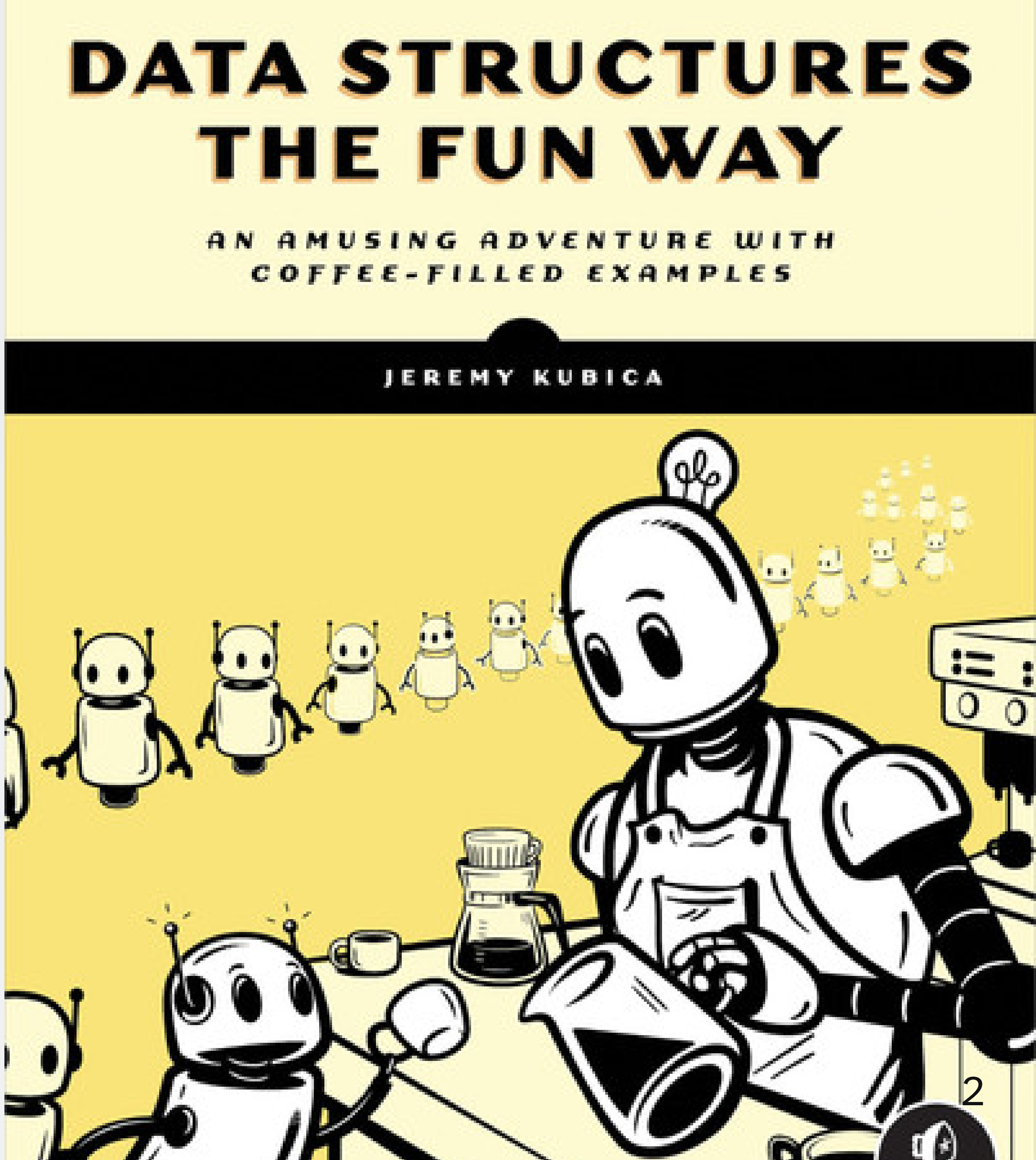
Bloom Filters

Material adapted from *Data Structures the Fun Way*, freely available to you as a Penn Student through Penn Libraries.


