2^{W-M}$ is an integer. When a, b are random integers, $h_{ab}$ forms a universal family.

For non-integers, we decompose our input and then combine the hash values of the individual pieces. For example, using Multiply-Shift, initialize a random vector a of odd integers $< 2^{2W}$ and then

$h_a(x) = (\ (\sum_{i=0}^{k-1} x_i\ a_i) \bmod 2^{2W}\ )\ \mathrm{div}\ 2^{2W}$ - M

### Modern Methods

Modern general-purpose hash functions tend to do more thorough mixing and recombinationof the inputs. These have been thoroughly tested and optimized. Reasonable choices include

- FarmHash
- Murmur3 (see digest package for R)
- CityHash
- Spooky
- JenkinsHash (?)

with the top two or three particularly recommended. You can google these.

## Cryptographic hash functions

A cryptographic hash function is used for cryptography, secure communications, and various security protocols (authentication, digital signatures, etc).

Cryptographic hash functions act as "one-way functions". Given the value of the function it is very hard to invert to find the corresponding input. To be secure, it should be very hard to find two distinct inputs with the same hash value.

Crytpographic hash functions have good collision properties, but they tend to produce long bit strings and they tend to be rather slow to compute.

Hash functions: SHA-2 and SHA-3

## Rolling Hash Function

A rolling hash function allows easy updating of the hash value with new inputs. It keeps a window and can remove and add a character from the window easily.

For example:

$$h_k(c) = c_1 a^k - 1 + c_2 a^k - 1 + ... + c_k a^0 \bmod M$$

for a constant a and input characters c. Removing and adding the end terms "shifts the window."

Similarly, given a hash function on characters, we can do

$$h(c) = \text{shift}(h(c_1), k-1) \,\text{xor} \cdots \text{xor}\, \text{shift}(h(c_k), 0)$$

with similar effect.

## Universal Hashing and Other Guarantees

Any single hash function can be "beaten" with the wrong inputs. One approach to mitigating this is to select a random hash function (or more than one) from a large family of functions that gives useful guarantees.

(We will need this for Locality-Sensitive Hashing and other statistical applications.)

A family $\mathcal{H}$ of hash functions mapping to $M$ values is said to be **universal** if $x \neq y$

$$P\{h(x) = h(y)\} \leq 1/M$$

for $h$ chosen uniformly from the family $\mathcal{H}$.

The family is *near*-universal if 1/M is replaced by c/M for some constant c.

Example: A near-universal family.

Let p > M be prime and a in {0,. . . ,p-1}. Then

$$h_a(x) = (ax \bmod p) \bmod M$$

is near universal with c = 2.

A universal family: Modifying the above, let $a \in \{1, ..., p-1\}$ and $b \in \{0, ..., p-1\}$. Then

$$h_{ab}(x) = ((ax + b) \bmod p) \bmod M$$

is universal.

Note that we often want stronger assumptions on our family: 3+-independence, independence, uniformity. These can *sometimes* be achieved.

## Hash Tables (aka Dictionaries, Maps, Associative Arrays)

A hash table (a.k.a. hash, hashmap, map, dictionary, associative array) is a data structure for associating arbitrary values with (almost) arbitrary keys.

We need to support three principal operations:

- Lookup(hash-table, object)
- Insert(hash-table, object)
- Remove(hash-table, object)

We will use a hash function hash() in ways described below.

### An Analogy

Consider a simple method of accessing a collection of objects. We assign each an integer key in the range 0..M-1 and store the objects in an array of size M.

To find an object, we access the array at its key index; to remove it, we clear the array at that index. And so forth.

This is fine as far as it goes, but what if:

- the number of potential keys is very large,
- the number of stored objects is relatively small,
- the objects are not easily mappable to integers.

Then, using an array directly like this will be impractical, inefficient, or both.

Instead, in hashing, we *derive a key* from the object and use that to access the object.

We start with a universe U of possible objects and a hash function h that maps U to the range 0..M-1.

For a value u, h(u) is called the hash value (or sometimes hash code, hash key, or similar).

There are various ways to store and access an object based on this key.

