

**CSE 312**

# **Foundations of Computing II**

**Lecture 11: Bloom Filters**

## Last Class:

- Variance
- Independent random variables

## Today:

- Properties of Independent RVs ◀
- An application: Bloom Filters!

Kandinsky

[This Photo](#) by Unknown Author is licensed under [CC BY-SA](#)



## Recap Variance – Properties

**Definition.** The **variance** of a (discrete) RV  $X$  is

$$\text{Var}(X) = \mathbb{E}[(X - \mathbb{E}[X])^2] = \sum_x p_X(x) \cdot (x - \mathbb{E}[X])^2$$

**Theorem.** For any  $a, b \in \mathbb{R}$ ,  $\text{Var}(a \cdot X + b) = a^2 \cdot \text{Var}(X)$

**Theorem.**  $\text{Var}(X) = \mathbb{E}[X^2] - \mathbb{E}[X]^2$

## Recap Variance of Indicator Random Variables

Suppose that  $X_A$  is an indicator RV for event  $A$  with  $P(A) = p$  so

$$\mathbb{E}[X_A] = P(A) = p$$

Since  $X_A$  only takes on values 0 and 1, we always have  $X_A^2 = X_A$  so

$$\text{Var}(X_A) = \mathbb{E}[X_A^2] - \mathbb{E}[X_A]^2 = \mathbb{E}[X_A] - \mathbb{E}[X_A]^2 = p - p^2 = p(1 - p)$$

## Important Facts about Independent Random Variables

**Theorem.** If  $X, Y$  independent,  $\mathbb{E}[X \cdot Y] = \mathbb{E}[X] \cdot \mathbb{E}[Y]$

**Theorem.** If  $X, Y$  independent,  $\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y)$

**Corollary.** If  $X_1, X_2, \dots, X_n$  mutually independent,

$$\text{Var}\left(\sum_{i=1}^n X_i\right) = \sum_i \text{Var}(X_i)$$

## Example – Coin Tosses

We flip  $n$  independent coins, each one heads with probability  $p$

- $X_i = \begin{cases} 1, & i^{\text{th}} \text{ outcome is heads} \\ 0, & i^{\text{th}} \text{ outcome is tails.} \end{cases}$
- $Z = \text{number of heads}$

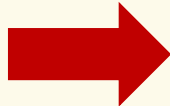
$$\text{Fact. } Z = \sum_{i=1}^n X_i$$

$$\begin{aligned} P(X_i = 1) &= p \\ P(X_i = 0) &= 1 - p \end{aligned}$$

What is  $\mathbb{E}[Z]$ ? What is  $\text{Var}(Z)$ ?

$$P(Z = k) = \binom{n}{k} p^k (1 - p)^{n-k}$$

Note:  $X_1, \dots, X_n$  are mutually independent! [Verify it formally!]


$$\text{Var}(Z) = \sum_{i=1}^n \text{Var}(X_i) = n \cdot p(1 - p)$$

$$\text{Note } \text{Var}(X_i) = p(1 - p)$$

## (Not Covered) Proof of $\mathbb{E}[X \cdot Y] = \mathbb{E}[X] \cdot \mathbb{E}[Y]$

**Theorem.** If  $X, Y$  independent,  $\mathbb{E}[X \cdot Y] = \mathbb{E}[X] \cdot \mathbb{E}[Y]$

### Proof

Let  $x_i, y_i, i = 1, 2, \dots$  be the possible values of  $X, Y$ .

$$\begin{aligned}\mathbb{E}[X \cdot Y] &= \sum_i \sum_j x_i \cdot y_j \cdot P(X = x_i \wedge Y = y_j) \\ &= \sum_i \sum_j x_i \cdot y_j \cdot P(X = x_i) \cdot P(Y = y_j) \quad \text{independence} \\ &= \sum_i x_i \cdot P(X = x_i) \cdot \left( \sum_j y_j \cdot P(Y = y_j) \right) \\ &= \mathbb{E}[X] \cdot \mathbb{E}[Y]\end{aligned}$$

Note: NOT true in general; see earlier example  $\mathbb{E}[X^2] \neq \mathbb{E}[X]^2$