DATA STRUCTURES THE FUN WAY
AN AMUSING ADVENTURE WITH COFFEE-FILLED EXAMPLES
JEREMY KUBICA

`bit.ly`

`bit.ly` is a service that shortens URLs to make them easier to share:

Turn `https://www.cis.upenn.edu/~cit5940/current/` into `bit.ly/cit5940`.

Problem: Malicious Links

If you get a link from someone called `bit.ly/win_money`, you don't know where you're headed!

- `bit.ly` has a vested interest in making sure that people aren't afraid to click their links
- Need some system of refusing to create links for malicious websites.

Problem: Malicious Links

Idea: check if each link belongs to a set of known malicious sites.

Problems:

1. There are lots of malicious links. More than we could possibly store in fast memory.
2. Our malicious link lookup needs to be fast.
3. Our malicious link lookup can't let anything slip through the cracks.

Interacting with `bit.ly`

A naive solution to the problem would be to maintain some hash-indexed database (or just a hash set) containing all of the banned links. So:

1. User submits a link, making a request to `bit.ly`'s servers
2. `bit.ly` has to search its databases of malicious links for this one
3. If the link is known to be evil, reject; otherwise, produce a shortened link

Problem: to even decide whether a link is acceptable, the user has to (1) make a web request and (2) wait for a database lookup. 🐌🐢🐌🐢🐌

Pre-Filtering

Web requests and database lookups are slow and expensive. What if we had an efficient way of deciding whether or not we even need to check the bad URL?

- If we could know with absolute certainty whether or not we had to look up a URL, that would be the same as just accepting or rejecting it…
- Maybe we could accept some *uncertainty* to make the pre-lookup step faster!

Refined: Interacting with ***`bit.ly`***

1. User submits a link, making a request to an *oracle* about whether or not we need to check `bit.ly`'s servers
   i. If the *oracle* says "this is a malicious URL," double-check the database
   ii. If the *oracle* says "this URL is not part of our banned set," make no request
2. If the link is known to be evil, reject; otherwise, produce a shortened link

Observations:

- This only makes sense to do if the query to the oracle is somehow cheaper than the database lookup
- We can only trust this process if we have some guarantees about the oracle's correctness

Trusting the Oracle?

	Not Known to Be Malicious	Known to Be Malicious
Oracle Says "This Might Be Malicious"	✅ + a wasted database lookup	✅ + a database lookup
Oracle Says "This Is Definitely Not Known to Be Malicious"	✅	😠😠😠😠😠😠

- As long as our oracle never says "it's fine!" when it's not to not be fine, we'll always be correct.
- As long as our oracle *rarely* says "please check" when we didn't need to check, we won't waste much time.

Building an Oracle

Need to build something that:

1. fits in program memory/page downloads
2. can be queried quickly
3. never gives a **false negative**
4. rarely gives a **false positive**

➡️ the Bloom Filter!

Structure of a Bloom Filter

A **Bloom Filter** is an array of binary values.

- If position `i` stores a `1`, we **have** seen some item `k` such that `h(k) = i` before.
- If position `i` stores a `0`, we **have not** seen some item `k` such that `h(k) = i`.

Using a Filter

Analogy: searching for a friend at a restaurant.

- On our own, best we can do is just look at every person one by one.
- If there is a host with a great memory, we can ask them, e.g. "Have you seen someone tall, with glasses, wearing a suit?"
    - If they have seen such a person, then we go looking—might be someone else with same description
    - If they have not seen such a person, no need to bother looking!

Here, the *hashing* is our way of describing our friend and the host is the *Filter*.

Simple Bloom Filter

A **binary indicator array** mapped to by a **single hash function**.

When looking up a key, hash it and check the bit in the slot.

