

Summary of the previous lecture

- ▶ **Hashing**: storing sets of objects (inserting/lookup/deleting) identified by keys from a (huge) universe
- ▶ **Goal**: get all operation run in $O(1)$ amortized time.
- ▶ **Hashing by chaining**:
 - ▶ division hash function
 - ▶ multiplication hash function
- ▶ **Hashing by open addressing**:
 - ▶ linear probing
 - ▶ quadratic probing
 - ▶ double hashing
- ▶ A good hash function should (i) be easy to compute, and (ii) distribute data uniformly

Plan

- ▶ "Classical" hashing
 - ▶ hashing by chaining
 - ▶ hashing by open addressing
- ▶ Universal hashing
- ▶ Perfect hashing (quick review)
- ▶ Cuckoo hashing
- ▶ Bloom filters
- ▶ Locality-sensitive hashing

Universal hashing

(Variant of hashing with chaining)

Universal hashing (with chaining)

- ▶ *Motivation:* avoid pathological ("adversary") datasets
 - ▶ practical applications in avoiding DDoS attacks

Denial of Service via Algorithmic Complexity Attacks

Scott A. Crosby
scrosby@cs.rice.edu

Dan S. Wallach
dwallach@cs.rice.edu

Department of Computer Science, Rice University

Abstract

We present a new class of low-bandwidth denial of service attacks that exploit algorithmic deficiencies in many common applications' data structures. Frequently used data structures have "average-case" expected running time that's far more efficient than the worst case. For example, both binary trees and hash tables can degenerate to linked lists with carefully chosen input. We show how an attacker can effectively compute such input, and we demonstrate attacks against the hash table implementations in two versions of Perl, the Squid web proxy, and the Bro intrusion detection system. Using bandwidth less than a typical dialup modem, we can bring a dedicated Bro server to its knees; after six minutes of carefully chosen packets, our Bro server was dropping as much as 71% of its traffic and consuming all of its CPU. We show how modern universal hashing techniques can yield performance comparable to commonplace hash functions while being provably secure against these attacks.

sume $O(n)$ time to insert n elements. However, if each element hashes to the same bucket, the hash table will also degenerate to a linked list, and it will take $O(n^2)$ time to insert n elements.

While balanced tree algorithms, such as red-black trees [11], AVL trees [1], and treaps [17] can avoid predictable input which causes worst-case behavior, and universal hash functions [5] can be used to make hash functions that are not predictable by an attacker, many common applications use simpler algorithms. If an attacker can control and predict the inputs being used by these algorithms, then the attacker may be able to induce the worst-case execution time, effectively causing a denial-of-service (DoS) attack.

Such algorithmic DoS attacks have much in common with other low-bandwidth DoS attacks, such as stack smashing [2] or the ping-of-death¹, wherein a relatively short message causes an Internet server to crash or misbehave. While a variety of techniques *can* be used to address these DoS attacks, com-



Breaking Murmur: Hash-flooding DoS Reloaded

Dec 14th, 2012

DISCLAIMER: Do not use any of the material presented here to cause harm. I will find out where you live, I will surprise you in your sleep and I will tickle you so hard that you will promise to behave until the end of your days.

The story so far

Last year, at 28c3, [Alexander Klink](#) and [Julian Wälde](#) presented [a way to run a denial-of-service attack](#) against web applications.

One of the most impressive demonstrations of the attack was sending crafted data to a web application. The web application would dutifully parse that data into a hash table, not knowing that the data was carefully chosen in a way so that each key being sent would cause a collision in the hash table. The result is that the malicious data sent to the web application elicits worst case performance behavior of the hash table implementation. Instead of amortized constant time for an insertion, every insertion now causes a collision and degrades our hash table to nothing more than a fancy linked list, requiring linear time in the table size for each consecutive insertion. Details can be found in the [Wikipedia article](#).