## (Not Covered) Proof of $\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y)$

**Theorem.** If  $X, Y$  independent,  $\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y)$

### Proof

$$\begin{aligned} & \text{Var}(X + Y) \\ &= \mathbb{E}[(X + Y)^2] - (\mathbb{E}[X + Y])^2 \\ &= \mathbb{E}[X^2 + 2XY + Y^2] - (\mathbb{E}[X] + \mathbb{E}[Y])^2 \\ &= \mathbb{E}[X^2] + 2 \mathbb{E}[XY] + \mathbb{E}[Y^2] - (\mathbb{E}[X]^2 + 2 \mathbb{E}[X] \mathbb{E}[Y] + \mathbb{E}[Y]^2) \\ &= \mathbb{E}[X^2] - \mathbb{E}[X]^2 + \mathbb{E}[Y^2] - \mathbb{E}[Y]^2 + 2 \mathbb{E}[XY] - 2 \mathbb{E}[X] \mathbb{E}[Y] \\ &= \text{Var}(X) + \text{Var}(Y) + 2 \mathbb{E}[XY] - 2 \mathbb{E}[X] \mathbb{E}[Y] \\ &= \text{Var}(X) + \text{Var}(Y) \end{aligned}$$

**linearity**

**equal by independence**



## Last Class:

- Variance
- Independent random variables

## Today:

- Properties of Independent RVs
- An application: Bloom Filters! ◀

Kandinsky

[This Photo](#) by Unknown Author is licensed under [CC BY-SA](#)



## Basic Problem

**Problem:** Store a subset  $S$  of a large set  $U$ .

**Example.**  $U$  = set of 128 bit strings  
 $S$  = subset of strings of interest

$$|U| \approx 2^{128}$$
$$|S| \approx 1000$$

### Two goals:

1. **Very fast** (ideally constant time) answers to queries “Is  $x \in S$ ?” for any  $x \in U$ .
2. **Minimal storage** requirements.

## Naïve Solution I – Constant Time

**Idea:** Represent  $S$  as an array  $A$  with  $2^{128}$  entries.

$$A[x] = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{if } x \notin S \end{cases}$$

$S = \{0, 2, \dots, K\}$



0	1	2	...	K	...		
1	0	1	0	1	...	0	0

**Membership test:** To check  $x \in S$  just check whether  $A[x] = 1$ .

→ constant time!  

**Storage:** Require storing  $2^{128}$  bits, even for small  $S$ .



## Naïve Solution II – Small Storage

**Idea:** Represent  $S$  as a list with  $|S|$  entries.



**Storage:** Grows with  $|S|$  only  

**Membership test:** Check  $x \in S$  requires time linear in  $|S|$

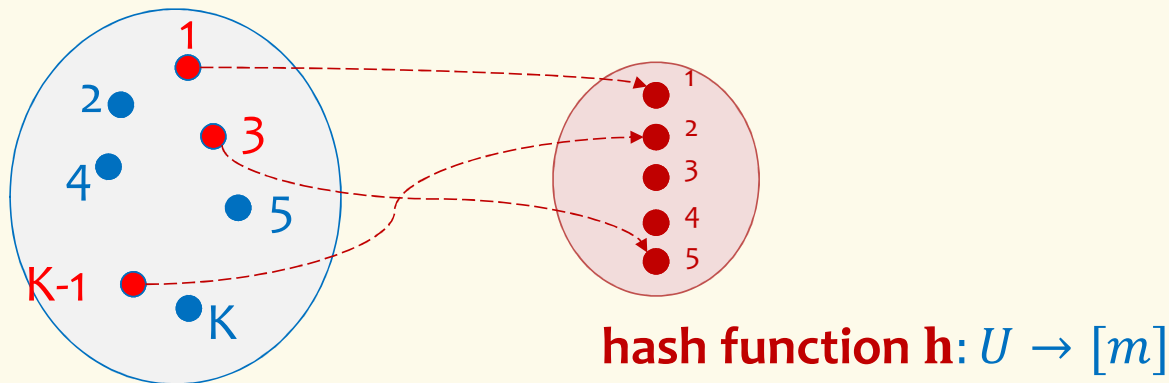
(Can be made logarithmic by using a tree)  