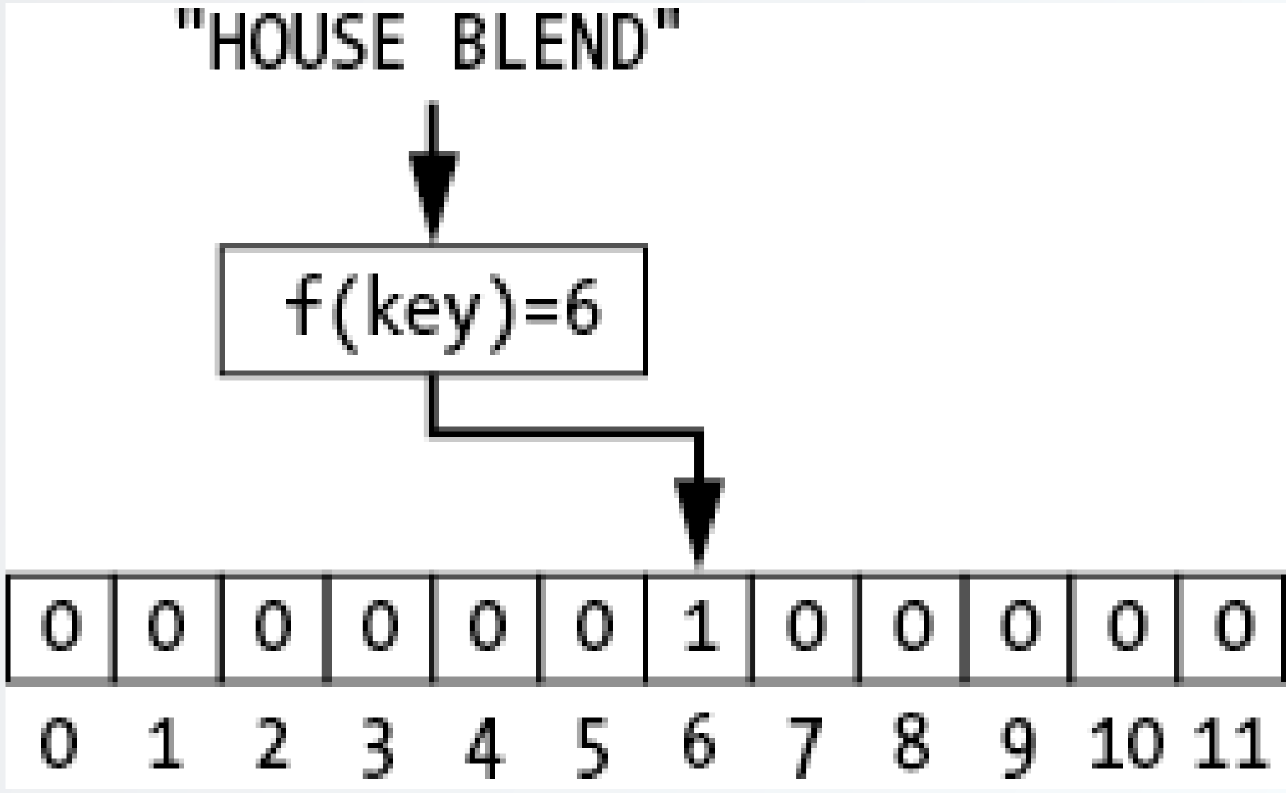
- `1` ➡️ "I've seen that (or something like that) before."
- `0` ➡️ "I have never seen that before."

Book's Example: Sampling Coffees

As a coffee lover, I want to try every blend listed in an alphabetized, 1000-page catalogue. Imagine we implement this simple single-hash-function filter for our thousand-page coffee log. Whenever we want to look up a type of coffee in our log, we first ask the filter the simple question, “Have I tried this type of coffee before?” This filter often allows us to avoid binary searching through a thousand pages if we know that we have never tried the coffee before.

At the Start, All Good

Try the new coffee called `"HOUSE BLEND"` and note it down in the table, flipping `f("HOUSE BLEND") = 6` from `0` to `1`.

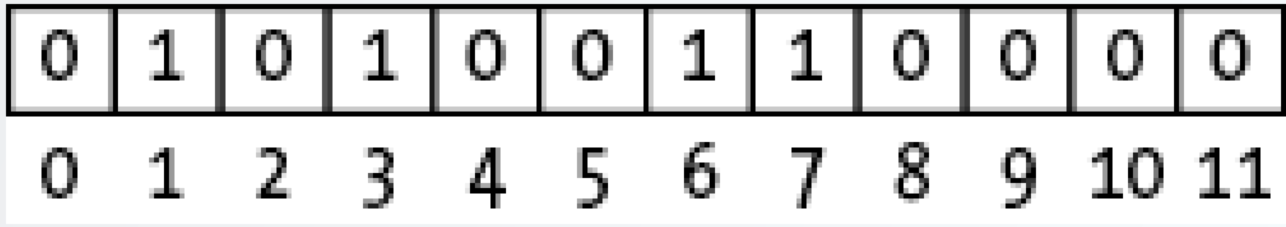


Drink a Few More...