### Chaining

In *chaining*, we use the hash value as an array index, but instead of storing objects at that index, we store a *list* of objects. (The array index is commonly called a bucket; the list of objects is often called a chain.)

When there are no objects for a key, the list is empty. Otherwise, we "chain" the objects in a linked list, as in Figure 1 below.

The operations are:

- Lookup(hash-table, object): find the bucket, use linear search to find the object+data
- Insert(hash-table, object, data): if not in the list, add object+data to the head of the list
- Remove(hash-table, object): unlink from the chain.

If the hash function "randomizes" the keys sufficiently, most of the chains will be short, and lookup will be fast. But a bad hash function – one with many collisions – will lead to long chains and search that is no faster (even slower) than a simple linear search.

The hash function is the essential ingredient; we tend to use heuristics here. The performance of chaining depends on the hash function and the *load factor*, the average number of objects per bucket.

**Exercise**: Assume you have a function hash() to compute hash values. Write simple versions of lookup(), insert(), and remove() in a language of your choice for a hash table of strings.

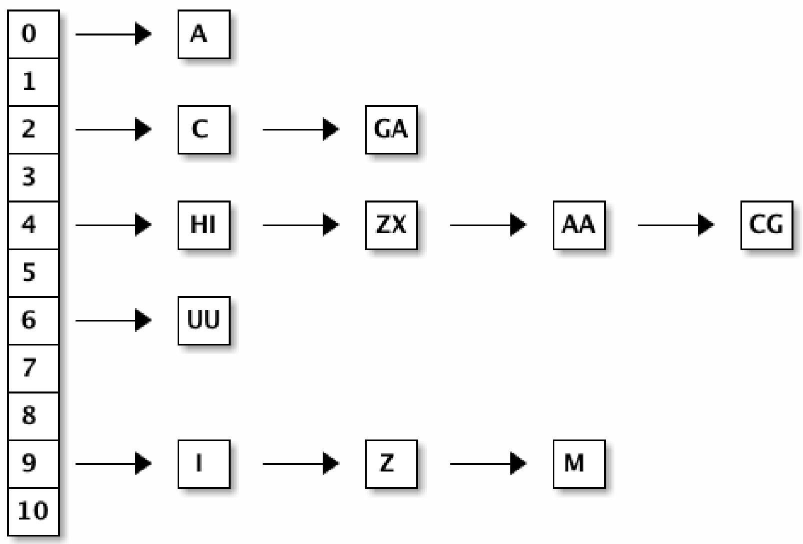

Figure 1: A hash table based on chaining.

```r
make_hash_table <- function(size) {
    return( vector("list", size) )
}

lookup <- function(hash_table, key, default=NA) {
    hash_value <- hash(key)
    bucket <- hash_table[[hash_value]]
    if ( key %in% bucket ) {
        return( bucket[[key]] )
    } else {
        return( default )
    }
}

insert <- function(hash_table, object) {
    if ( !lookup(hash_table, object) ) {
        bucket_index <- hash(object$key)
        hash_table[[bucket_index]] <- c(object, hash_table[[bucket_index]])
    }
}
```

## (Extra) Other Methods for Organizing Hash Tables

### Open Addressing

Chaining is a simple idea and is often effective, but it is not the only choice. In modern architectures, locality of reference can dominate performance depending on whether the items references fit in the fast cache memory.

In open addressing, we store the objects in the hash table itself. We systematically search the table for an object starting the index determined by that object's hash value. We then probe the table by traversing a specific sequence of slots that eventually covers the entire table.

If the hash function is h(x), then write the position after k probes as h(x,k).

Methods:

- Linear Probing: h(x,k) = (h(x) + k) mod M
- Quadratic Probing: h(x,k) = (h(x) + a k + b $k^2$) mod M
- Double Hashing: h(x,k) = (h(x) + g(x) k) mod M for another hash function g()

These methods trade off the locality of their references with their tendency to cluster the full positions in the table.

For lookup, we probe until we either find the object or an empty slot. For insertion, we do the same and put the object in the first empty slot if it is not already present. The remove operation requires some care here. (Why?)

