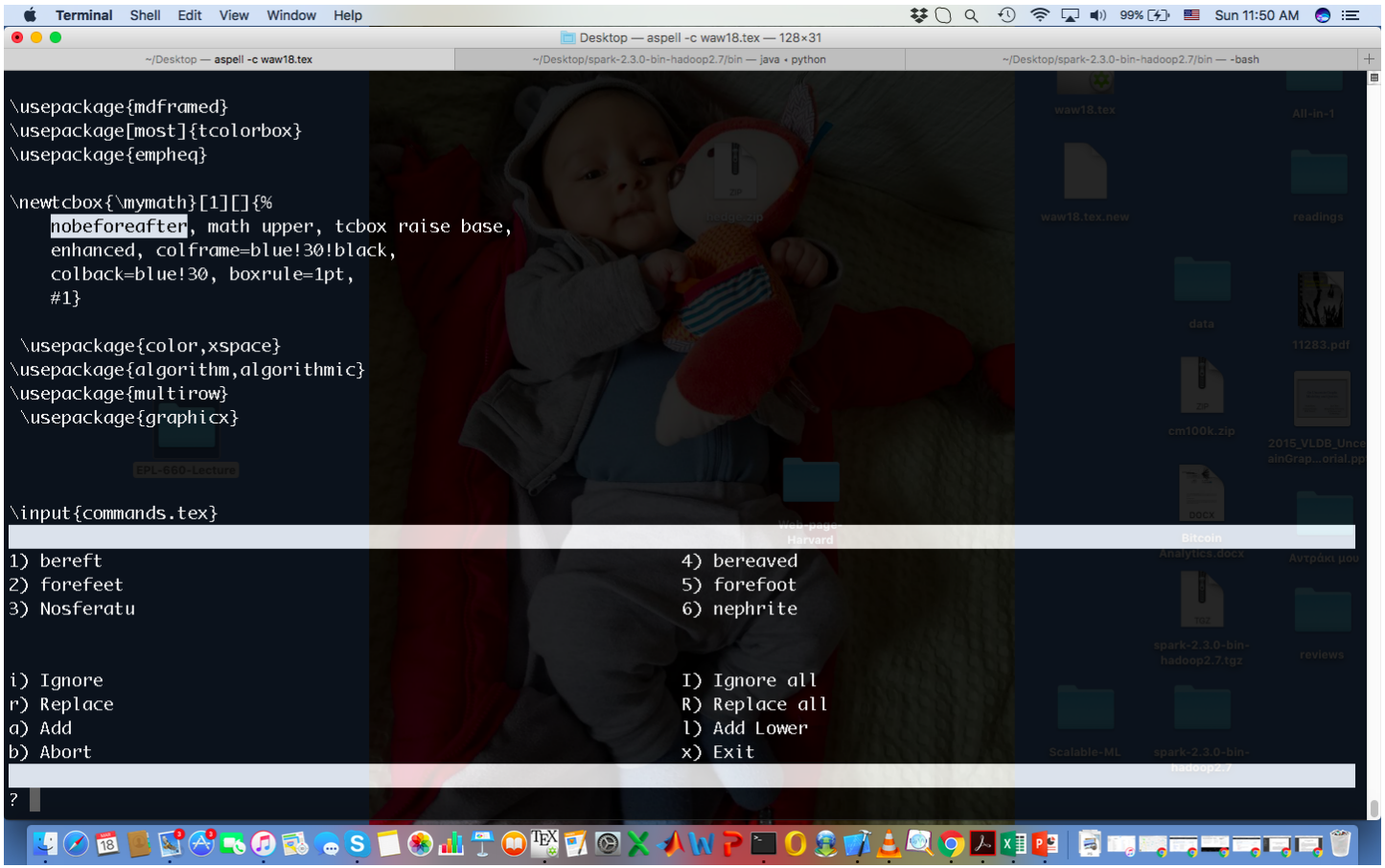
EPL 660, Guest Lecture
Nuts and Bolts of a Spell Checker
UCY

March 19th, 2018

What is a spell checker?

Definition: A spell checker is an application program that flags words in a document that may not be spelled correctly.



(Some) Nuts & Bolts of a Spell Checker

Today we will show how to create a spell checker on a laptop. On my Mac, I use /usr/share/dict/words as our lexicon.

Basics of Hashing

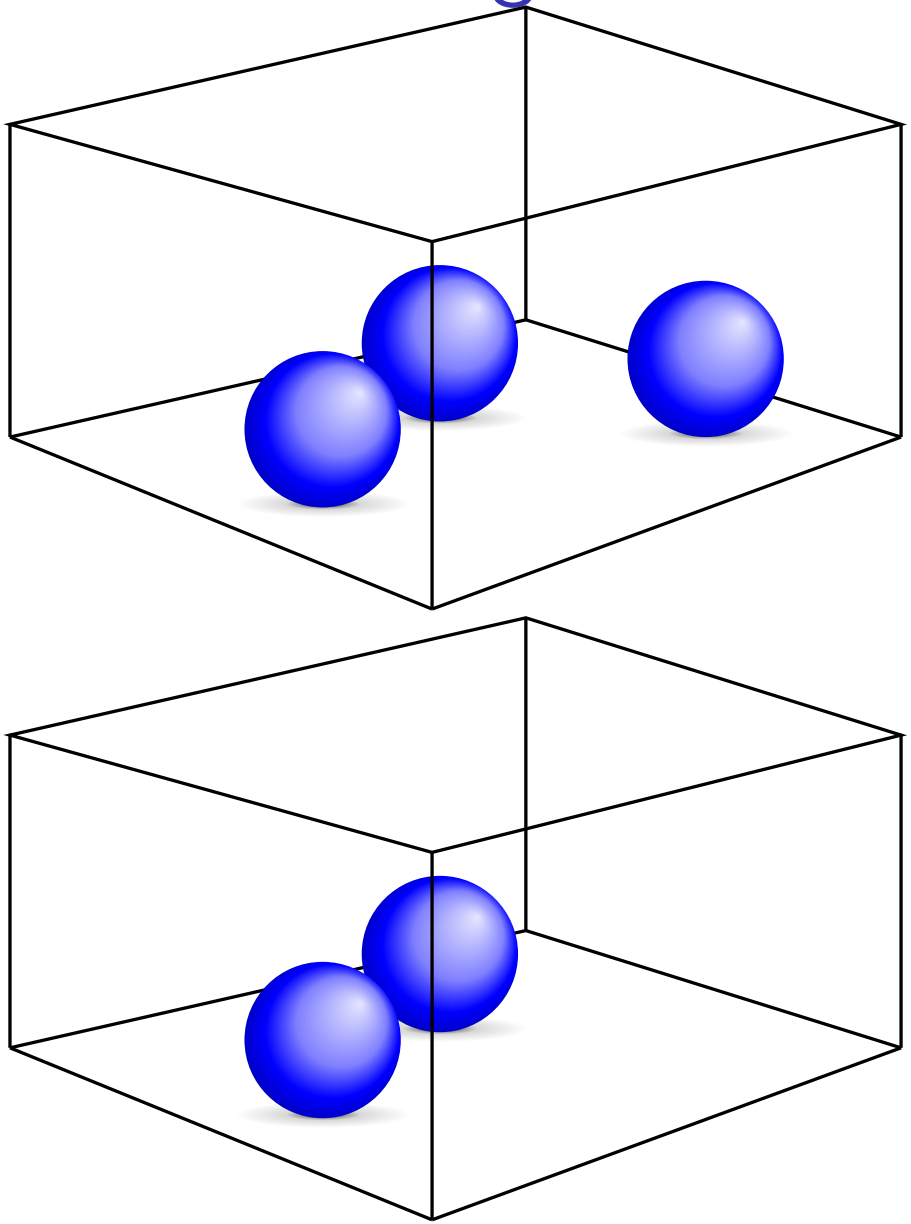
- Check quickly if a word in our document appears or not in the lexicon.