I drink three more blends with three different hashes, and here we are.

Can answer questions:

- Have you tried `VELVET BLEND` where `f("VELVET BLEND") = 4`? **Definitely not.**
- Have you tried `SUMMER SIN` where `f("SUMMER SIN") = 7`? **Maybe...**

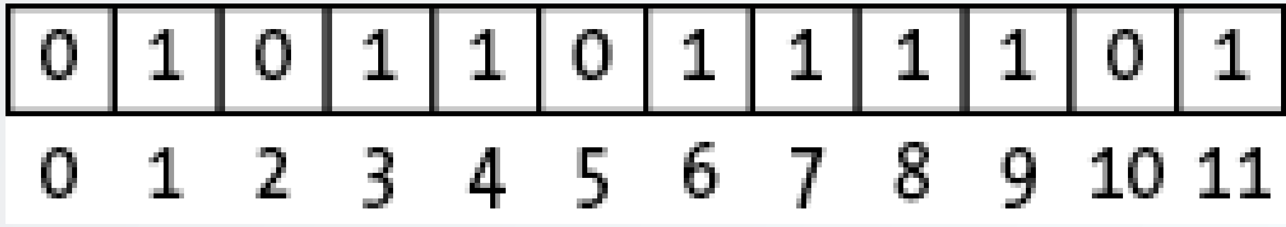


0	1	0	1	0	0	1	1	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11

No Collision Resolution

This is a table using hashing, but we don't have any probing or other collision resolution policies

As the table fills up, we end up with more coffees that hash to filled positions. This leads to a higher number of **Maybe...** answers that take more time to look up!



0	1	0	1	1	0	1	1	1	1	0	1
0	1	2	3	4	5	6	7	8	9	10	11

Collisions

We know from hashing that collisions become more likely when the load factor is higher.

➡️ Could lower the false positive rate by increasing the size of the table.

🤔 But we need to make sure that we're still keeping this small enough for working memory!

Simple Bloom Filters and Restaurant Guests

For this simplest Bloom Filter, it's analogous to asking our restaurant hosts some very foolish questions.

- Making a decision based on a single hash value is like making a decision based on a single property.
- Having more `1`s in the Bloom Filter is like dealing with a more crowded restaurant.

