CSE 312

Foundations of Computing II

Lecture 10: Bloom Filter

Announcements

- PSet 3 due today
- PSet 2 ~~returned yesterday~~ today
- PSet 4 will be posted today
  - – Last PSet prior to midterm (midterm is in exactly two weeks from now)
  - – Midterm info will follow soon
  - – PSet 5 will only come after the midterm in two weeks
- Midterm feedback/evaluation to come soon (Tomorrow or Friday).

Recap Variance – Properties

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

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

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

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

Agenda

- Variance
- Properties of Variance
- Independent Random Variables
- Properties of Independent Random Variables ◀
- An Application: Bloom Filters!

Important Facts about Independent Random Variables

$E[X+Y] = E[X] + E[Y]$

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

$\mathrm{Var}(X \cdot Y) \neq$????

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

Corollary. If $X_1, X_2, \ldots, X_n$ mutually independent,

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

Example – Coin Tosses

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

$X_1 \cdot \cdot X_n$

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

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

$P(X_i = 1) = p$
$P(X_i = 0) = 1 - p$

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

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

$n \cdot p$

$\|$

$E(Z^2) - (E(Z))^2$

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

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

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

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

$$\mathbb{E}[X \cdot Y] = \sum_i \sum_j x_i \cdot y_j \cdot P(X = x_i \wedge Y = y_j)$$

independence

$$= \sum_i \sum_j x_i \cdot y_i \cdot P(X = x_i) \cdot P(Y = y_j)$$

$$= \sum_i x_i \cdot P(X = x_i) \cdot \left(\sum_j y_j \cdot P(Y = y_j) \right)$$

$$= \mathbb{E}[X] \cdot \mathbb{E}[Y]$$

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

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

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

Proof

$$Var(X+Y)$$

$$= \mathbb{E}[(X+Y)^2] - (\mathbb{E}[X+Y])^2$$

linearity

$$= \mathbb{E}[X^2 + 2XY + Y^2] - (\mathbb{E}[X] + \mathbb{E}[Y])^2$$

$$= \mathbb{E}[X^2] + 2\,\mathbb{E}[XY] + \mathbb{E}[Y^2] - (\mathbb{E}[X]^2 + 2\,\mathbb{E}[X]\,\mathbb{E}[Y] + \mathbb{E}[Y]^2)$$

$$= \mathbb{E}[X^2] - \mathbb{E}[X]^2 + \mathbb{E}[Y^2] - \mathbb{E}[Y]^2 + 2\,\mathbb{E}[XY] - 2\,\mathbb{E}[X]\,\mathbb{E}[Y]$$

$$= Var(X) + Var(Y) + 2\,\mathbb{E}[XY] - 2\,\mathbb{E}[X]\,\mathbb{E}[Y]$$

$$= Var(X) + Var(Y)$$

equal by independence

Agenda

- Variance
- Properties of Variance
- Independent Random Variables
- Properties of Independent Random Variables
- An Application: Bloom Filters! ◀

Basic Problem

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

Example. $U =$ set of 128 bit strings $\quad |U| \approx 2^{128}$

$S =$ subset of strings of interest $\quad |S| \approx 1000$

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.

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

$S = \{0,2, \dots, K\}$ →

0	1	2	…	K	…		
1	0	1	0	1	…	0	0

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

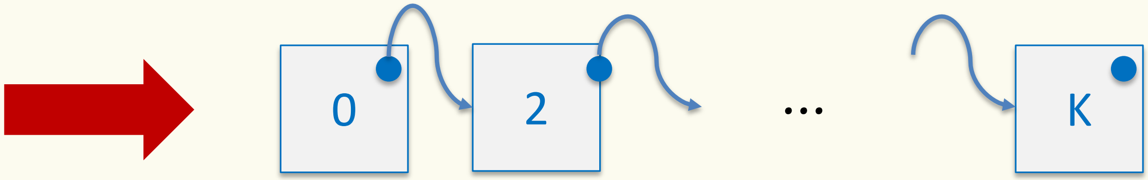
→ **constant time!** 👍 😀

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

Naïve Solution II – Small Storage

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

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



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

$x \in S$ set $A[h(x)] = x$

Idea: Map elements in S into an array A of size m using a hash function $\mathbf{h}$