Universal hashing (with chaining)

- ▶ introduced by Carter&Wegman (1979)
- ▶ **Definition:** A family H of hash functions is called **universal** iff for any pair of keys k, l ,

$$P_{h \in H}[h(k) = h(l)] \leq 1/m$$

(Equiv., the nb of hash functions h with $h(k) = h(l)$ is $\leq |H|/m$)

- ▶ **Theorem:** the expected time (over $h \in H$) of INSERT, DELETE, LOOKUP is $O(1 + \alpha)$

NB: no assumption on the distribution of keys

Universal class of hash functions

- ▶ Choose a large prime p (larger than the maximum key)
- ▶ Let $0 \leq a, b \leq p - 1, a \neq 0$
- ▶ for a key k , define $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$
- ▶ $H_{p,m} = \{h_{a,b}\}$ is a universal family of hash functions
- ▶ *Proof idea:*
 - ▶ given a, b , values $((ak + b) \bmod p)$ are distinct for different k 's
 - ▶ for $k_1 \neq k_2$ distinct pairs (a, b) yield distinct pairs $((ak_1 + b) \bmod p, (ak_2 + b) \bmod p)$
 - ▶ $P[(ak_1 + b) \bmod p =_{\bmod m} (ak_2 + b) \bmod p] \leq 1/m$
- ▶ In practice p is often set to $2^{31} - 1$ for 32-bit numbers and to $2^{61} - 1$ for 64-bit numbers (Mersenne primes)

Example of universal hashing

- ▶ Assume we are hashing IP addresses $x_1.x_2.x_3.x_4$ with $0 \leq x_i \leq 255$
- ▶ Choose m a prime number
- ▶ Consider quadruples $a = (a_1, a_2, a_3, a_4)$ with $0 \leq a_i \leq m - 1$
- ▶ Define
$$h_a(x_1.x_2.x_3.x_4) = (a_1 \cdot x_1 + a_2 \cdot x_2 + a_3 \cdot x_3 + a_4 \cdot x_4) \bmod m$$
- ▶ $H = \{h_a\}$ is a universal family (can be proved)

Remarks

- ▶ Other (more) efficient universal hashing schemes exist, such as Multiply-shift [[Dietzfelbinger et al. A reliable randomized algorithm for the closest-pair problem. J. Algorithms, 25:19–51, 1997](#)]

$h_a: [0..2^w - 1] \rightarrow [0..2^l - 1]$ defined by

$h_a(x) = \lfloor (ax \bmod 2^w) / 2^{w-l} \rfloor$ for a random w -bit odd integer a

- ▶ for details see [[M.Thorup, High Speed Hashing for Integers and Strings, arxiv:1504.06804, May 2019](#)]

Remarks (cont)


- For strings, the following Karp-Rabin hash functions provides good guarantees:

for $s = s[0..n - 1]$,

$$h_x(s) = (s[0] \cdot x^{n-1} + s[1] \cdot x^{n-2} + \dots + s[n - 1]) \bmod p$$

where p is a *fixed* large prime number and x is a *random* number uniformly drawn from $[1..p - 1]$.

In practice p is often set to $2^{31} - 1$ for 32-bit numbers and to $2^{61} - 1$ for 64-bit numbers (Mersenne primes)



Perfect hashing



Motivation

- ▶ Can we guarantee a **worst-case** $O(1)$ time for hash table operations?

Should Tables Be Sorted?

ANDREW CHI-CHIH YAO

Stanford University, Stanford, California

ABSTRACT Optimality questions are examined in the following information retrieval problem. Given a set S of n keys, store them so that queries of the form, “Is $x \in S$?” can be answered quickly. It is shown that in a rather general model including all the commonly used schemes, $\lceil \lg(n + 1) \rceil$ probes to the table are needed in the worst case, provided the key space is sufficiently large. The effects of smaller key space and arbitrary encoding are also explored.