Multiple Properties & Bloom Filters

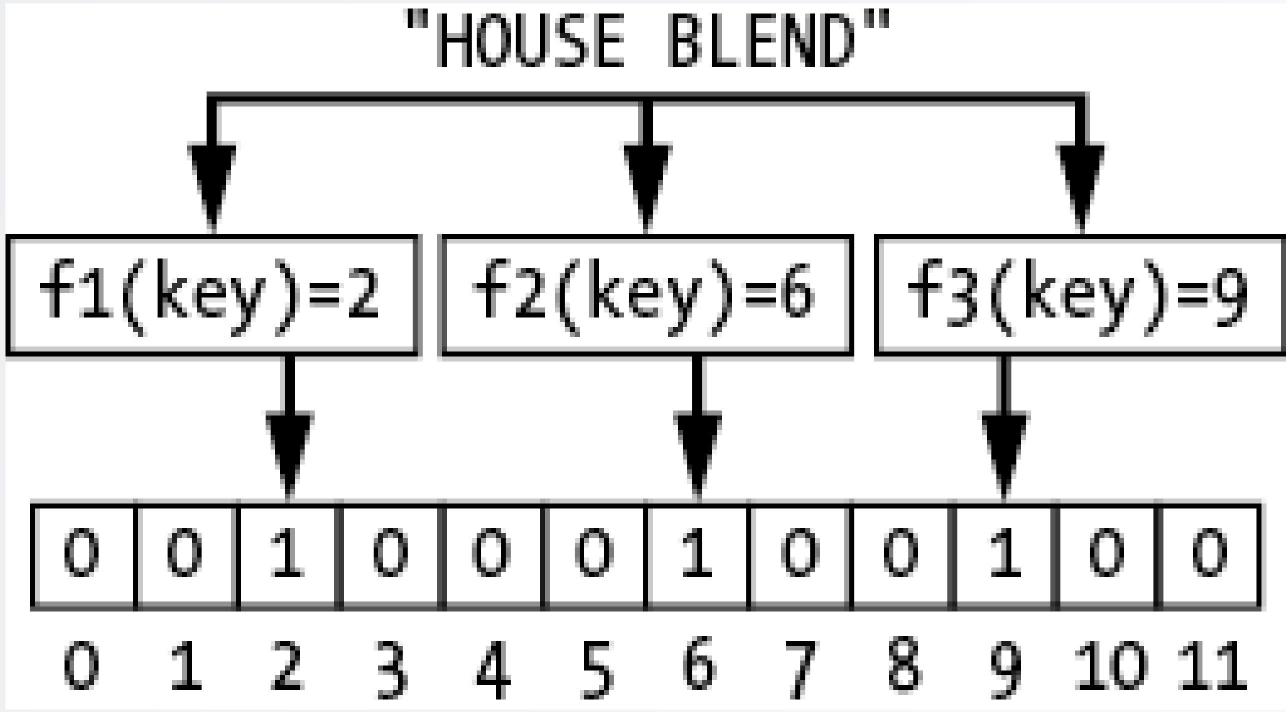
If you're looking for me in a Philadelphia restaurant, the following questions are all basically useless on their own:

- *"Have you seen my friend? He's a man."*
- *"Have you seen my friend? They're white."*
- *"Have you seen my friend? They're tall."*
- *"Have you seen my friend? They have a beard."*
- *"Have you seen my friend? They have grey hair."*

But what about asking them altogether?

Bloom Filters with Multiple Hashes

A **binary indicator array** mapped to by k **hash functions**.



Formally:

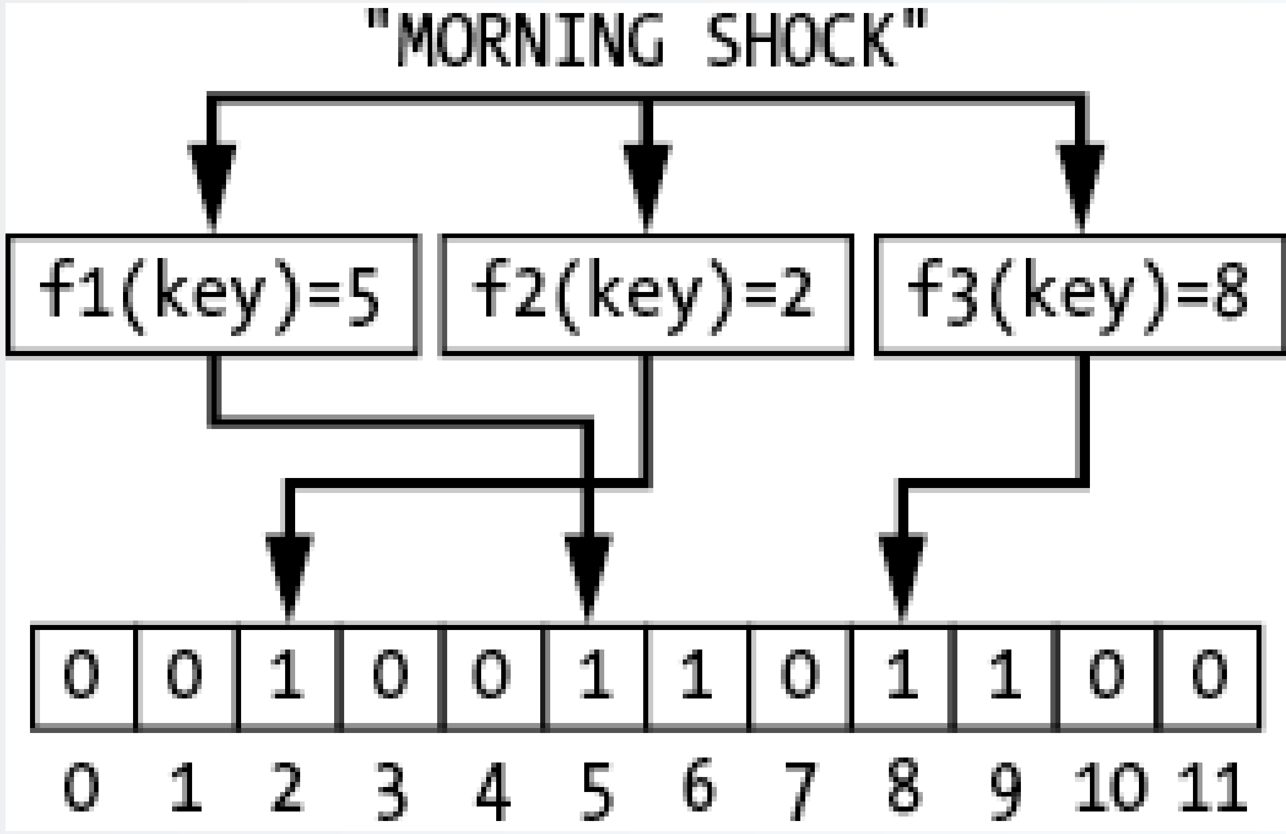
```
public void insert(Coffee c) {
  this.indicatorArray[h1(c)] = 1;
  this.indicatorArray[h2(c)] = 1;
  this.indicatorArray[h3(c)] = 1;
}

public void lookup(Coffee c) {
  return this.indicatorArray[h1(c)] == 1 &&
          this.indicatorArray[h2(c)] == 1 &&
          this.indicatorArray[h3(c)] == 1;
}
```