Bloom filters

- What if space is a constraint? How to create a space efficient spell checker.

TIME FOR HASHING
imgflip.com

Balls and bins again : n balls, n bins



Problem: Maximum bin load?

n balls into n bins

Two ways to prove this claim.

1. Chernoff and union bound
2. Binomials and union bound

$$\mathbf{Pr}\left[\exists i : X_i \geq \underbrace{\frac{3\log n}{\log\log n}}_{k}\right] \leq n\binom{n}{k}\frac{1}{n^k} \leq \frac{1}{n}.$$

Reminder, Chernoff bound: Let $X_1, \ldots, X_n$ be independent RVs with $X_i \in \{0,1\}$, $X = \sum_{i=1}^{n} X_i$, then:

$$\mathbf{Pr}\left[X \geq (1+\delta)\mathbb{E}\left[X\right]\right] \leq \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mathbb{E}[X]}$$

n balls into n bins

$$\mathbf{Pr}\left[\exists \text{bin with more than } \frac{3\log n}{\log\log n} \text{ balls}\right] \leq \frac{1}{n}$$

Changing the maximum load

$$c\frac{\log n}{\log\log n}$$

by playing with constant c, we can decrease the failure probability as $\frac{1}{\text{poly}(n)}$.

Dictionary problem

Universe $U = [u] = \{0, \ldots, u-1\}$

Set $S \subseteq U$, $|S| = n$, $|S| \ll U$

Goal: design a data structure that supports efficiently the following operations.

- $\textsc{Make}()$: Initializes an empty dictionary
- $\textsc{Insert}(x)$: Add element x in S
- $\textsc{Lookup}(x)$: Does x appear in S
- $\textsc{Delete}(x)$: Removes x from S, if present

Questions:

- Why not a linked list?
- Why not an array over U?

Python dictionary

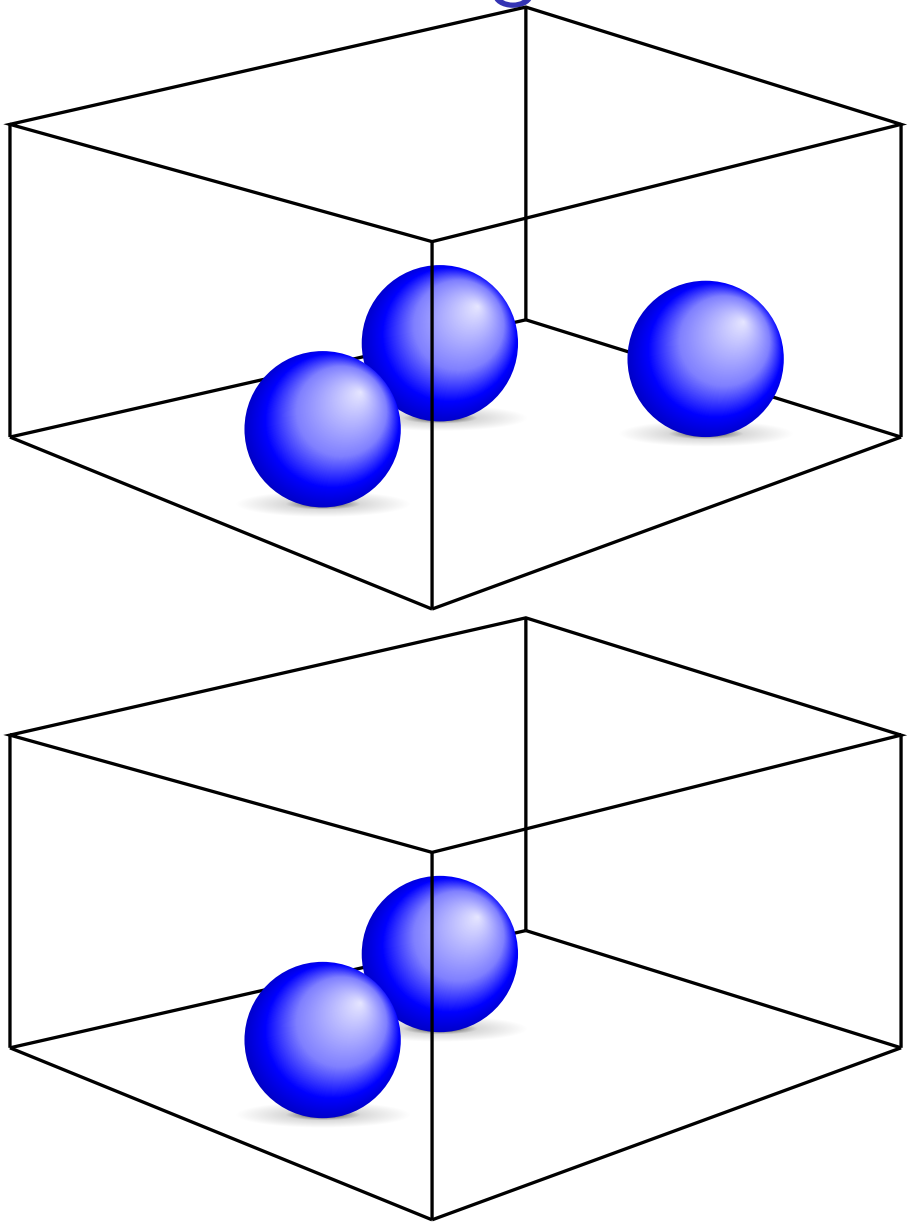
```
#empty table
d ={}

#insert
d["Andrei␣Rublev"] = "Tarkovsky"
d["Stalker"] = "Tarkovsky"
d["Viridiana"] ="Bunuel"
d[('123','a')] = "a123"

#lookup
print(d["Stalker"])
print(d[('123','a')])

#delete
del d["Stalker"]
print(d["Stalker"]) #KeyError: 'Stalker'
```

Hashing

- **Basic idea:** Work with an array of size $m = O(|S|)$ rather than of size $O(|U|)$!
- **Hash function:** $h : [u] \to [m]$
- **Hash table:** Array. We place $x \in S$ at position $h(x)$.
- **Collision:** $x \neq y \in U$ get mapped to $h(x) = h(y)$.