KEY WORDS AND PHRASES information retrieval, lower bound, optimality, query, Ramsey’s theorem, search strategy, sorted table

CR CATEGORIES. 3.74, 4.34, 5.25, 5.31

1. Introduction

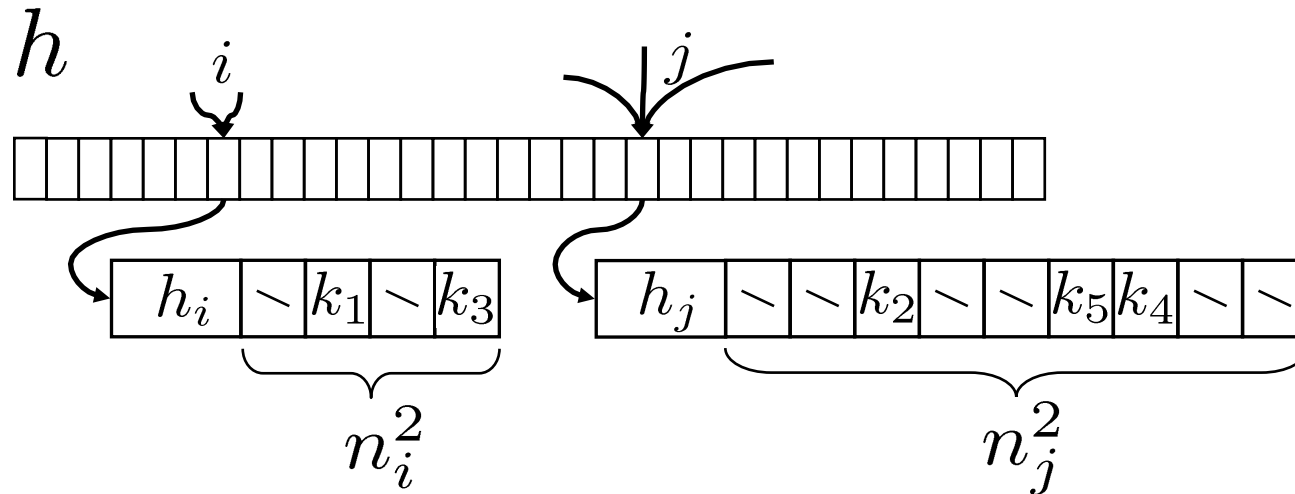
Given a set S of n distinct keys from a key space $M = \{1, 2, \dots, m\}$, a basic information retrieval problem is to store S so that *membership* queries of the form, “Is j in S ?” can be answered quickly. Two commonly used schemes are the sorted table and the hash table. In the first case, a query can be answered in $\lceil \lg(n + 1) \rceil$ probes by means of a binary search.¹ The hash table scheme has a good average-case cost, but requires $O(n)$ probes in the worst case for typical hashing schemes. Looking at various alternative methods, one gets the feeling that $\sim \log n$ probes must be necessary in the worst case, if the key space M is large and we only use about

Motivation

- ▶ Can we guarantee a **worst-case** $O(1)$ time for hash table operations?
- ▶ **Yes** (if the set of keys is static)
- ▶ However, the construction time is **expected** $O(n)$ (Las Vegas algorithm)

Perfect hashing


- ▶ Fredman, Komlós, Szemerédi (1984)
- ▶ Guarantees $O(1)$ **worst-case** time of LOOKUP for a **static** set of keys. Solution uses universal hashing.
- ▶ 2-level hash scheme:



- ▶ LOOKUP: **worst-case** $O(1)$
- ▶ practical implementations exist (e.g. `gperf` in C++)

Why $\sum n_i^2$ can be made $O(n)$

- ▶ $\sum n_i^2 = n + [\text{nb of pairs which collide}]$
- ▶ $E[\text{nb of pairs which collide}] \leq n(n-1) \cdot \frac{1}{n} = n-1$

$P[\text{collision}]$ 
- ▶ $\Rightarrow E[\sum n_i^2] < 2n \Rightarrow P[\sum n_i^2 > 4n] < 1/2$