# Hash Table

**Idea:** Map elements in  $S$  into an array  $A$  of size  $m$  using a hash function  $h$

**Membership test:** To check  $x \in S$  just check whether  $A[h(x)] = x$

**Storage:**  $m$  elements (size of array)

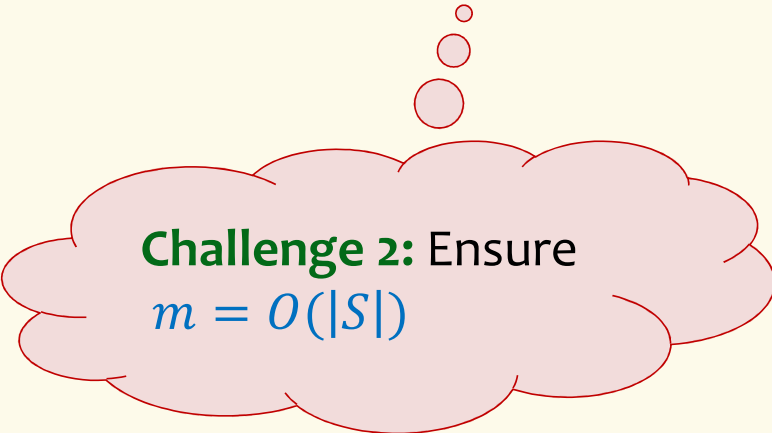


# Hash Table

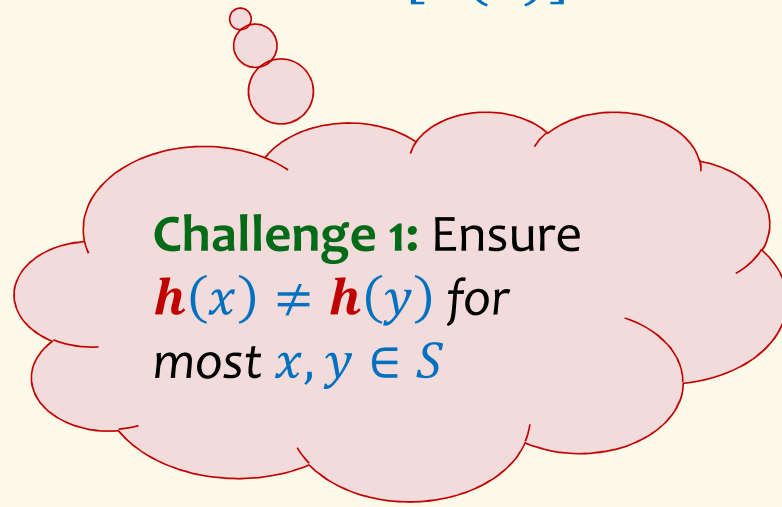
**Idea:** Map elements in  $S$  into an array  $A$  of size  $m$  using a hash function  $h$

**Membership test:** To check  $x \in S$  just check whether  $A[h(x)] = x$

**Storage:**  $m$  elements (size of array)



**Challenge 2:** Ensure  
 $m = O(|S|)$

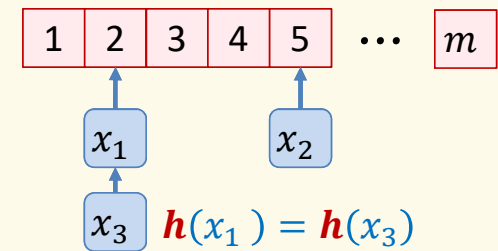


**Challenge 1:** Ensure  
 $h(x) \neq h(y)$  for  
most  $x, y \in S$

## Hashing: collisions

**Collisions** occur when  $h(x) = h(y)$  for some distinct  $x, y \in S$ , i.e., two elements of set map to the same location

- Common solution: chaining – at each location (bucket) in the table, keep linked list of all elements that hash there.



## Good hash functions to keep collisions low

- The hash function  $h$  is good iff it
  - distributes elements uniformly across the  $m$  array locations so that
  - pairs of elements are mapped independently

“Universal Hash Functions” – see CSE 332



# Hashing: summary

## Hash Tables

- They store the data itself
- With a good hash function, the data is well distributed in the table and lookup times are small.
- However, they need at least as much space as all the data being stored, i.e.,  $m = \Omega(|S|)$

In some cases,  $|S|$  is huge, or not known a-priori ...

Can we do better!?