### Cuckoo Hashing

Like open addressing but uses a different method to resolve collisions. Instead of probing as in open addressing, we use two hash functions to associate *two* indices with each object.

- On lookup, search for the object in its two indices (based on the two hash functions).
- On insertion, examine the first index for the object. If it empty, store the object. Otherwise, "bump" the object that is there to the bumped object's alternative location. This bumping continues until an empty slot is found.

  If no empty slot is found and the algorithm starts to cycle, the table is rebuilt using two new hash functions (randomly selected from a family, say).
- Deletion is handled directly.

It guarantees a worst-case constant time lookup because only two locations need to be checked. Insertion also performs well (on average, amortized over many operations) as long as the table is not too full (« 50%).

### Tabular Hashing

Partition the input object into a sequence of chunks of a specified size (e.g., bytes or words).

Create a *lookup table* T that contains uniformly random values of the chunk size.

If object x = $x_1$ $x_2$ ... $x_n$, compute

$$h(x) = T[x_1] \operatorname{xor} T[x_2] \operatorname{xor} \cdots \operatorname{xor} T[x_n].$$

This generates a universal family of hash functions with constant expected time per operation.

# Statistical Applications of Hashing

## Locality Sensitive Hashing (LSH)

In many applications of hashing, our main goal is for the hash functions is to spread hash values *uniformly* to minimize collisions. But in some applications, we want to make some collisions more likely than others.

Suppose, for example, that we had a hash function that operated on $d$-dimensional vectors of numbers in such a way that for points $x$ and $y$:

- if $x$ and $y$ are close together, $h(x)$ and $h(y)$ are more likely to be the same, and
- if $x$ and $y$ are far apart, $h(x)$ and $h(y)$ are more likely to be different.

With such an $h$ in hand, we could approximate a solution to the **nearest-neighbor problem** in high dimensions. Given $n$ data points in $d$ dimensions, we compute the hash value of each data point and a query point. Data points in the same "bucket" as the query point are likely to be near neighbors.

This is an example of **Locality Sensitive Hashing (LSH)**. Here, we do not try to avoid collisions so much as *manage* them. We want the hash values to implicitly encode the distance between points.

To make this work, we start with a *family* of hash functions and these to arrange that the probability of collision is much higher for closer points than for those more distant.

To use LSH for the nearest neighbor problem, for instance, we would LSH is a randomized algorithm that has been successfully used on problems in probabilistic clustering, approximate search, and dimension reduction.

Here is the basic idea.

A family of hash functions $\mathcal{H}$ is called $(r, c, \alpha, \beta)$ sensitive, with parameters $c > 1$ and $\alpha > \beta$, iff

1. $d(p, q) \le r$ implies $P\{h(p) = h(q)\} \ge 1 - \alpha$
2. $d(p, q) \ge cr$ implies $P\{h(p) = h(q)\} \le \beta$

where $h$ is chosen uniformly at random from $\mathcal{H}$.

We want both probabilities $\alpha$ and $\beta$ to be small, and for this to be useful, we need $1 - \alpha > \beta$, or equivalently $\alpha + \beta < 1$. So overall, we want both $c > 1$ and $\alpha + \beta < 1$ to be as small as possible, though these two values trade off.

*Example*: Assume the data points are $d$ dimensional binary vectors (all 0s and 1s). We can measure the distance between such points by *Hamming distance*, where $d(p, q)$ measures the number of coordinates that differ between points $p$ and $q$.

Let $\mathcal{H}$ contain all the functions $h_i(p) = p_i$. Then $P\{h(p) = h(q)\}$ is the proportion of coordinates in which $p$ and $q$ agree. Choosing $\alpha = r/d$ and $\beta = 1 - cr/d$ with $c > 1$ fits the bill.

LSH works by **amplifying** the gap between the collision probabilities for close and distant points. We do this by *combining several* randomly chosen hash functions.

In the example above, for instance, making $c$ large helps keep $\alpha + \beta$ small and vice versa. We'd like to be able to make *both* small.