Why Fill Up the Filter Faster?

Now, to insert a new value, we're filling up to **three slots** per coffee inserted...

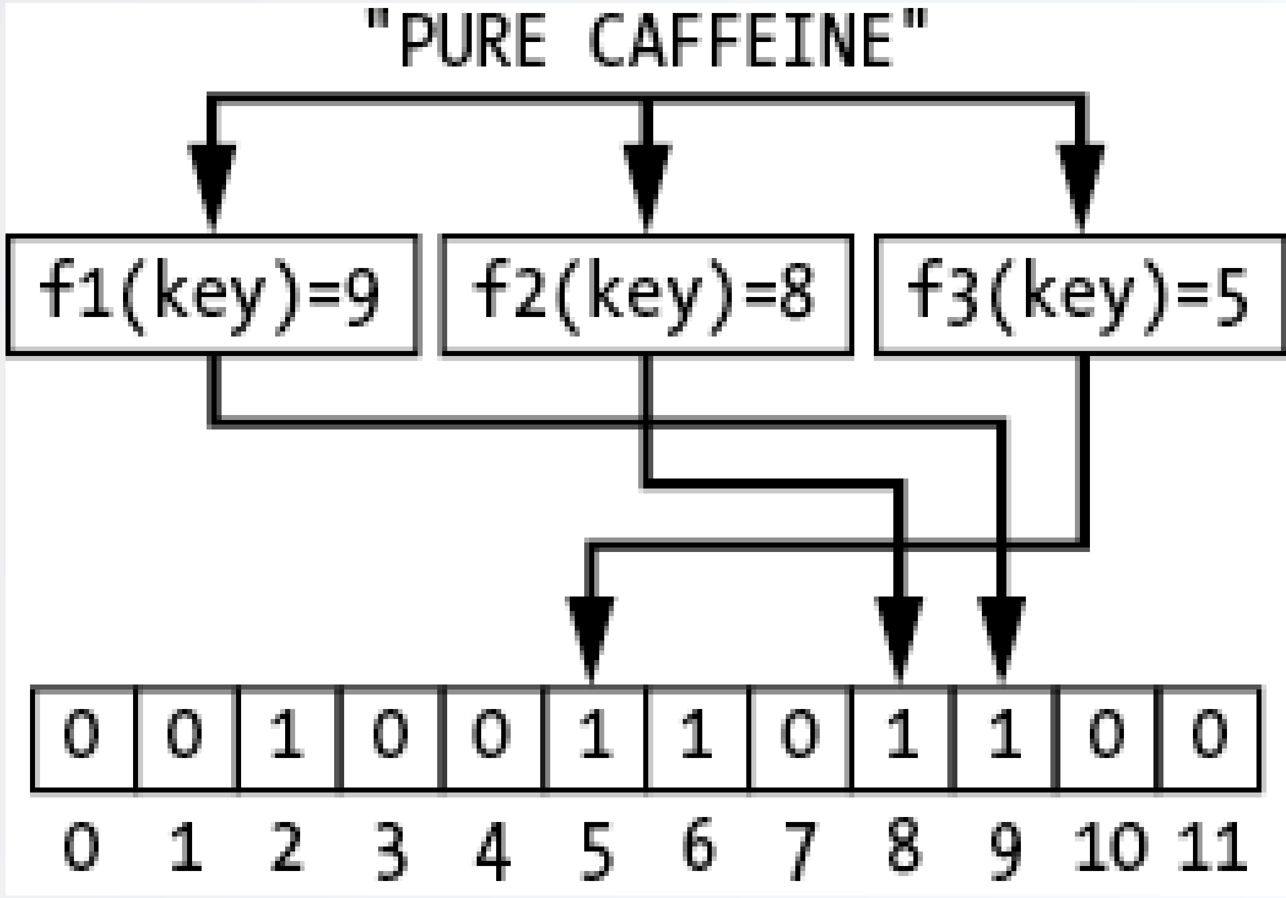
...but we still need all three slots to "hit" to be a confirmed match!