**Bloom Filters**

**to the rescue**

(Named after Burton Howard Bloom)

## Bloom Filters – Main points

- Probabilistic data structure.
- Close cousins of hash tables.
  - But: Ridiculously space efficient
- Occasional errors, specifically false positives.

## Bloom Filters

- Stores information about a set of elements  $S \subseteq U$ .
- Supports two operations:
  1. **add**( $x$ ) - adds  $x \in U$  to the set  $S$
  2. **contains**( $x$ ) – ideally: true if  $x \in S$ , false otherwise

### Relaxed guarantees:

- False  $\rightarrow$  **definitely** not in  $S$
- True  $\rightarrow$  **possibly** in  $S$   
[i.e. we could have *false positives*]

## Bloom Filters – Why Accept False Positives?

- **Speed** – both **add** and **contains** very very fast.
- **Space** – requires a miniscule amount of space relative to storing all the actual items that have been added.
  - Often just 8 bits per inserted item!
- **Fallback mechanism** – can distinguish false positives from true positives with extra cost
  - Ok if mostly negatives expected + low false positive rate

## Bloom Filters: Application

- Google Chrome has a database of malicious URLs, but it takes a long time to query.
- Want an in-browser structure, so needs to be efficient and be space-efficient
- Want it so that can check if a URL is in structure:
  - If return False, then definitely not in the structure (don't need to do expensive database lookup, website is safe)
  - If return True, the URL may or may not be in the structure. Have to perform expensive lookup in this rare case.

## Bloom Filters – More Applications

- Any scenario where space and efficiency are important.
- Used a lot in networking
- In distributed systems when want to check consistency of data across different locations, might send a Bloom filter rather than the full set of data being stored.
- Google BigTable uses Bloom filters to reduce disk lookups
- Internet routers often use Bloom filters to track blocked IP addresses.
- And on and on...

## What you can't do with Bloom filters

- There is no **delete** operation
  - Once you have added something to a Bloom filter for  $S$ , it stays
- You can't use a Bloom filter to name any element of  $S$

But what you **can** do makes them very effective!



## Bloom Filters – Ingredients

Basic data structure is a  $k \times m$  binary array  
“the Bloom filter”

- $k$  rows  $t_1, \dots, t_k$ , each of size  $m$
- Think of each row as an  $m$ -bit vector

$k$  different hash functions  $\mathbf{h}_1, \dots, \mathbf{h}_k: U \rightarrow [m]$

## Bloom Filters – Three operations

- Set up Bloom filter for  $S = \emptyset$

```
function INITIALIZE( $k, m$ )  
  for  $i = 1, \dots, k$ : do  
     $t_i =$  new bit vector of  $m$  0s
```

- Update Bloom filter for  $S \leftarrow S \cup \{x\}$

```
function ADD( $x$ )  
  for  $i = 1, \dots, k$ : do  
     $t_i[h_i(x)] = 1$ 
```

- Check if  $x \in S$

```
function CONTAINS( $x$ )  
  return  $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \dots \wedge t_k[h_k(x)] == 1$ 
```

## Bloom Filters - Initialization

Number of  
hash  
functions

Size of array  
associated to  
each hash  
function.

```
function INITIALIZE( $k, m$ )  
  for  $i = 1, \dots, k$ : do  
     $t_i =$  new bit vector of  $m$  0s
```

for each hash  
function, initialize  
an empty bit  
vector of size  $m$

## Bloom Filters: Example

Bloom filter  $t$  of length  $m = 5$  that uses  $k = 3$  hash functions

```
function INITIALIZE( $k, m$ )  
  for  $i = 1, \dots, k$ : do  
     $t_i =$  new bit vector of  $m$  0s
```

Index →	0	1	2	3	4
$t_1$	0	0	0	0	0
$t_2$	0	0	0	0	0
$t_3$	0	0	0	0	0

## Bloom Filters: Add

```
function ADD( $x$ )  
  for  $i = 1, \dots, k$ : do  
     $t_i[h_i(x)] = 1$ 
```

→ for each hash function  $\mathbf{h}_i$

Index into  $i$ -th bit-vector, at index produced by hash function and set to 1

$\mathbf{h}_i(x) \rightarrow$  result of hash function  $\mathbf{h}_i$  on  $x$

## Bloom Filters: Example

Bloom filter  $t$  of length  $m = 5$  that uses  $k = 3$  hash functions

```
function ADD( $x$ )  
  for  $i = 1, \dots, k$ : do  
     $t_i[h_i(x)] = 1$ 
```

add("thisisavirus.com")