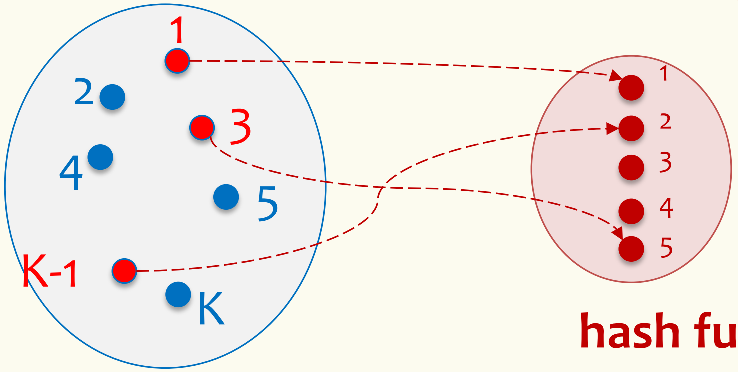
Membership test: To check $x \in S$ just check whether $A[\mathbf{h}(x)] = x$

$A[h(x)] = x'$

Storage: m elements (size of array)

total $m \times |x|$

$x' \in S$ $A[h(x')] = x'$



hash function h: $U \rightarrow [m]$

Hash Table

$x, y \in S$

$A[h(x)] = x$

$A[h(y)] = y$

$h(x) = h(y)$

Idea: Map elements in S into an array A of size m using a hash function $\mathbf{h}$

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

Storage: m elements (size of array)

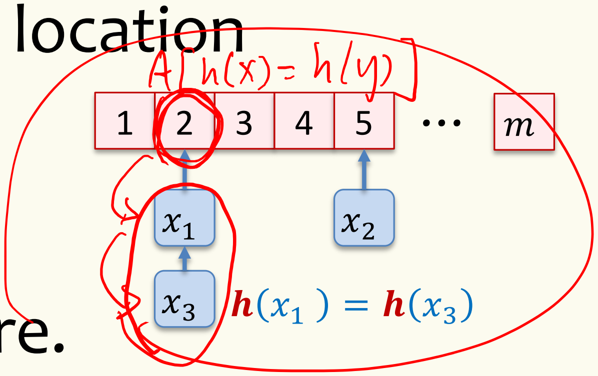
Challenge 1: Ensure $\boldsymbol{h}(x) \neq \boldsymbol{h}(y)$ *for most* $x, y \in S$

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

Hashing: collisions

Collisions occur when $\boldsymbol{h}(x) = \boldsymbol{h}(y)$ 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.



$$\binom{|S|}{2}\cdot\frac{1}{|S|} = \frac{\cancel{|S|}(|S|-1)}{2}\cdot\frac{1}{\cancel{|S|}}$$

A

Q: How many collisions in expectation if the table has size |S| and hash function assigns each x to a random position? $\in [|S|]$

birthdays $\in [365]$

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. $c \cdot |S| \times |X|$
- 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

- 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

Possible *false positives*

Combine with fallback mechanism – can distinguish false positives from true positives with extra cost

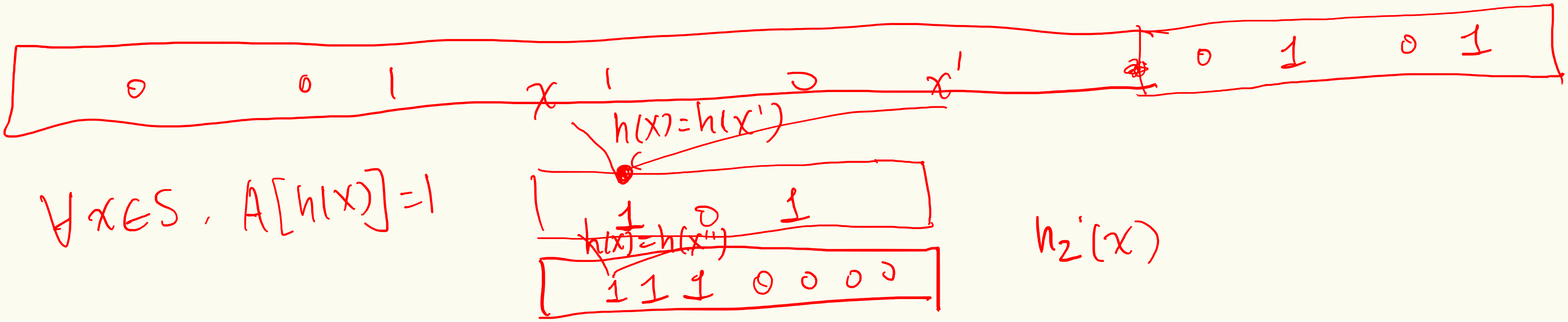
Bloom Filters – Ingredients

t_1	1	0	1	0	0
t_2	0	1	0	0	1
t_3	1	0	0	1	0

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 $\mathbf{h}_1, \dots, \mathbf{h}_k : U \to [m]$



