# COMPSCI 514: ALGORITHMS FOR DATA SCIENCE

Andrew McGregor

Lecture 5

- Can sometimes get tighter bounds than Markov via:

$$\Pr[|X - \mathbb{E}[X]| \geq \lambda] = \Pr[|X - \mathbb{E}[X]|^k \geq \lambda^k] \leq \frac{\mathbb{E}[|X - \mathbb{E}[X]|^k]}{\lambda^k}$$

- **Moment Generating Function:** Consider for any $t > 0$:

$$M_t(\mathbf{X}) = e^{t \cdot (\mathbf{X} - \mathbb{E}[\mathbf{X}])}$$

and note $M_t(\mathbf{X})$ is monotonic for any $t > 0$

# EXPONENTIAL CONCENTRATION BOUNDS

- Can sometimes get tighter bounds than Markov via:

$$\Pr[|X - \mathbb{E}[X]| \geq \lambda] = \Pr[|X - \mathbb{E}[X]|^k \geq \lambda^k] \leq \frac{\mathbb{E}[|X - \mathbb{E}[X]|^k]}{\lambda^k}$$

- **Moment Generating Function:** Consider for any $t > 0$:

$$M_t(\mathbf{X}) = e^{t\cdot(\mathbf{X} - \mathbb{E}[\mathbf{X}])} = \sum_{k=0}^{\infty} \frac{t^k(\mathbf{X} - \mathbb{E}[\mathbf{X}])^k}{k!}$$

and note $M_t(\mathbf{X})$ is monotonic for any $t > 0$ and so

$$\Pr[|X - \mathbb{E}[X]| \geq \lambda] = \Pr[M_t(X) \geq e^{t\lambda}] \leq \frac{\mathbb{E}[M_t(X)]}{e^{t\lambda}}$$

- Can sometimes get tighter bounds than Markov via:

$$\Pr[|X - \mathbb{E}[X]| \geq \lambda] = \Pr[|X - \mathbb{E}[X]|^k \geq \lambda^k] \leq \frac{\mathbb{E}[|X - \mathbb{E}[X]|^k]}{\lambda^k}$$

- **Moment Generating Function:** Consider for any $t > 0$:

$$M_t(\mathbf{X}) = e^{t \cdot (\mathbf{X} - \mathbb{E}[\mathbf{X}])} = \sum_{k=0}^{\infty} \frac{t^k (\mathbf{X} - \mathbb{E}[\mathbf{X}])^k}{k!}$$

  and note $M_t(\mathbf{X})$ is monotonic for any $t > 0$ and so

$$\Pr[|X - \mathbb{E}[X]| \geq \lambda] = \Pr[M_t(X) \geq e^{t\lambda}] \leq \frac{\mathbb{E}[M_t(X)]}{e^{t\lambda}}$$

- Weighted sum of all moments ($t$ controls the weights) and choosing $t$ appropriately lets one prove a number of very powerful exponential concentration bounds such as Chernoff, Bernstein, Hoeffding, Azuma, Berry-Esseen, etc.

**Bernstein Inequality:** Consider independent random variables $\mathbf{X}_1, \ldots, \mathbf{X}_n$ all falling in $[-M, M]$. Let $\mu = \mathbb{E}[\sum_{i=1}^n \mathbf{X}_i]$ and $\sigma^2 = \mathrm{Var}[\sum_{i=1}^n \mathbf{X}_i] = \sum_{i=1}^n \mathrm{Var}[\mathbf{X}_i]$. For any $t \geq 0$:

$$\Pr\left(\left|\sum_{i=1}^n \mathbf{X}_i - \mu\right| \geq t\right) \leq 2\exp\left(-\frac{t^2}{2\sigma^2 + \frac{4}{3}Mt}\right).$$