\uparrow
by Markov inequality
- ▶ **Algorithm (sketch)**
 - ▶ hash to primary table of size $O(n)$ using a universal h.f. h
 - ▶ hash each non-empty bucket to a table of size n_i^2 ; if $\sum n_i^2 > 4n$, rehash
 - ▶ using a universal h.f. h_i ; if collision, rehash until there is none (expected $O(1)$ time by birthday paradox)

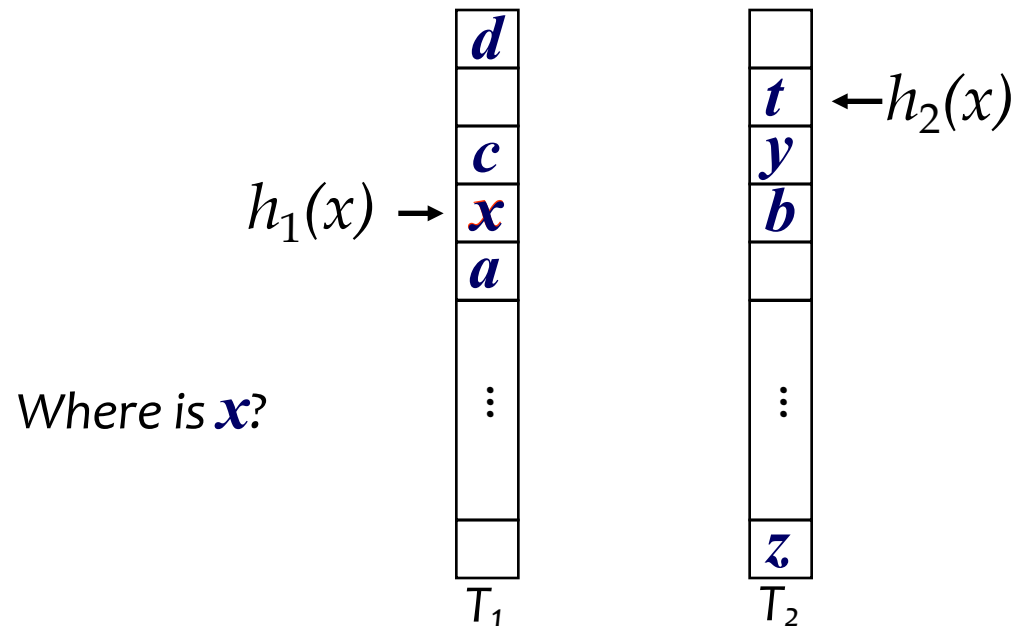


Cuckoo hashing

(perfect hashing with open addressing)

Cuckoo hashing

- ▶ Introduced by Pagh&Rodler in 2001
- ▶ uses two tables T_1 and T_2 with hash functions h_1 and h_2 respectively
- ▶ LOOKUP(k): check $h_1(k)$ and $h_2(k)$



Cuckoo hashing

- ▶ Introduced by Pagh&Rodler in 2001
- ▶ uses two tables T_1 and T_2 with hash functions h_1 and h_2 respectively
- ▶ LOOKUP(k): check $h_1(k)$ and $h_2(k)$
- ▶ INSERT(k):

check $h_1(k)$ in T_1 ;