$\forall x \in S,\ A[h(x)]=1$

$h_2'(x)$

Bloom Filters – Three operations

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

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

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

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

- Check if $x \in S$

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

Bloom Filters - Initialization

Number of hash functions

Size of array associated to each hash function.

```
function INITIALIZE($k, m$)
    for $i = 1, \ldots, k$: 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 $\boldsymbol{t}$ of length $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 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** $i = 1, \ldots, k$: **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 $\boldsymbol{m} = 5$ that uses $\boldsymbol{k} = 3$ hash functions

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

add("thisisavirus.com")

h_1("thisisavirus.com") → 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 $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

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

add("thisisavirus.com")

h_1("thisisavirus.com") → 2

h_2("thisisavirus.com") → 1

Index →	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 $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

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

add("thisisavirus.com")

h_1("thisisavirus.com") → 2

h_2("thisisavirus.com") → 1

h_3("thisisavirus.com") → 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	0

Bloom Filters: Example

Bloom filter t of length $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 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") → 2

h_2("thisisavirus.com") → 1

h_3("thisisavirus.com") → 4

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

Bloom Filters: Contains

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

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 $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

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

contains("thisisavirus.com")

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 $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

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

True

contains(“thisisavirus.com”)

h_1(“thisisavirus.com”) → 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 $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

```
**function** CONTAINS($x$)
    **return** $t_1[h_1(x)] == 1 \land t_2[h_2(x)] == 1 \land \cdots \land t_k[h_k(x)] == 1$
```

True True

contains("thisisavirus.com")

h_1("thisisavirus.com") → 2

h_2("thisisavirus.com") → 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 $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

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

True True True

contains(“thisisavirus.com”)

h_1(“thisisavirus.com”) → 2

h_2(“thisisavirus.com”) → 1

h_3(“thisisavirus.com”) → 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 $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

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

True True True

contains("thisisavirus.com")

h_1("thisisavirus.com") → 2

h_2("thisisavirus.com") → 1

h_3("thisisavirus.com") → 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

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

add(“totallynotsuspicious.com”)

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 $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

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

add(“totallynotsuspicious.com”)

h_1(“totallynotsuspicious.com”) → 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 $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 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") → 1

h_2("totallynotsuspicious.com") → 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 $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 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") → 1

h_2("totallynotsuspicious.com") → 0

h_3("totallynotsuspicious.com") → 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 $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

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

add("totallynotsuspicious.com")

h_1("totallynotsuspicious.com") → 1

h_2("totallynotsuspicious.com") → 0

h_3("totallynotsuspicious.com") → 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 $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

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

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 $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

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

True

contains(“verynormalsite.com”)

h_1(“verynormalsite.com”) → 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 $\boldsymbol{m} = 5$ that uses $\boldsymbol{k} = 3$ hash functions

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

True

True

contains("verynormalsite.com")

h_1("verynormalsite.com") → 2

h_2("verynormalsite.com") → 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 $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

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

True True True

contains("verynormalsite.com")

h_1("verynormalsite.com") → 2 0

h_2("verynormalsite.com") → 0 0

h_3("verynormalsite.com") → 4 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 $\boldsymbol{m}$ = 5 that uses $\boldsymbol{k}$ = 3 hash functions

```
**function** CONTAINS($x$)
    **return** $t_1[h_1(x)] == 1 \land t_2[h_2(x)] == 1 \land \cdots \land t_k[h_k(x)] == 1$
```

True True True

contains("verynormalsite.com")

h_1("verynormalsite.com") → 2

h_2("verynormalsite.com") → 0

h_3("verynormalsite.com") → 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)

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 $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_k$

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

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

False positive probability – Events

Assume we perform `add`$(x_1), \dots,$`add`(x_n)
+ `contains`(x) for $x \notin \{x_1, \dots, x_n\}$

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

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

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

False positive probability – Events

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

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

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

LTP

False positive probability – Events

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

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

Independence of values of $\boldsymbol{h}_i$ 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 $\boldsymbol{h}_i$ uniformly spread

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

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

False positive probability – Events

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

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

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

➡

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

False Positivity Rate – Example

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

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)
$5{,}000{,}000 \times 40B = 200\text{MB}$

Bloom Filter

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

< 10 MB

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%

0.5 seconds DB lookup

false positives

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

total URLs

malicious URLs

Bloom filter lookup

Bloom Filters typical of….

… randomized algorithms and randomized data structures.

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