$h_1(\text{"thisisavirus.com"}) \rightarrow 2$

Index →	0	1	2	3	4
$t_1$	0	0	0	0	0
$t_2$	0	0	0	0	0
$t_3$	0	0	0	0	0

## Bloom Filters: Example

Bloom filter  $t$  of length  $m = 5$  that uses  $k = 3$  hash functions

```
function ADD( $x$ )  
  for  $i = 1, \dots, k$ : do  
     $t_i[h_i(x)] = 1$ 
```

add("thisisavirus.com")

$h_1$ ("thisisavirus.com")  $\rightarrow$  2

$h_2$ ("thisisavirus.com")  $\rightarrow$  1

Index $\rightarrow$	0	1	2	3	4
$t_1$	0	0	1	0	0
$t_2$	0	0	0	0	0
$t_3$	0	0	0	0	0

## Bloom Filters: Example

Bloom filter  $t$  of length  $m = 5$  that uses  $k = 3$  hash functions

```
function ADD( $x$ )  
  for  $i = 1, \dots, k$ : do  
     $t_i[h_i(x)] = 1$ 
```

add("thisisavirus.com")

$h_1$ ("thisisavirus.com")  $\rightarrow$  2

$h_2$ ("thisisavirus.com")  $\rightarrow$  1

$h_3$ ("thisisavirus.com")  $\rightarrow$  4

Index $\rightarrow$	0	1	2	3	4
$t_1$	0	0	1	0	0
$t_2$	0	1	0	0	0
$t_3$	0	0	0	0	0



## Bloom Filters: Example

Bloom filter  $t$  of length  $m = 5$  that uses  $k = 3$  hash functions

```
function ADD( $x$ )  
  for  $i = 1, \dots, k$ : do  
     $t_i[h_i(x)] = 1$ 
```

**add("thisisavirus.com")**

$h_1(\text{"thisisavirus.com"}) \rightarrow 2$

$h_2(\text{"thisisavirus.com"}) \rightarrow 1$

$h_3(\text{"thisisavirus.com"}) \rightarrow 4$

Index →	0	1	2	3	4
$t_1$	0	0	1	0	0
$t_2$	0	1	0	0	0
$t_3$	0	0	0	0	1

## Bloom Filters: Contains

```
function CONTAINS( $x$ )  
  return  $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \dots \wedge t_k[h_k(x)] == 1$ 
```

Returns True if the bit vector  $t_i$  for each hash function has bit 1 at index determined by  $h_i(x)$ ,  
Returns False otherwise

## Bloom Filters: Example

Bloom filter  $t$  of length  $m = 5$  that uses  $k = 3$  hash functions

```
function CONTAINS(x)
  return  $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \dots \wedge t_k[h_k(x)] == 1$ 
```

**contains("thisisavirus.com")**

Index →	0	1	2	3	4
$t_1$	0	0	<b>1</b>	0	0
$t_2$	0	<b>1</b>	0	0	0
$t_3$	0	0	0	0	<b>1</b>

## Bloom Filters: Example

Bloom filter  $t$  of length  $m = 5$  that uses  $k = 3$  hash functions

```
function CONTAINS(x)
  return  $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \dots \wedge t_k[h_k(x)] == 1$ 
```

**True**

**contains("thisisavirus.com")**

$h_1(\text{"thisisavirus.com"}) \rightarrow 2$

Index →	0	1	2	3	4
$t_1$	0	0	1	0	0
$t_2$	0	1	0	0	0
$t_3$	0	0	0	0	1

## Bloom Filters: Example

Bloom filter  $t$  of length  $m = 5$  that uses  $k = 3$  hash functions

```
function CONTAINS(x)
  return  $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \dots \wedge t_k[h_k(x)] == 1$ 
```