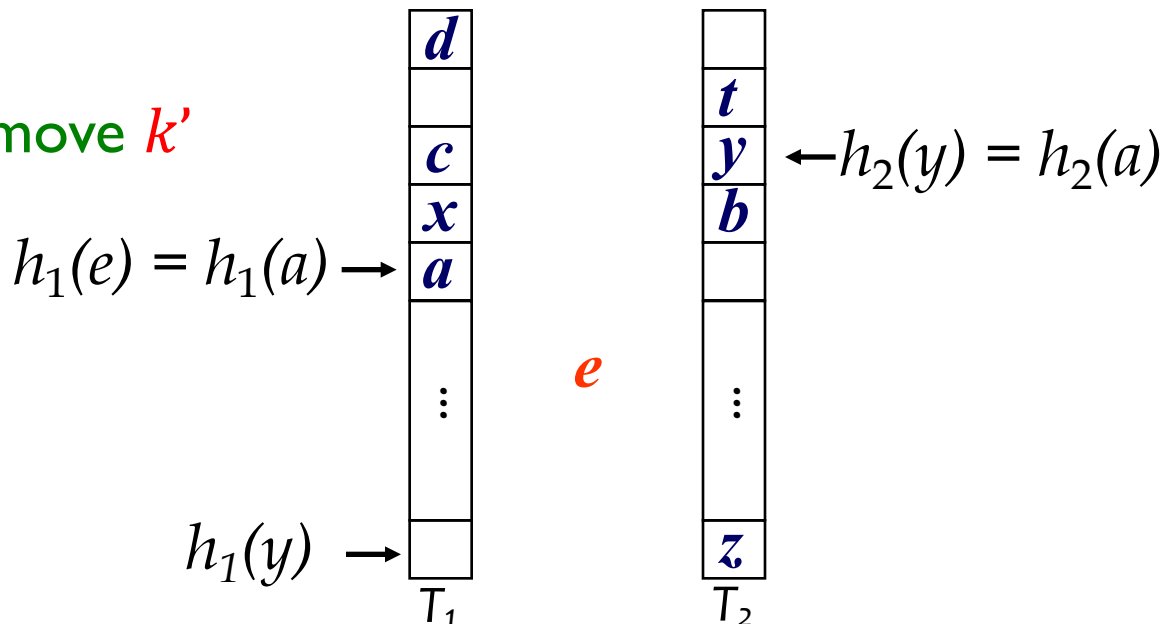
if occupied by some k' , move k'

out to $h_2(k')$ in T_2 ;

iterate until finding

a free bucket;

if it loops, rehash



Cuckoo hashing: summary

- ▶ LOOKUP, DELETE: worst-case $O(1)$ (two probes)
- ▶ INSERT: expected $O(1)$
- ▶ inserting $O(n)$ items triggers $O(1)$ rehashes, each rehash takes $O(n)$ time \Rightarrow rehash takes amortized $O(1)$ time
- ▶ no dynamic memory allocation (as in chaining)
- ▶ reasonable memory use

Don't Throw Out Your Algorithms Book Just Yet: Classical Data Structures That Can Outperform Learned Indexes

by Peter Bailis, Kai Sheng Tai, Pratiksha Thaker, and Matei Zaharia

11 Jan 2018

There's recently been a lot of excitement about a **new proposal** from authors at Google: to replace conventional indexing data structures like B-trees and hash maps by instead fitting a neural network to the dataset. The paper compares such learned indexes against several standard data structures and reports promising results. For range searches, the authors report up 3.2x speedups over B-trees while using 9x less memory, and for point lookups, the authors report up to 80% reduction of hash table memory overhead while maintaining a similar query time.

While learned indexes are an exciting idea for many reasons (e.g., they could enable self-tuning databases), there is a long literature of other optimized data structures to consider, so naturally researchers have been trying to see whether these can do better. For example, Thomas Neumann posted about **using spline interpolation in a B-tree for range search** and showed that this easy-to-implement strategy can be competitive with learned indexes. In this post, we examine a second use case in the paper: memory-efficient hash tables. We show that for this problem, a simple and beautiful data structure, the **cuckoo hash**, can achieve 5-20x less space overhead than learned indexes, and that it can be surprisingly fast on modern hardware, running nearly 2x faster. These results are interesting because the cuckoo hash is *asymptotically* better than simpler hash tables at load balancing, and thus makes optimizing the hash function using machine learning less important: it's always great to see cases where beautiful theory produces great results in practice.