Positive Lookups

We've tried a coffee with the same properties (hashes) as "PURE CAFFEINE".

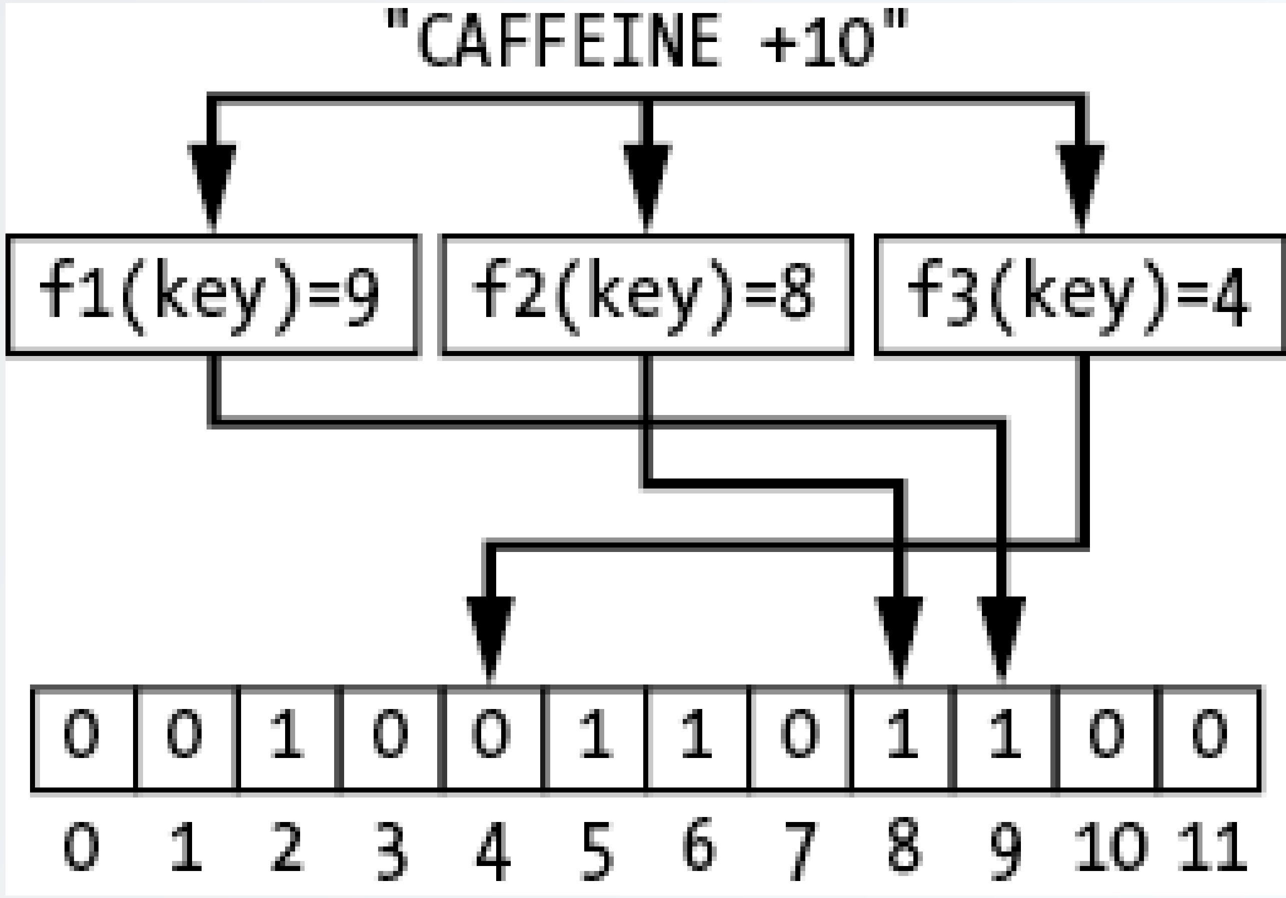
Maybe it's PURE CAFFEINE, or maybe it's a false positive.