**Bernstein Inequality:** Consider independent random variables $\mathbf{X}_1, \ldots, \mathbf{X}_n$ all falling in $[-M, M]$. Let $\mu = \mathbb{E}[\sum_{i=1}^n \mathbf{X}_i]$ and $\sigma^2 = \mathrm{Var}[\sum_{i=1}^n \mathbf{X}_i] = \sum_{i=1}^n \mathrm{Var}[\mathbf{X}_i]$. For any $t \geq 0$:

$$\Pr\left(\left|\sum_{i=1}^n \mathbf{X}_i - \mu\right| \geq t\right) \leq 2\exp\left(-\frac{t^2}{2\sigma^2 + \frac{4}{3}Mt}\right).$$

Assume that $M = 1$ and plug in $t = s \cdot \sigma$ for $s \leq \sigma$.

**Bernstein Inequality:** Consider independent random variables $\mathbf{X}_1, \ldots, \mathbf{X}_n$ all falling in [-1,1]. Let $\mu = \mathbb{E}[\sum_{i=1}^n \mathbf{X}_i]$ and $\sigma^2 = \mathrm{Var}[\sum_{i=1}^n \mathbf{X}_i] = \sum_{i=1}^n \mathrm{Var}[\mathbf{X}_i]$. For any $s \geq 0$:

$$\Pr\left(\left|\sum_{i=1}^n \mathbf{X}_i - \mu\right| \geq s\sigma\right) \leq 2\exp\left(-\frac{s^2}{4}\right).$$

Assume that $M = 1$ and plug in $t = s \cdot \sigma$ for $s \leq \sigma$.

**Bernstein Inequality:** Consider independent random variables $\mathbf{X}_1, \ldots, \mathbf{X}_n$ all falling in [-1,1]. Let $\mu = \mathbb{E}[\sum_{i=1}^n \mathbf{X}_i]$ and $\sigma^2 = \mathrm{Var}[\sum_{i=1}^n \mathbf{X}_i] = \sum_{i=1}^n \mathrm{Var}[\mathbf{X}_i]$. For any $s \geq 0$:

$$\Pr\left(\left|\sum_{i=1}^n \mathbf{X}_i - \mu\right| \geq s\sigma\right) \leq 2\exp\left(-\frac{s^2}{4}\right).$$

Assume that $M = 1$ and plug in $t = s \cdot \sigma$ for $s \leq \sigma$.

**Compare to Chebyshev's:** $\Pr\left(\left|\sum_{i=1}^n \mathbf{X}_i - \mu\right| \geq s\sigma\right) \leq \frac{1}{s^2}$.

**Bernstein Inequality:** Consider independent random variables $\mathbf{X}_1, \ldots, \mathbf{X}_n$ all falling in [-1,1]. Let $\mu = \mathbb{E}[\sum_{i=1}^n \mathbf{X}_i]$ and $\sigma^2 = \text{Var}[\sum_{i=1}^n \mathbf{X}_i] = \sum_{i=1}^n \text{Var}[\mathbf{X}_i]$. For any $s \geq 0$:

$$\Pr\left(\left|\sum_{i=1}^n \mathbf{X}_i - \mu\right| \geq s\sigma\right) \leq 2\exp\left(-\frac{s^2}{4}\right).$$

Assume that $M = 1$ and plug in $t = s \cdot \sigma$ for $s \leq \sigma$.

**Compare to Chebyshev's:** $\Pr\left(\left|\sum_{i=1}^n \mathbf{X}_i - \mu\right| \geq s\sigma\right) \leq \frac{1}{s^2}$.

- An exponentially stronger dependence on $s$!