Going Cuckoo for Fast Hash Tables

Let's start by understanding the hashing use case in the learned indexes paper. A typical hash function distributes keys randomly across the slots in a hash table, causing some slots to be empty, while others have collisions, which require some form of **chaining** of items. If lots of memory is available, this is not a problem: simply create many more slots than there are keys in the table (say, 2x) and collisions will be rare. However, if memory is scarce, heavily loaded tables will result in slower lookups due to more chaining. The authors show that, by learning a hash function that *spreads the input keys more evenly* throughout



Bloom filters

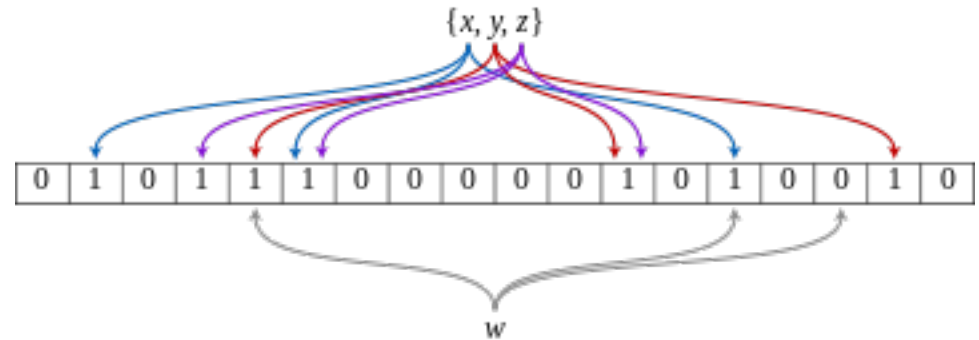


Bloom filters: generalities

- ▶ Bloom (1970)
- ▶ generalizes the bitmap representation of sets
- ▶ supports INSERT and LOOKUP
- ▶ LOOKUP only checks for the presence, no satellite data
- ▶ produces false positives (with low probability)
- ▶ cannot iterate over the elements of the set
- ▶ DELETE is not supported (in the basic variant)
- ▶ **very** space efficient
- ▶ *Example*: forbidden passwords

Bloom filter: how it works

- ▶ U : universe of possible elements
- ▶ K : subset of elements, $|K| = n$
- ▶ m : size of allocated **bit array**
- ▶ define d hash functions $h_1, \dots, h_d: U \rightarrow \{0, \dots, m-1\}$
- ▶ INSERT(k): set $h_i(k) = 1$ for all i
- ▶ LOOKUP(k): check $h_i(k) == 1$ for all i
- ▶ false positives but no false negatives