1. How do we choose h?
2. How do we resolve conflicts?

Balls and bins again : n balls, r bins



Problem: Collision?

Balls and Bins Revisited: k-wise independence

Consider the load of some bin.

$$\sum_{K \subseteq S, |S|=k} \frac{1}{r^k} \leq \left(\frac{en}{k}\right)^k r^{-k} = \left(\frac{en}{rk}\right)^k$$

- If $k > 2en/r > 2 \log r$ the probability of k balls in any single bucket is $< 1/r$.
- No need for full randomness, but randomness over all subsets of k hash values.

Source: See also Rasmus Pagh's slides

Balls and Bins Revisited: k-wise independence

Definition: RVs $X_1, \ldots, X_n$ are k-wise independent iff for any set of indices $i_1, \ldots, i_k$, RVs $X_{i_1}, \ldots, X_{i_k}$ are independent.

Definition: A set of hash function $\mathcal{H}$ is a k-wise independent family iff the random variables $h(0), \ldots, h(u-1)$ are k-wise independent when $h \in \mathcal{H}$ is drawn uniformly at random.

Example 1: The set $\mathcal{H}$ of all functions from $[u]$ to $[m]$ is k-wise independent for all k.

 # Bits: $u \log m$ (u is enormous!)

2-wise independent family

Exercise: We can construct a 2-wise independent family as follows.

- p is prime
- a, b chosen uar from $[p]$
- The hash of x is

$$h(x) = ax + b \mod p,$$

How many bits do we need now?

Generalization: Polynomials with random coefficients, see `https://en.wikipedia.org/wiki/K-independent_hashing/Polynomials_with_random_coefficients`

Universal hash family

- A family $\mathcal{H}$ of hash functions is **strongly 2-universal** if for any $x_1 \neq x_2$,

$$\mathbf{Pr}\left[h(x_1) = y_1, h(x_2) = y_2\right] = \frac{1}{m^2}.$$

for a uniform $h \in \mathcal{H}$.

What is the connection with the previous slide?

Avoiding Modular Arithmetic

- Modular arithmetic can be slow
- [Dietzfelbinger et al., 1997] proposed the following hash function (collisions twice as likely):
- For each k, l they define a class $\mathcal{H}_{k,l}$ of hash functions from $U = [2^k]$ to $M = [2^l]$

$$\mathcal{H}_{k,l} = \{h_\alpha | h_\alpha = (ax \bmod 2^k) \operatorname{div} 2^{k-l}.$$

- **Claim:** If α is a random odd $0 < \alpha < 2^l$, and $x_1 \neq x_2$, then

$$\mathbf{Pr}\left[h(x) = h(y)\right] \leq 2^{-l+1}.$$

RANDOM
HASH FUNCTION
memegenerator.net

String hashing: bad choice, why?

```
unsigned long hash(unsigned char *str)
{
        unsigned int hash = 0;
        int c;

        while (c = *str++)
            hash += c;
        return hash;
}
```

String hashing: djbx33a

```
unsigned long hash(unsigned char *str)
{
    unsigned long hash = 5381;
    int c;
    while (c == *str++)
            hash = ((hash << 5) + hash) + c;
    return hash;
}
```

SO YOU ARE TELLING ME
THAT DJBX33A WORKS FINE?
memegenerator.net

djbx33a is Vulnerable to attacks

```
#include <iostream>
#include <cstring>

// author: Charalampos Tsourakakis

unsigned long hash(std::string str){
    unsigned long hash = 5381;
    int c;

    for( int i = 0; i < str.length(); i++)
        hash = ((hash << 5) + hash) + str.at(i)
    return hash;
}
```

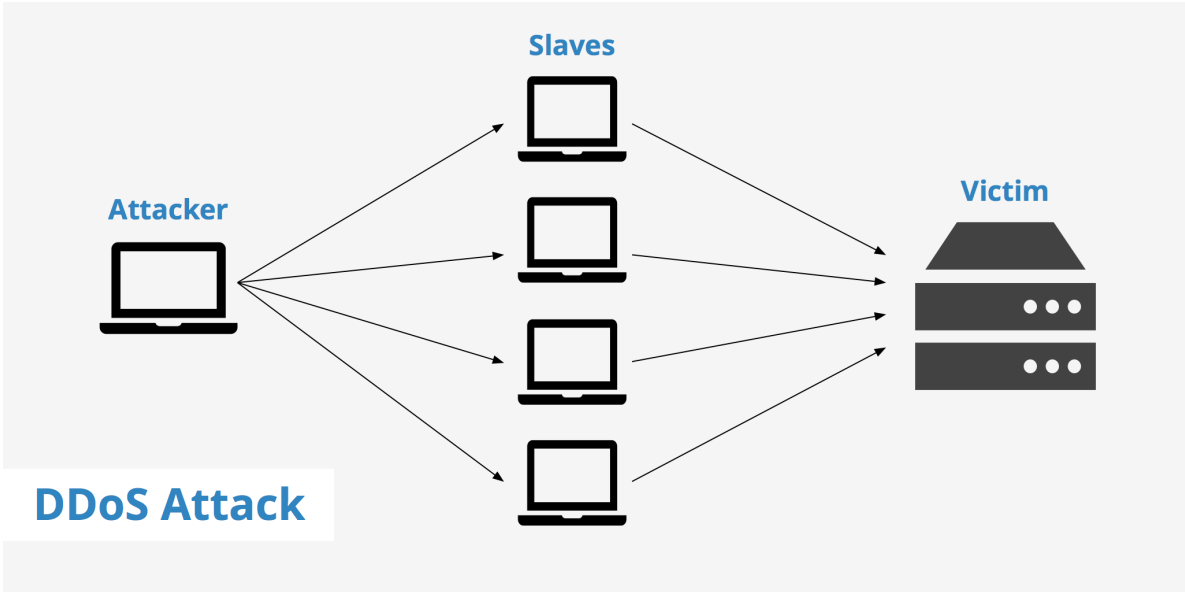
djbx33a is Vulnerable to attacks

```
int main()
{
std::string s="Ey";
std::cout<<"h(Ey)="<<hash(s)<<std::endl;
s = "FZ";
std::cout<<"h(FZ)="<<hash(s)<<std::endl;
return 0;
}
```

```
>> g++ -o DoSdjbx33a DoSdjbx33a.cc
>> ./DoSdjbx33a
h(Ey)=5862307
h(FZ)=5862309
```

Verify (exercise) that $h(Ey) = h(FZ)$ for djbx33a hash function.