Here's a schematic:

- Pick integers K and L.
- Choose $KL$ hash functions from $\mathcal{H}$ independently and uniformly:

$$h_{k\ell} \text{ for } 1 \le k \le K, 1 \le \ell \le L$$

- Create $L$ new "concatenated" hash functions:

$$g_\ell(q) = (h_{1\ell}(q), \ldots, h_{k\ell}(q)) \text{ for } \ell = 1, \ldots, L.$$

- Process a query using the L hash values $g_\ell(q)$.

Think of this as generating $L$ different hash tables of the data and using $L$ different queries to for search, where we use points in *any* of the buckets found.

The key idea here is that:

1. concatenation (K) ensures that dissimilar objects have low collision probability, and
2. repetition (L) ensures a high chance of finding a query.

In particular, the choices of K and L trade off. A more refined version of this, **multiprobe LSH**, adds another parameter to allow good probabilities with smaller $K$ and $L$.

The FALCONN library provides Fast C++ code for LSH, with a python interface. The R Package Falconnr (still in beta) provides a flexible R wrapper that exposes the full functionality of the library.

```
# library(devtools)
# install_github("genovese/falconnr")
X <- matrix(rnorm(10000), 1000, 10)
search.X <- LshTable(X)
q <- rnorm(10)
similar(search.X, q)                      # index of approximate NN
similar(search.X, q, points=TRUE)         # approximate NN
similar(search.X, q, k=10, points=TRUE) # 10 approximate NN's
similar(search.X, q, radius=12.4)         # ANN indices within radius
```

## Feature Hashing

Many statistical procedures (e.g., regression) depend on inner products. To extend these models to be more flexible (e.g., nonparametric or nonlinear regression), we can express the model not in terms of the original data ($x$) but in terms of **features** of the original data ($\phi(x)$), which tend to be much higher dimension.

For example, in regression or classification, moving from a linear model to a nonparametric model means going from linear functions of the data $x$ to linear functions of a basis expansion $\phi(x) = (\phi_1(x), \ldots, \phi_m(x))$, usually for a large $m$. In classification, we choose the feature vectors to get nice, nearly linear separations between our groups.

An important tool in statistical learning is the **kernel trick**: given objects $x_1, x_2, \ldots, x_n$, we define a **feature vector** $\phi(x)$ and use a kernel function

$$k(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle$$

to represent inner products. What makes this a "trick" is that if our procedure *only depends on inner products* we can work with a high-dimensional feature vector *without ever computing the features.* Put another way, once we define the kernel, that's all we need; we don't need to compute $\phi$.

But in some problems – text classification – for example, the problem is sometimes in the opposite direction: the groups in the raw data are already linearly separable but of such high dimension that the computational cost of the analysis is high. For this case, we can use the **hashing trick**.

**Feature Hashing** is based on this trick. It gives a fast way to convert arbitrary features into indices in a (usually lower-dimensional) vector or matrix. Let's illustrate with an example.

A common approach to text analysis is the *Bag of Words*, which many of you saw in homework. Consider three documents:

1. Louisa enjoys singing classic rock.

2. John enjoys singing too, mostly opera.

3. Lousia also enjoys football.

*Step 1*: Map words to indices in the bag of words

```
Louisa => 1, enjoys => 2, singing => 3, classic => 4,
rock => 5, John => 6, too => 7, mostly => 8,
opera => 9, also => 10, football => 11.
```

Step 2: Convert each document into a Bag of words, a vector counting how many times each word appears in the document. (The columns are in the index order from step 1.)

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Document 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| Document 2 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| Document 3 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |

Step 3: Do the analysis (e.g., classification, . . . )

This is fine, but if the lexicon is large and if there are many documents, we end up with a *huge* (and typically sparse) matrix.

In feature hashing, we reduce our feature vectors to a fixed, smaller dimension based on hashing. We use two hash functions to build these feature vectors faster and with less storage.

For example, choose two hash functions $h$, which maps to integers 0..m-1, and $g$, which maps to -1,1.