**True**

**True**

**contains("thisisavirus.com")**

$h_1(\text{"thisisavirus.com"}) \rightarrow 2$

$h_2(\text{"thisisavirus.com"}) \rightarrow 1$

Index →	0	1	2	3	4
$t_1$	0	0	1	0	0
$t_2$	0	1	0	0	0
$t_3$	0	0	0	0	1

# Bloom Filters: Example

Bloom filter  $t$  of length  $m = 5$  that uses  $k = 3$  hash functions

```
function CONTAINS(x)
  return  $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \dots \wedge t_k[h_k(x)] == 1$ 
```

True

True

True

**contains("thisisavirus.com")**

$h_1(\text{"thisisavirus.com"}) \rightarrow 2$

$h_2(\text{"thisisavirus.com"}) \rightarrow 1$

$h_3(\text{"thisisavirus.com"}) \rightarrow 4$

Index →	0	1	2	3	4
$t_1$	0	0	1	0	0
$t_2$	0	1	0	0	0
$t_3$	0	0	0	0	1

# Bloom Filters: Example

Bloom filter  $t$  of length  $m = 5$  that uses  $k = 3$  hash functions

```
function CONTAINS(x)
  return  $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \dots \wedge t_k[h_k(x)] == 1$ 
```

True

True

True

contains("thisisavirus.com")

$h_1$ ("thisisavirus.com")  $\rightarrow$  2

$h_2$ ("thisisavirus.com")  $\rightarrow$  1

$h_3$ ("thisisavirus.com")  $\rightarrow$  4

Index	0	1	2	3	4
$t_1$	0	0	1	0	0
$t_2$	0	1	0	0	0
$t_3$	0	0	0	0	1

Since all conditions satisfied, returns **True** (correctly)

## Bloom Filters: False Positives

Bloom filter  $t$  of length  $m = 5$  that uses  $k = 3$  hash functions

`add("totallynotsuspicious.com")`

```
function ADD( $x$ )  
  for  $i = 1, \dots, k$ : do  
     $t_i[h_i(x)] = 1$ 
```

Index	0	1	2	3	4
→					
$t_1$	0	0	1	0	0
$t_2$	0	1	0	0	0
$t_3$	0	0	0	0	1



## Bloom Filters: False Positives

Bloom filter  $t$  of length  $m = 5$  that uses  $k = 3$  hash functions

```
function ADD( $x$ )  
  for  $i = 1, \dots, k$ : do  
     $t_i[h_i(x)] = 1$ 
```

add("totallynotsuspicious.com")

$h_1$ ("totallynotsuspicious.com")  $\rightarrow$  1

Index $\rightarrow$	0	1	2	3	4
$t_1$	0	0	1	0	0
$t_2$	0	1	0	0	0
$t_3$	0	0	0	0	1

## Bloom Filters: False Positives

Bloom filter  $t$  of length  $m = 5$  that uses  $k = 3$  hash functions

```
function ADD( $x$ )  
  for  $i = 1, \dots, k$ : do  
     $t_i[h_i(x)] = 1$ 
```

**add("totallynotsuspicious.com")**

$h_1(\text{"totallynotsuspicious.com"}) \rightarrow 1$

$h_2(\text{"totallynotsuspicious.com"}) \rightarrow 0$

Index →	0	1	2	3	4
$t_1$	0	1	1	0	0
$t_2$	0	1	0	0	0
$t_3$	0	0	0	0	1

## Bloom Filters: False Positives

Bloom filter  $t$  of length  $m = 5$  that uses  $k = 3$  hash functions

```
function ADD( $x$ )  
  for  $i = 1, \dots, k$ : do  
     $t_i[h_i(x)] = 1$ 
```

**add("totallynotsuspicious.com")**

$h_1(\text{"totallynotsuspicious.com"}) \rightarrow 1$

$h_2(\text{"totallynotsuspicious.com"}) \rightarrow 0$

$h_3(\text{"totallynotsuspicious.com"}) \rightarrow 4$

Index →	0	1	2	3	4
$t_1$	0	1	1	0	0
$t_2$	1	1	0	0	0
$t_3$	0	0	0	0	1

## Bloom Filters: False Positives

Bloom filter  $t$  of length  $m = 5$  that uses  $k = 3$  hash functions

```
function ADD( $x$ )  
  for  $i = 1, \dots, k$ : do  
     $t_i[h_i(x)] = 1$ 
```