Negative Lookups

We've never tried a coffee with the same properties (hashes) as "CAFFEINE + 10".

We have *definitely* never tried this coffee before.



False Positives

Back to the restaurant example. More likely to have a false match for your friend if:

- you ask about just one or two properties
- restaurant has a ton of people in it

Better luck in a smaller restaurant if you ask: "*I'm looking for my friend. He's a white guy, tall, has a beard, and grey hair.*"

False Positives

Happen when **all** of a key's hash values collide with previous entries.

- More likely to happen with a higher load factor
- More likely to happen with fewer hash functions

We can decrease the likelihood of a false positive by **adding more space** and **using more hash functions.**

How Common Are False Positives?

A simple approximation is the following:

$$\left(1-\left(1-\frac{1}{m}\right)^{nk}\right)^{k}$$

where m is the table size, n is the number of items inserted, and k is the number of hash functions used.

How Common Are False Positives?

$$\left(1-\left(1-\frac{1}{m}\right)^{nk}\right)^{k}$$

where m is the table size, n is the number of items inserted, and k is the number of hash functions used.

- What happens as we increase the size of the array?
- What happens as we increase the number of items inserted?
- What happens as we increase the number of hash functions used?

How Common Are False Positives?

$$\left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^{k}$$

where m is the table size, n is the number of items inserted, and k is the number of hash functions used.

- What happens as we increase the size of the array? **False positives decrease.**
- What happens as we increase the number of items inserted? **False positives increase.**
- What happens as we increase the number of hash functions used? **False positives decrease or increase.**

Empirical Results for $n = 100$

m	$k = 1$	$k = 3$	$k = 5$
200	0.3942	0.4704	0.6535
400	0.2214	0.1473	0.1855
600	0.1536	0.0610	0.0579
800	0.1176	0.0306	0.0217
1000	0.0952	0.0174	0.0094

➡️ tune the filter to your own setting!

Other Desirable Properties

Recall the `bit.ly` setting:

- Want to avoid as many web requests & database lookups as possible
- Would be nice to avoid *publishing* the list of known malicious sites for security concerns

How does a Bloom Filter help?

Compactness

A Bloom Filter is a binary/boolean array. This is **much** smaller than the underlying database.

- If $m = 1000000$, the size of the Bloom Filter is almost exactly 1MB.
- `bit.ly` can actually just send the Bloom Filter over the web at sizes like these so that you can check *on your machine* whether or not the input URL is pre-cleared.

Secrecy

`bit.ly` might not want to reveal all of the URLs that are known to be malicious

- makes it too easy for attackers to realize they're "beat" and come up with a new URL.
- the Bloom Filter makes decisions based on the *hashes* of the URLs, and hashes are non-reversable.
- sharing the Bloom Filter with the user does not reveal the evil URLs!