Suppose we pick $m = 5$. For document 1, above, we compute

| word | h(word) | g(word) |
|---|---|---|
| Louisa | 0 | 1 |
| enjoys | 3 | 1 |
| singing | 1 | -1 |
| classic | 1 | 1 |
| rock | 4 | -1 |

This gives the feature vector $(1, 2, 0, 1, 1)$, ignoring $g$, where the entry at index $k$ is the number of times $k$ appeared in the list. We then do the same for each document. To incorporate $g$, we would sum up the values of $g$ for each word times the indicator of whether that word is in the document.

In general, feature hashing builds a vector of fixed length that is indexed by a hash value of the features. This feature vector is what we can use in our statistical procedure (in place of the bag of words in this case).

Specifically, we use two hash functions $h$, mapping to $\{0, \ldots, m-1\}$, and $g$, mapping to $\{-1, 1\}$. Define

$$\phi_k^{h,g}(x) = \sum_{w:h(w)=k} g(w)\, x_w$$

$$\left\langle x, x' \right\rangle_\phi = \left\langle \phi^{h,g}(x), \phi^{h,g}(x') \right\rangle$$

and this inner product (hash kernel) is used in analysis. The purpose of the $g$ function is to reduce collisions, giving an unbiased estimator.

For our text data, the $x_w$'s are the bag-of-word entries, either 0 or 1, indicating presence of a word in a document.

In this case, the feature code looks like:

```python
def hashed_feature(data, m, h, g):
    phi = [0] * m
    for obj in data:
        k = h(obj) % m
        phi[k] += g(obj)
    return phi
```

This has many applications to document classification, protein and genome sequencing, multi-task learning.

Key advantages: strong dimension reduction, preserves sparsity, unbiased in a meaningful sense, concentration inequalities.

See also:

- `FeatureHashing` package in R
- `sklearn.feature_extraction.FeatureHasher` in `scikit-learn` for Python
- `FeatureVectorEncoder` (in `mahout`) or `HashingTF` (in `spark`) for JVM languages (Java, Clojure, Scala)

## MinHash (also cf. SimHash) for Set Comparison

For two sets A and B in X define their *Jacard similarity* by

$$J(A,B) = \frac{\#(A \cap B)}{\#(A \cup B)}.$$

Let h be a hash function that maps elements of X to integers. For $S \subset X$, define $h_{\min}(S)$ to be the member of S with the *minimum value* of h. Therefore,

$$P\{h_{\min}(A) = h_{\min}(B)\} = J(A,B).$$

Why?

Hence, $1_{\{h_{\min}(A)=h_{\min}(B)\}}$ is an unbiased estimator of J(A,B).

The idea of MinHash is to reduce the variance of this estimator by averaging.

Given K independent hash functions h$_{1,\ldots,\mathrm{hk}}$, we compute the proportion of them for which the corresponding indicator equals one. This is a lower variance, unbiased estimator of J(A,B).

Variants of this exist that use a single hash function in a clever way to achieve a similar effect.

Applications: clustering, duplicate elimination, document classification, it can also be seen as a version of LSH.

## String Matching (e.g., Finding Common Base Strings in a Genome)

Example problem: We have two (long) strings S and T (e.g., gene sequences), and we'd like to see if they share a common substring of length L.

With long strings, string comparison can be the limiting step. To see this, consider the naive approach: compare all L-substring of S to all L-substrings of T.

We can improve this in several ways with hashing. For instance, we can hash the substrings when L is long and comparing the hashes first. Two issues:

- substring hashes are expensive to compute,
- average load is O(n$^2$).

We can do this in O(n) overall time with rolling hashes:

1. Use rolling hash to compute n-L substring hashes in O(n) time.
2. Reduce effective load by using a second hash function as a *signature.* We make the second hash function g map to {0,...,$n^2$-1} but we don't keep a larger table. We insert (g(s), s) into the main hash table, using g(s) as a signature. Only if the signatures are equal do we compare the strings.

## Approximate Nearest Neighbor Search

Given a set of points $p_1$, ..., $p_n$ in high dimensions, we would like to be able to find the nearest neighbor for any query point q.

$$p_* = \text{argmin}_{p_i} d(p_i, q).$$

Applications: search, clustering, database retrieval, compression

### In two dimensions: consider the Voronoi diagram.

The *voronoi diagram* of a point set is a partitioning of the plane into "cells," where each cell is the set of points closer to one element of the point set than any other.