Hash-flooding DoS



Definition: Send to a server many inputs with a same hash (enforces linear)

String hashing: java.lang.String.hashCode()

```
unsigned long hash(unsigned char *str)
{
    unsigned long hash = 0;
    int c;
    while (c == *str++)
            hash = ((hash << 5) - hash) + c;
    return hash;
}
```

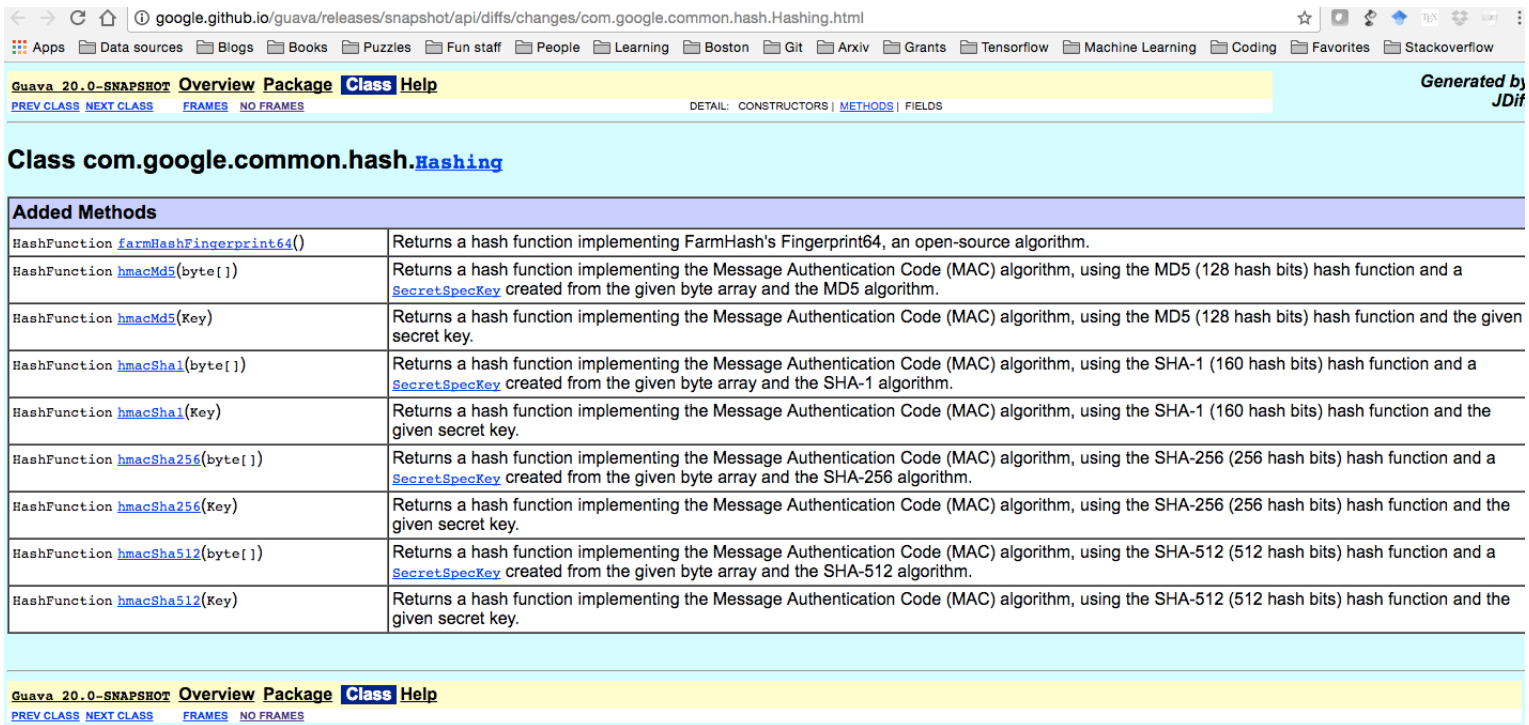
Hash-flooding DoS

Here is what your website may look like after a successful Denial of Service Attack:

Figure from: How to Detect a Denial of Service (DoS) Attack

Hash-flooding DoS

For example: FARMHASH::FINGERPRINT64() takes as input a *string*, and outputs a *uint64*. [Not secure!]



google.github.io/guava/releases/snapshot/api/diffs/changes/com.google.common.hash.Hashing.html

Apps | Data sources | Blogs | Books | Puzzles | Fun staff | People | Learning | Boston | Git | Arxiv | Grants | Tensorflow | Machine Learning | Coding | Favorites | Stackoverflow

Guava 20.0-SNAPSHOT Overview Package Class Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES DETAIL: CONSTRUCTORS | METHODS | FIELDS

Generated by JDif

Class com.google.common.hash.Hashing

Added Methods	
HashFunction farmHashFingerprint64()	Returns a hash function implementing FarmHash's Fingerprint64, an open-source algorithm.
HashFunction hmacMd5(byte[])	Returns a hash function implementing the Message Authentication Code (MAC) algorithm, using the MD5 (128 hash bits) hash function and a SecretSpecKey created from the given byte array and the MD5 algorithm.
HashFunction hmacMd5(Key)	Returns a hash function implementing the Message Authentication Code (MAC) algorithm, using the MD5 (128 hash bits) hash function and the given secret key.
HashFunction hmacSha1(byte[])	Returns a hash function implementing the Message Authentication Code (MAC) algorithm, using the SHA-1 (160 hash bits) hash function and a SecretSpecKey created from the given byte array and the SHA-1 algorithm.
HashFunction hmacSha1(Key)	Returns a hash function implementing the Message Authentication Code (MAC) algorithm, using the SHA-1 (160 hash bits) hash function and the given secret key.
HashFunction hmacSha256(byte[])	Returns a hash function implementing the Message Authentication Code (MAC) algorithm, using the SHA-256 (256 hash bits) hash function and a SecretSpecKey created from the given byte array and the SHA-256 algorithm.
HashFunction hmacSha256(Key)	Returns a hash function implementing the Message Authentication Code (MAC) algorithm, using the SHA-256 (256 hash bits) hash function and the given secret key.
HashFunction hmacSha512(byte[])	Returns a hash function implementing the Message Authentication Code (MAC) algorithm, using the SHA-512 (512 hash bits) hash function and a SecretSpecKey created from the given byte array and the SHA-512 algorithm.
HashFunction hmacSha512(Key)	Returns a hash function implementing the Message Authentication Code (MAC) algorithm, using the SHA-512 (512 hash bits) hash function and the given secret key.

Guava 20.0-SNAPSHOT Overview Package Class Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES

For secure cryptographic functions, a good start is the MD5 algorithm.

Perfect Hashing

- So far, we've seen that the average case behavior of hashing is significantly superior to the worst case.
- However, we can get excellent worst case performance if the set of keys is static.
- Perfect hashing requires $O(1)$ memory accesses in the worst case.