**add("totallynotsuspicious.com")**

$h_1(\text{"totallynotsuspicious.com"}) \rightarrow 1$

$h_2(\text{"totallynotsuspicious.com"}) \rightarrow 0$

$h_3(\text{"totallynotsuspicious.com"}) \rightarrow 4$

Index →	0	1	2	3	4
$t_1$	0	1	1	0	0
$t_2$	1	1	0	0	0
$t_3$	0	0	0	0	1

## Bloom Filters: False Positives

Bloom filter  $t$  of length  $m = 5$  that uses  $k = 3$  hash functions

```
function CONTAINS(x)  
  return  $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \dots \wedge t_k[h_k(x)] == 1$ 
```

contains("verynormalsite.com")

Index →	0	1	2	3	4
$t_1$	0	1	1	0	0
$t_2$	1	1	0	0	0
$t_3$	0	0	0	0	1

## Bloom Filters: False Positives

Bloom filter  $t$  of length  $m = 5$  that uses  $k = 3$  hash functions

```
function CONTAINS(x)
  return  $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \dots \wedge t_k[h_k(x)] == 1$ 
```

**True**

**contains("verynormalsite.com")**

$h_1(\text{"verynormalsite.com"}) \rightarrow 2$

Index →	0	1	2	3	4
$t_1$	0	1	1	0	0
$t_2$	1	1	0	0	0
$t_3$	0	0	0	0	1

## Bloom Filters: False Positives

Bloom filter  $t$  of length  $m = 5$  that uses  $k = 3$  hash functions

```
function CONTAINS(x)
  return  $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \dots \wedge t_k[h_k(x)] == 1$ 
```

**True**

**True**

**contains("verynormalsite.com")**

$h_1(\text{"verynormalsite.com"}) \rightarrow 2$

$h_2(\text{"verynormalsite.com"}) \rightarrow 0$

Index →	0	1	2	3	4
$t_1$	0	1	1	0	0
$t_2$	1	1	0	0	0
$t_3$	0	0	0	0	1

# Bloom Filters: False Positives

Bloom filter  $t$  of length  $m = 5$  that uses  $k = 3$  hash functions

```
function CONTAINS(x)
  return  $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \dots \wedge t_k[h_k(x)] == 1$ 
```

True

True

True

contains("verynormalsite.com")

$h_1$ ("verynormalsite.com")  $\rightarrow$  2

$h_2$ ("verynormalsite.com")  $\rightarrow$  0

$h_3$ ("verynormalsite.com")  $\rightarrow$  4

Index $\rightarrow$	0	1	2	3	4
$t_1$	0	1	1	0	0
$t_2$	1	1	0	0	0
$t_3$	0	0	0	0	1



# Bloom Filters: False Positives

Bloom filter  $t$  of length  $m = 5$  that uses  $k = 3$  hash functions

```
function CONTAINS(x)  
  return  $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \dots \wedge t_k[h_k(x)] == 1$ 
```

True

True

True

contains("verynormalsite.com")

$h_1$ ("verynormalsite.com")  $\rightarrow$  2

$h_2$ ("verynormalsite.com")  $\rightarrow$  0

$h_3$ ("verynormalsite.com")  $\rightarrow$  4

Index	0	1	2	3	4
$t_1$	0	1	1	0	0
$t_2$	1	1	0	0	0
$t_3$	0	0	0	0	1

Since all conditions satisfied, returns **True** (incorrectly)

# Brain Break



## Analysis: False positive probability

**Question:** For an element  $x \in U$ , what is the probability that **contains**( $x$ ) returns true if **add**( $x$ ) was never executed before?

Probability over what?! Over the choice of the  $h_1, \dots, h_k$

Assumptions for the analysis (somewhat stronger than for ordinary hashing):

- Each  $h_i(x)$  is uniformly distributed in  $[m]$  for all  $x$  and  $i$
- Hash function outputs for each  $h_i$  are mutually independent (not just in pairs)
- Different hash functions are independent of each other

## False positive probability – Events

Assume we perform **add**( $x_1$ ), ..., **add**( $x_n$ ) but  $x \notin \{x_1, \dots, x_n\}$

Event  $E_i$  holds iff  $\mathbf{h}_i(x) \in \{\mathbf{h}_i(x_1), \dots, \mathbf{h}_i(x_n)\}$

$$P(\text{false positive}) = P(E_1 \cap E_2 \cap \dots \cap E_k) = \prod_{i=1}^k P(E_i)$$