For nearest neighbor in O(log n) time and O(n) space, we can map each query point $q$ to the center of its Voronoi cell. Sounds good!

### In smallish dimension (say $< 7$): consider the K-d tree

### In high dimensions: uh-oh

- Voronoi diagram has size $n^{O(d)}$.
- Linear search takes O(nd) time
- KD-tree (see repository) is a common approach but only works in low-medium dimensions, near linear query time in high dimensions.

### In general: Consider *Approximate Near Neighbor* search

If there is a point p with $d(p, q) \leq r$, returns a point p' with $d(p', q) \leq cr$, for some $c > 1$.

### LSH approach to ANN search

1. For every p, hash it to L hash tables using $g_1$(p), ..., $g_L$(p) respectively.
2. For a query q, retrieve points from buckets $g_1$(q), ..., $g_L$(q) until either

   - Total number of points exceeds 2 L, or
   - All the points from the buckets have been retrieved.

   The first gives a near-neighbor search; the second gives a near-neighbor-recording search.

   In the latter case, we can then find the nearest neighbor among that smaller number of points.

Total time: O(dL)

Systems exist for a variety of distances and geometries, although extending this idea still has open questions.

# Other Applications of Hashing

## File Comparison and Verification – rsync and git

Compute hashes of file chunks for easy comparison and efficient transmission.

Key idea: only do the more expensive comparison if the hashes are equal.

```
commit 9af836e6d9cec603ee0aa75ce91472c6a7d13bc6
Author: Christopher R. Genovese <genovese@cmu.edu>
Date:   Sun Feb 1 23:58:28 2015 -0500

    Fixed require, added tests, added README, updated infrastructure

commit b755e7b1c65ae44a8a07c4bfde6fc363d6c5a36e
Author: Christopher R. Genovese <genovese@cmu.edu>
Date:   Sun Jan 25 23:04:20 2015 -0500

    Fixed file header information

commit 5df2a99483f945d831926a6c2289e4414d1300ab
Author: Christopher R. Genovese <genovese@cmu.edu>
Date:   Sun Jan 25 22:59:54 2015 -0500

    Added comparison operator handling and updated documentation
```

## Cryptographic Protocols

Hash functions are a fundamental tool in cryptography. Their many uses include:

- Authentication Protocols (OAuth, token methods)
- Message Authentication
- Digital Signatures
- Intrusion detection
- Secure ecommerce

## Signatures/Fingerprints

Use hash values as signatures that identify an object with high probability.

## Perfect Hashing

For a given, static set of objects (e.g., keywords in a programming language), define a hash function that efficiently represents the set with no collisions.

Strategy: use two hash functions, the second on a larger range and selected to ensure no collisions.

## Activity: Bloom Filters

A space-efficient, probabilistic data structure that represents a **set**. We use the Bloom filter to test whether an element belongs to the set, where negatives are certain but false positives are possible though *unlikely*.

Basic idea:

- Represent a set by an array of M bits.
- Choose K independent hash functions mapping objects to 0..M-1.
- To insert an object x into the set, set to 1 all the bits $h_1(x), \ldots, h_K(x)$.
- To lookup an object x, compute the K hash values and check if all K of those bits are set to 1. If not, x definitely has not been inserted; if so, it may have been.

Applications: many, see problem repository.

## Resources

- R Packages: FALCONNR, FeatureHashing, digest
- LSH Review Paper `http://mags.acm.org/communications/200801/?pg=119#pg119`

  (Overview `http://mags.acm.org/communications/200801/?pg=117#pg117`)
- LSH Primer `https://github.com/FALCONN-LIB/FALCONN/wiki/LSH-Primer`

  See bibliography
- Fast LSH Code `https://github.com/FALCONN-LIB/FALCONN`

  In C++ with python interface. An R package `Falconnr` wrapping this library is available at `https://github.com/genovese/falconnr`.