Theorem: If $\mathcal{H}$ is 2-universal, $|S| = n$, $m \geq \alpha\binom{n}{2}$, then

$$\mathbf{Pr}\left[\text{h is perfect for } S\right] \geq 1 - \frac{1}{\alpha}.$$

Perfect Hashing

Proof sketch:

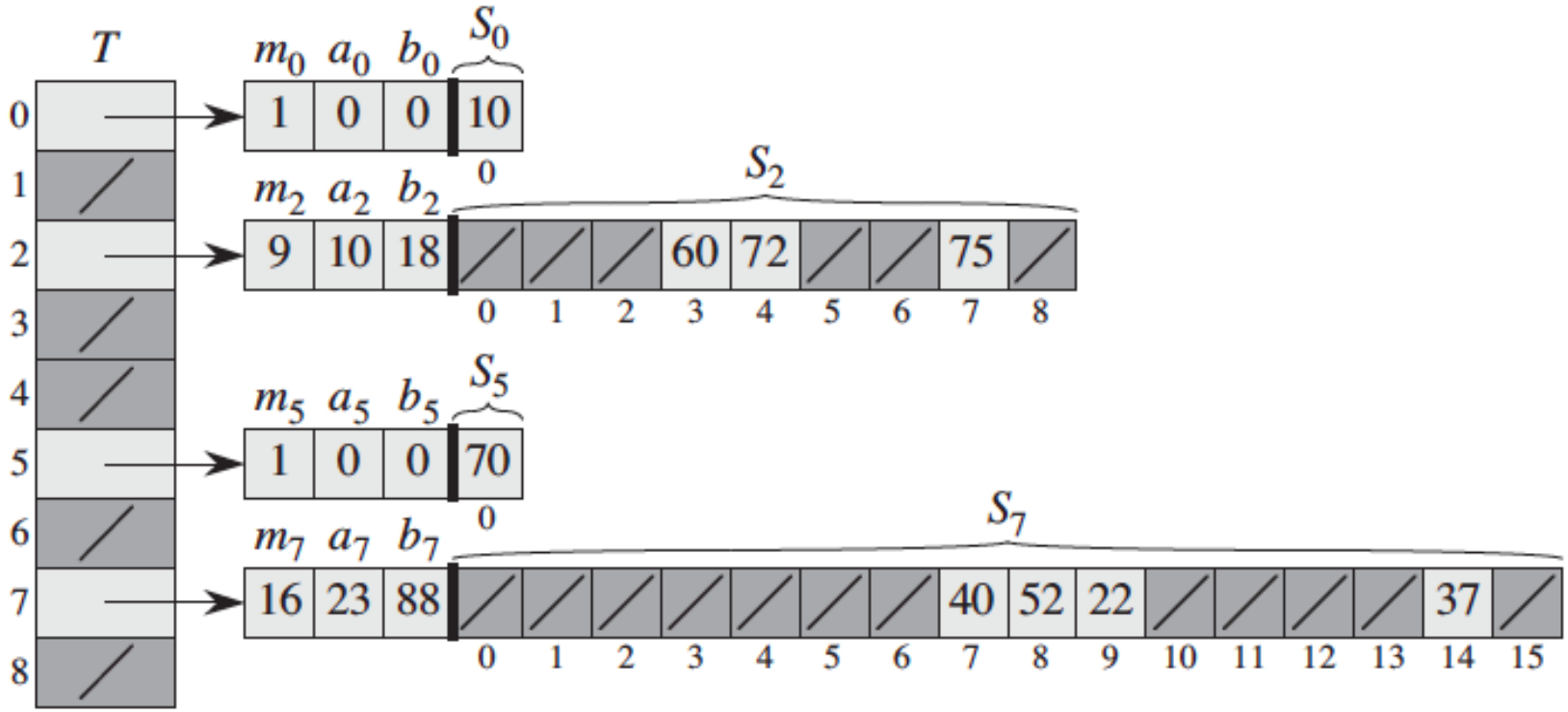
- Define $X = \#$ collisions, and let's compute $\mathbb{E}[X]$
- $X = \sum_{i \neq j} X_{ij}$
- $\mathbf{Pr}[X_{ij} = 1] = \frac{1}{m}$
- By linearity of expectation $\mathbb{E}[X] = \frac{\binom{n}{2}}{m} \leq \frac{1}{\alpha}$
- Apply Markov's inequality

$$1 - \mathbf{Pr}[X = 0] = \mathbf{Pr}[X \geq 1] \leq \mathbb{E}[X] \leq \frac{1}{\alpha}.$$

Perfect Hashing

- Issue: $O(n^2)$ space
- Question: Can we get away with $O(n)$ space?
- Yes: Fredman-Komlós-Szemerédi
- Idea: Two level hashing,
  1. Hash using a universal hash function to $n = |S|$ bins.
  2. Rehash perfectly within each bin at second level.

Perfect Hashing

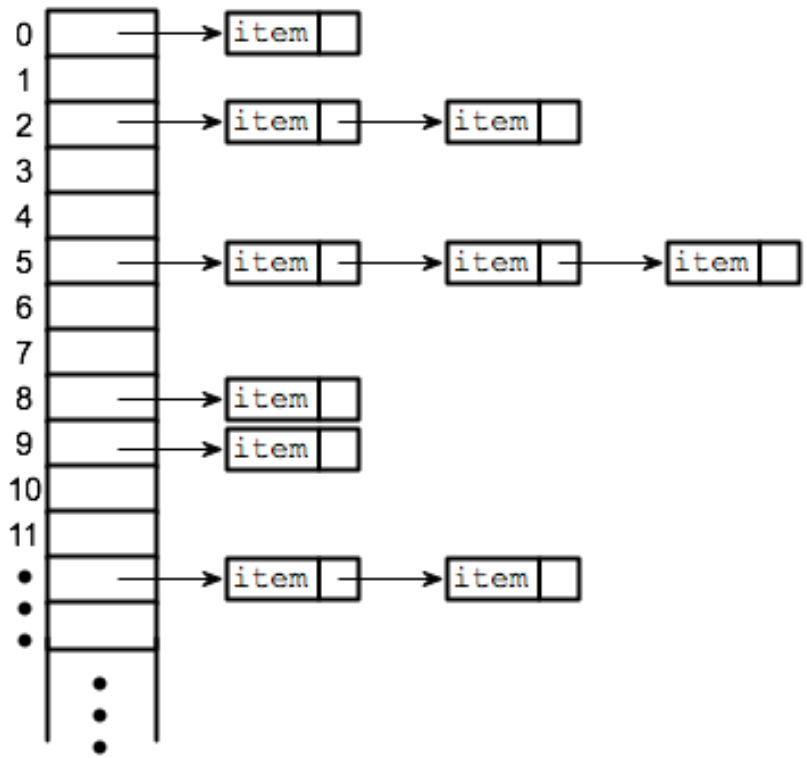


Source: CLRS book

Claim (Exercise):

$$\mathbb{E}\left[\sum_{j=0}^{n-1} n_j^2\right] \leq 2n.$$

Separate Chaining



Source: Hackerearth

Insertion

```
vector <string> Table[20];
int hashTableSize=20;

void insert(string s)
{
// Compute the index using Hash Function
int index = hashFunc(s);
// Insert the element
Table[index].push_back(s);
}
```