$\mathbf{h}_1, \dots, \mathbf{h}_k$  independent



## False positive probability – Events

Event  $E_i$  holds iff  $\mathbf{h}_i(x) \in \{\mathbf{h}_i(x_1), \dots, \mathbf{h}_i(x_n)\}$

Event  $E_i^c$  holds iff  $\mathbf{h}_i(x) \neq \mathbf{h}_i(x_1)$  and ... and  $\mathbf{h}_i(x) \neq \mathbf{h}_i(x_n)$

$$P(E_i^c) = \sum_{z=1}^m P(\mathbf{h}_i(x) = z) \cdot P(E_i^c \mid \mathbf{h}_i(x) = z)$$

LTP



## False positive probability – Events

Event  $E_i^c$  holds iff  $\mathbf{h}_i(x) \neq \mathbf{h}_i(x_1)$  and ...  
and  $\mathbf{h}_i(x) \neq \mathbf{h}_i(x_n)$

$$P(E_i^c | \mathbf{h}_i(x) = z) = P(\mathbf{h}_i(x_1) \neq z, \dots, \mathbf{h}_i(x_n) \neq z | \mathbf{h}_i(x) = z)$$

Independence of values  
of  $\mathbf{h}_i$  on different inputs

$$= P(\mathbf{h}_i(x_1) \neq z, \dots, \mathbf{h}_i(x_n) \neq z)$$

$$= \prod_{j=1}^n P(\mathbf{h}_i(x_j) \neq z)$$

Outputs of  $\mathbf{h}_i$  uniformly spread

$$= \prod_{j=1}^n \left(1 - \frac{1}{m}\right) = \left(1 - \frac{1}{m}\right)^n$$


$$\Rightarrow P(E_i^c) = \sum_{z=1}^m P(\mathbf{h}_i(x) = z) \cdot P(E_i^c | \mathbf{h}_i(x) = z) = \left(1 - \frac{1}{m}\right)^n$$

## False positive probability – Events

Event  $E_i$  holds iff  $\mathbf{h}_i(x) \in \{\mathbf{h}_i(x_1), \dots, \mathbf{h}_i(x_n)\}$

Event  $E_i^c$  holds iff  $\mathbf{h}_i(x) \neq \mathbf{h}_i(x_1)$  and ... and  $\mathbf{h}_i(x) \neq \mathbf{h}_i(x_n)$

$$P(E_i^c) = \left(1 - \frac{1}{m}\right)^n$$


$$\text{FPR} = \prod_{i=1}^k (1 - P(E_i^c)) = \left(1 - \left(1 - \frac{1}{m}\right)^n\right)^k$$

## False Positivity Rate\_– Example

$$\text{FPR} = \left( 1 - \left( 1 - \frac{1}{m} \right)^n \right)^k$$

e.g.,  $n = 5,000,000$

$k = 30$

$m = 2,500,000$



FPR = 1.28%



## Comparison with Hash Tables - Space

- Google storing 5 million URLs, each URL 40 bytes.
- Bloom filter with  $k = 30$  and  $m = 2,500,000$

### Hash Table

(optimistic)

$$5,000,000 \times 40B = 200MB$$

### Bloom Filter

$$2,500,000 \times 30 = 75,000,000 \text{ bits}$$

$$< 10 \text{ MB}$$

## Comparison with Hash Tables - Time

- Say avg user visits **102,000** URLs in a year, of which **2,000** are malicious.
- **0.5** seconds to do lookup in the database, **1ms** for lookup in Bloom filter.
- Suppose the false positive rate is **3%**

$$1\text{ms} + \frac{100000 \times 0.03 \times 500\text{ms} + 2000 \times 500\text{ms}}{102000} \approx 25.51\text{ms}$$

**Bloom filter lookup** (points to 1ms)

**false positives** (points to 100000 × 0.03)

**total URLs** (points to 102000)

**0.5 seconds DB lookup** (points to 500ms)

**malicious URLs** (points to 2000)

## Bloom Filters typical of....

... randomized algorithms and randomized data structures.

- **Simple**
- **Fast**
- **Efficient**
- **Elegant**
- **Useful!**