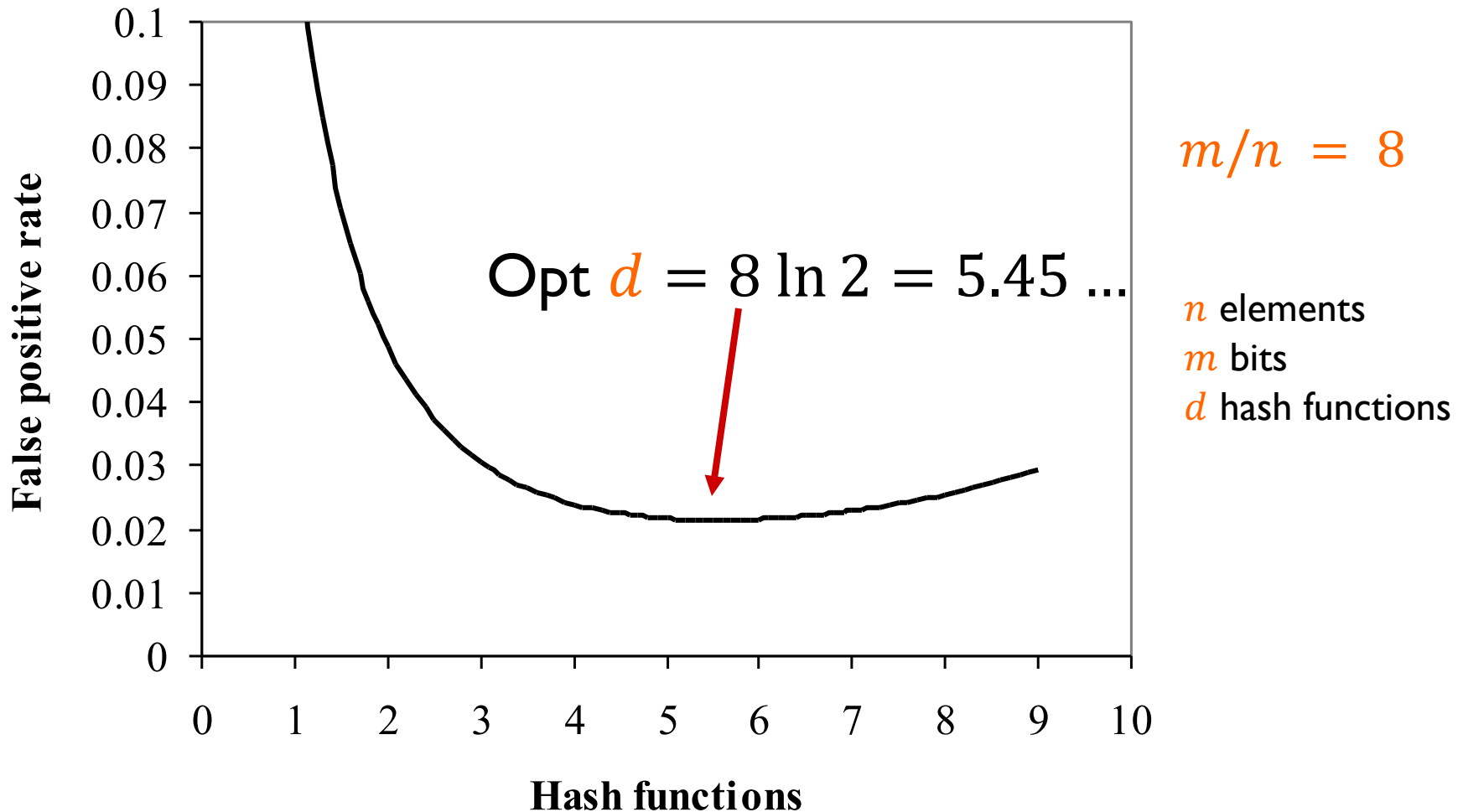
Bloom filters: analysis

- ▶ $P[\text{specific bit of filter is 0}] = (1 - 1/m)^{dn} \approx e^{-dn/m} \equiv p$
- ▶ $P[\text{false positive}] = (1 - p)^d = (1 - e^{-dn/m})^d$
- ▶ Optimal number d of hash functions: $d = \ln 2 \cdot \frac{m}{n}$
- ▶ Therefore, for the optimal number of hash functions,

$$P[\text{false positive}] = 2^{-\ln 2 \cdot \frac{m}{n}} \approx 0.6185^{\frac{m}{n}}$$

- ▶ E.g. with 10 bits per element, $P[\text{false positive}]$ is less than 1%
- ▶ To insure the FP rate ε : $m \approx 1.44 \cdot n \cdot \log_2 \frac{1}{\varepsilon}$

Dependence on the nb of hash functs



Bloom filter: properties

- ▶ For the optimal number of hash function, about a half of the bits is 1 [show]
- ▶ The Bloom filter for the union is the OR of the Bloom filters
- ▶ Is similar true for the intersection? [explain]
- ▶ If a Bloom filter is sparse, it is easy to halve its size
- ▶ Various generalizations exist, e.g. *counting Bloom filters* (support counting and deletion)
- ▶ Cuckoo filter: a combination of Bloom filters and Cuckoo hashing, supports deletion

Bloom filters: applications

- ▶ Bloom filters are very easy to implement
- ▶ Used e.g. for
 - ▶ spell-checkers (in early UNIX-systems)
 - ▶ unsuitable passwords
 - ▶ "approximate" unsuitable passwords (Manber&Wu 1994)
 - ▶ malicious sites in Google Chrome
 - ▶ read articles in publishing systems (Medium)
 - ▶ Google Bigtable, Apache HBase, Bitcoin, bioinformatics, ...
- ▶ Sometimes it is possible to store the set of false positives in a separate data structure