Search

```
 void search(string s)
{
  int index = hashFunc(s);
  for(int i = 0;i < Table[index].size();i++)
  {
      if(Table[index][i] == s)
      {
          cout << s << " is found!" << endl;
          return;
      }
  }
  cout << s << " is not found!" << endl;
}
```

Separate Chaining

Load factor α:

$$\alpha := \frac{n}{m}.$$

Claim: Under the assumption of simple uniform hashing, an unsuccessful search takes $O(1 + \alpha)$ time.

Proof sketch: $\mathbb{E}[n_j] = \alpha$ for all $j \in \{0, \ldots, m-1\}$.

Linear Probing

- Sequential memory accesses are fast
- Values stored directly to hash table
- We hash x to $h(x)$. If this cell is already occupied, then we check $h(x) + 1, h(x) + 1$ and so on (mod arithmetic).
- Pagh et al. proved that if hash function is 5-wise independent, then $\mathbb{E}[\text{operation}] = O(1)$.

Insertion

```
// Linear probing
void insert(string s)
{
   int index = hashFunc(s);
   while(Table[index] != "")
      index = (index + 1) % hashTableSize;
   hashTable[index] = s;
}
```

Search

```
void search(string s)
{
 int index = hashFunc(s);
 while(Table[index] != s&&Table[index] != "")
        index = (index+1)%hashTableSize;
 if(Table[index] == s)
      cout << s << "␣is␣found!" << endl;
 else
       cout << s << "␣is␣not␣found!" << endl;
}
```

Quadratic Probing

- Difference from linear probing is the choice between successive probes or entry slots

  $\text{index} = \text{index} \ \% \ \text{hashTableSize}$

  $\text{index} = (\text{index} + 1^2) \ \% \ \text{hashTableSize}$

  $\text{index} = (\text{index} + 2^2) \ \% \ \text{hashTableSize}$

  $\text{index} = (\text{index} + 3^2) \ \% \ \text{hashTableSize}$

Insertion

```
void insert(string s)
{
   int index = hashFunc(s);
   int h = 1;
   while(hashTable[index] != "") {
       index = (index + h*h) % hashTableSize;
       h++;}
   Table[index] = s;
}
```

Search

```
void search(string s)
{
   int ind = hashFunc(s);
   int h = 1;
   while(Table[ind] != s&&Table[ind] != ""){
       ind = (ind + h*h) % hashTableSize;
       h++;}
   if(Table[index] == s)
     cout << s << " is found!" << endl;
    else
     cout << s << " is not found!" << endl;
}
```

Double Hashing

- Difference from linear probing is that the interval between probes is computed by using two hash functions.

  indexH = hashFunc2(s);

  index = (index + 1 * indexH) % hashTableSize;

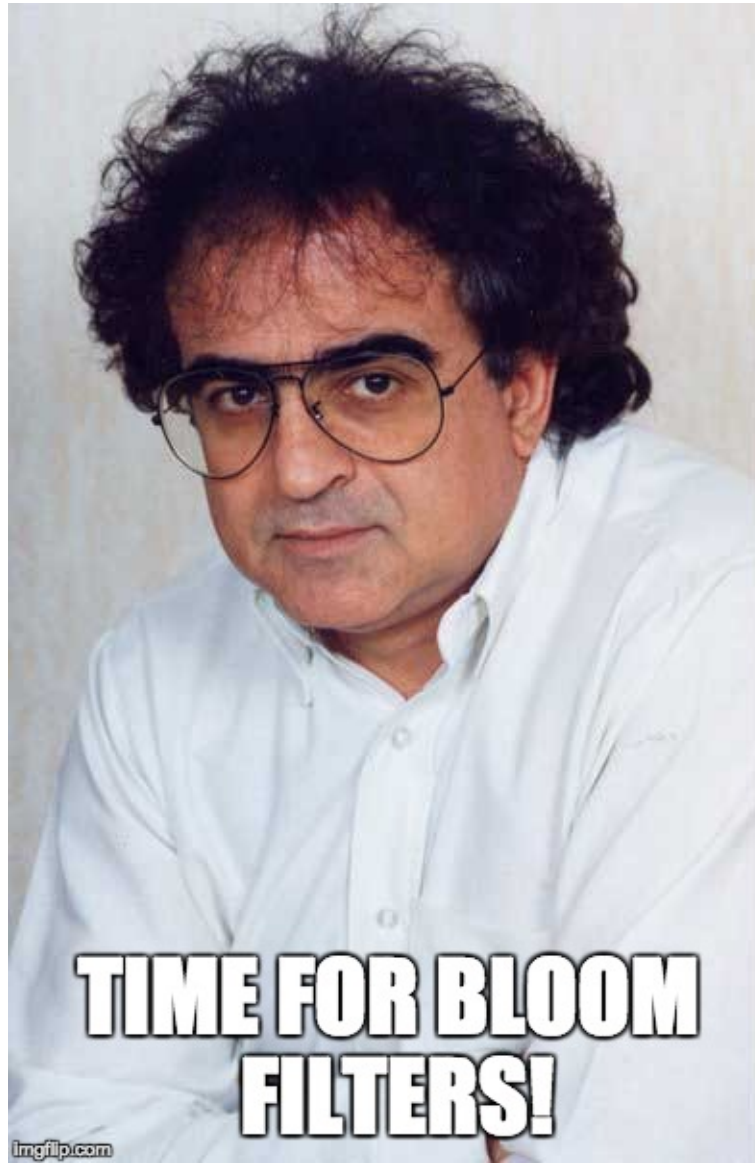
  index = (index + 2 * indexH) % hashTableSize;

Insertion

```
void insert(string s)
{
 int index = hashFunc1(s);
 int indexH = hashFunc2(s);
 while(hashTable[index] != "")
      index = (index+indexH)%hashTableSize;
  hashTable[index] = s;
}
```

Search

```
void search(string s)
{
  int index = hashFunc1(s);
  int indexH = hashFunc2(s);
  while(Table[index]!= s&&Table[index]!= "")
    index = (index + indexH)%hashTableSize;
  if(Table[index] == s)
    cout << s << "␣is␣found!" << endl;
   else
    cout << s << "␣is␣not␣found!" << endl;
}
```


TIME FOR BLOOM
FILTERS!
imgflip.com

Bloom Filter

- Approximate membership problem
- Highly space-efficient randomized data structure
- Its analysis shows an interesting tradeoff between space and error probability
- **The Bloom filter principle** :

  Wherever a list or set is used, and space is at a premium,consider using a Bloom filter if the effect of false positives can be mitigated.

Bloom Filter – Applications

Historically, Bloom filter was developed in the context of **dictionary applications** when space resources were scarce.

- Burton H. Bloom introduced Bloom filters (1970) for an application related to hyphenation programs.
- Bloom filters were also used in early UNIX spell-checker (space savings were crucial for functionality)
- Avoid weak passwords.
- Content Delivery in P2P networks
- Networks
- Distributed Caching
- Databases (e.g., Bloomjoin algorithm)
- ...

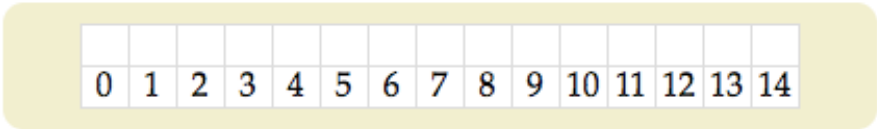
Bloom Filter – Description

1. A vector of m bits
2. k independent hash functions $h_1, \ldots, h_k$
3. A set S of n keys
4. To store key x, we set $A[h_i(x)] = 1$ for all $i \in [k]$
5. Lookup(x): if $A[h_i(x)] = 1$ for all $i \in [k]$, then $x \in S$.
6. No false negatives, but false positives may exist.

Bloom filter in Python – Pybloom library

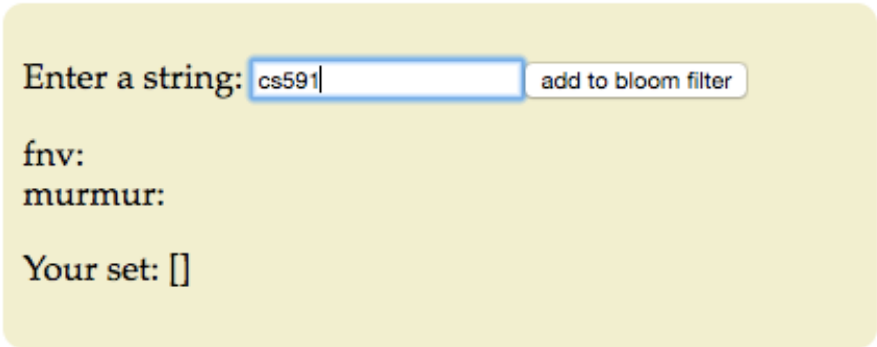
```
>>> from pybloom import BloomFilter
>>> f = BloomFilter(capacity=1000, err=0.001)
>>> [f.add(x) for x in range(10)]
>>> all([(x in f) for x in range(10)])
True
>>> 10 in f
False
>>> 5 in f
True
```

Bloom Filters – Example



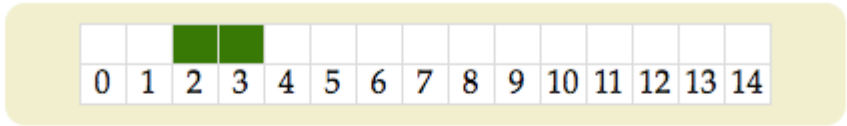
Each empty cell in that table represents a bit, and the number below it its index. To add an element to the Bloom filter, we simply hash it a few times and set the bits in the bit vector at the index of those hashes to 1.

It's easier to see what that means than explain it, so enter some strings and see how the bit vector changes. Fnv and Murmur are two simple hash functions:



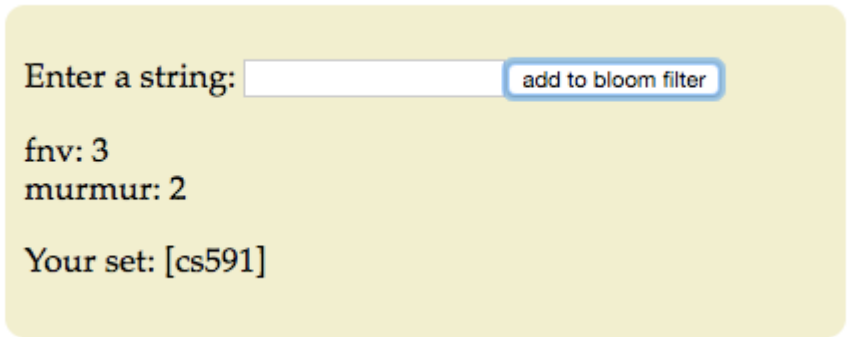
Demo: Bloom Filters by Example

Bloom Filters – Example



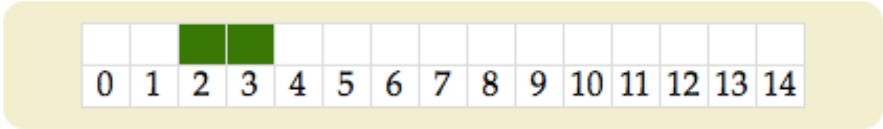
Each empty cell in that table represents a bit, and the number below it its index. To add an element to the Bloom filter, we simply hash it a few times and set the bits in the bit vector at the index of those hashes to 1.

It's easier to see what that means than explain it, so enter some strings and see how the bit vector changes. Fnv and Murmur are two simple hash functions:



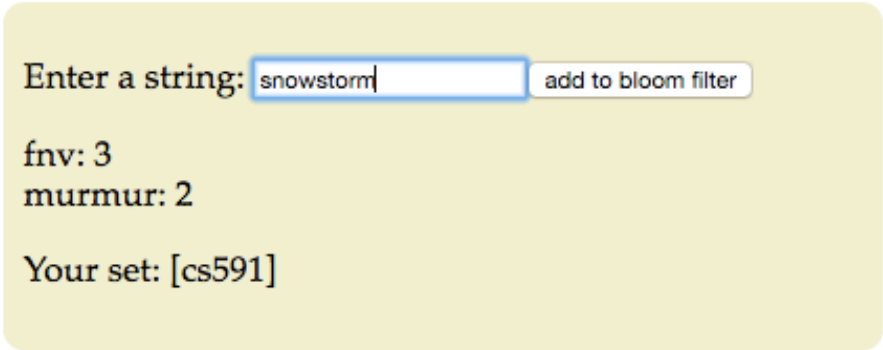
Demo: Bloom Filters by Example

Bloom Filters – Example



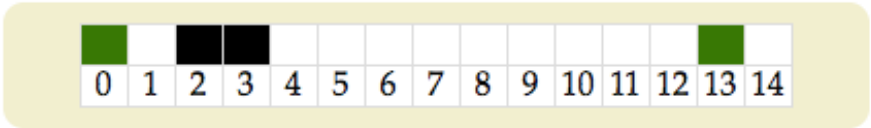
Each empty cell in that table represents a bit, and the number below it its index. To add an element to the Bloom filter, we simply hash it a few times and set the bits in the bit vector at the index of those hashes to 1.

It's easier to see what that means than explain it, so enter some strings and see how the bit vector changes. Fnv and Murmur are two simple hash functions:



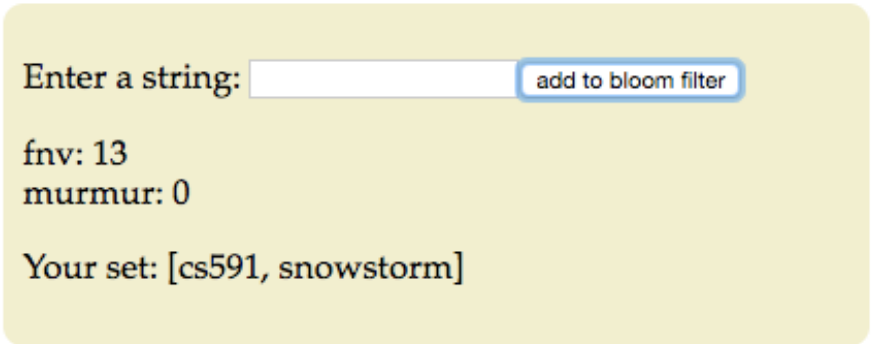
Demo: Bloom Filters by Example

Bloom Filters – Example



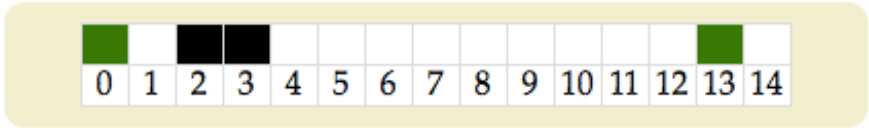
Each empty cell in that table represents a bit, and the number below it its index. To add an element to the Bloom filter, we simply hash it a few times and set the bits in the bit vector at the index of those hashes to 1.

It's easier to see what that means than explain it, so enter some strings and see how the bit vector changes. Fnv and Murmur are two simple hash functions:



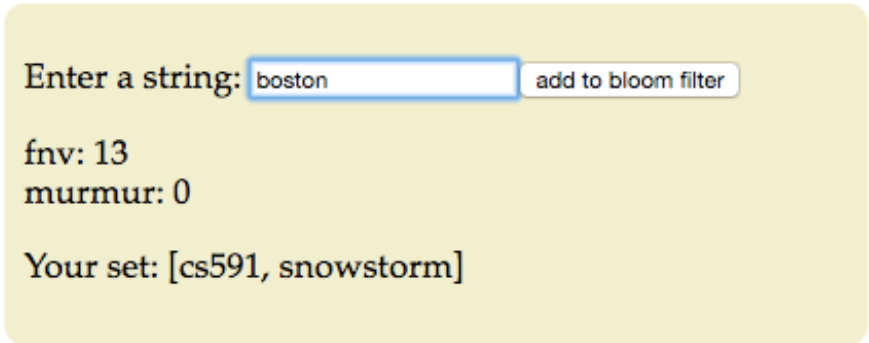
Demo: Bloom Filters by Example

Bloom Filters – Example



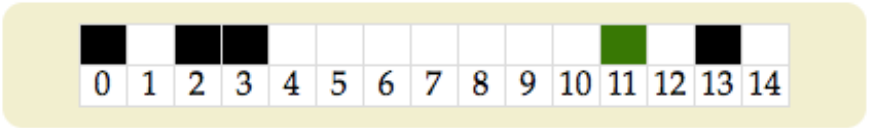
Each empty cell in that table represents a bit, and the number below it its index. To add an element to the Bloom filter, we simply hash it a few times and set the bits in the bit vector at the index of those hashes to 1.

It's easier to see what that means than explain it, so enter some strings and see how the bit vector changes. Fnv and Murmur are two simple hash functions:



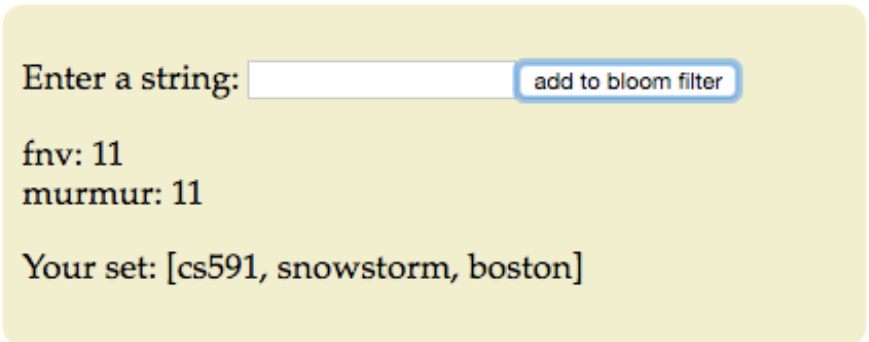
Demo: Bloom Filters by Example

Bloom Filters – Example



Each empty cell in that table represents a bit, and the number below it its index. To add an element to the Bloom filter, we simply hash it a few times and set the bits in the bit vector at the index of those hashes to 1.

It's easier to see what that means than explain it, so enter some strings and see how the bit vector changes. Fnv and Murmur are two simple hash functions:



Demo: Bloom Filters by Example

Bloom Filters – False positives

- **Assumption:** h_i are close to being independent hash functions, probes are uniform
- **Claim:** $\mathbf{Pr}\left[A(i) = 1\right] = 1 - (1 - \frac{1}{m})^{kn}$
- **Why?**

Bloom Filters – False positives

- Probability of a false positive.

$$p_f = \left(1 - (1 - \frac{1}{m})^{kn}\right)^k \approx (1 - e^{kn/m})^k.$$

Given n, m how do we optimally set k?

$$k = \log(2)\frac{m}{n}.$$

Bloom Filters – Do we need k hash functions?

Double hashing works!

- Instead of using k random hash functions, one can choose two sufficiently random hash functions h, h' and then set

$$h_i(x) = h(x) + ih'(x) \bmod m.$$

- This was proved by Kirsch et al.

- Dillinger and Manolios had earlier suggested

$$h_i(x) = h(x) + ih'(x) + i^2 \bmod m,$$

as an effective heuristic.

Spell Checker with Bloom Filters

https://github.com/tsourolampis/bloom-spell-checker

references I

Carter, J. L. and Wegman, M. N. (1979).

Universal classes of hash functions.

Journal of computer and system sciences, 18(2):143–154.

Dietzfelbinger, M., Hagerup, T., Katajainen, J., and Penttonen, M. (1997).

A reliable randomized algorithm for the closest-pair problem.

Journal of Algorithms, 25(1):19–51.