Consider again bounding the number of heads $\mathbf{H}$ in $n = 100$ independent coin flips.

| **Chebyshev's:** | **Bernstein:** | **In Reality:** |
| --- | --- | --- |
| $\Pr(\mathbf{H} \geq 60) \leq .25$ | $\Pr(\mathbf{H} \geq 60) \leq .15$ | $\Pr(\mathbf{H} \geq 60) = 0.0284$ |
| $\Pr(\mathbf{H} \geq 70) \leq .0625$ | $\Pr(\mathbf{H} \geq 70) \leq .00086$ | $\Pr(\mathbf{H} \geq 70) = .000039$ |
| $\Pr(\mathbf{H} \geq 80) \leq .04$ | $\Pr(\mathbf{H} \geq 80) \leq 3^{-7}$ | $\Pr(\mathbf{H} \geq 80) < 10^{-9}$ |

$\mathbf{H}$: total number heads in 100 random coin flips. $\mathbb{E}[\mathbf{H}] = 50$.

Consider again bounding the number of heads $\mathbf{H}$ in $n = 100$ independent coin flips.

| **Chebyshev's:** | **Bernstein:** | **In Reality:** |
|---|---|---|
| $\Pr(\mathbf{H} \geq 60) \leq .25$ | $\Pr(\mathbf{H} \geq 60) \leq .15$ | $\Pr(\mathbf{H} \geq 60) = 0.0284$ |
| $\Pr(\mathbf{H} \geq 70) \leq .0625$ | $\Pr(\mathbf{H} \geq 70) \leq .00086$ | $\Pr(\mathbf{H} \geq 70) = .000039$ |
| $\Pr(\mathbf{H} \geq 80) \leq .04$ | $\Pr(\mathbf{H} \geq 80) \leq 3^{-7}$ | $\Pr(\mathbf{H} \geq 80) < 10^{-9}$ |

Getting much closer to the true probability.

$\mathbf{H}$: total number heads in 100 random coin flips. $\mathbb{E}[\mathbf{H}] = 50$.

Want to store a set $S$ of items from a massive universe of possible items (e.g., images, text documents, IP addresses).

Want to store a set $S$ of items from a massive universe of possible items (e.g., images, text documents, IP addresses).

**Goal:** support *insert*$(x)$ to add $x$ to the set and *query*$(x)$ to check if $x$ is in the set. Both in $O(1)$ time.

Want to store a set $S$ of items from a massive universe of possible items (e.g., images, text documents, IP addresses).

**Goal:** support $\mathit{insert}(x)$ to add $x$ to the set and $\mathit{query}(x)$ to check if $x$ is in the set. Both in $O(1)$ time.

Want to store a set $S$ of items from a massive universe of possible items (e.g., images, text documents, IP addresses).

**Goal:** support *insert*$(x)$ to add $x$ to the set and *query*$(x)$ to check if $x$ is in the set. Both in $O(1)$ time.

- Allow small probability $\delta > 0$ of false positives. I.e., for any $x$,

$$\Pr(\mathit{query}(x) = 1 \text{ and } x \notin S) \leq \delta.$$

Want to store a set $S$ of items from a massive universe of possible items (e.g., images, text documents, IP addresses).

**Goal:** support *insert*$(x)$ to add $x$ to the set and *query*$(x)$ to check if $x$ is in the set. Both in $O(1)$ time.

- Allow small probability $\delta > 0$ of false positives. I.e., for any $x$,

$$\Pr(\textit{query}(x) = 1 \text{ and } x \notin S) \leq \delta.$$

**Solution:** Bloom filters (repeated random hashing). Will use much less space than a hash table.

Chose $k$ independent random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \to [m]$.

Chose $k$ independent random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \rightarrow [m]$.

- Maintain an array $A$ containing $m$ bits, all initially 0.

Chose $k$ independent random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \to [m]$.

- Maintain an array $A$ containing $m$ bits, all initially 0.
- *insert*$(x)$: set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.

Chose $k$ independent random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \to [m]$.

- Maintain an array $A$ containing $m$ bits, all initially 0.
- *insert*$(x)$: set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*$(x)$: return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.

Chose $k$ independent random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \to [m]$.

- Maintain an array $A$ containing $m$ bits, all initially 0.
- *insert*$(x)$: set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*$(x)$: return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| m bit array **A** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# BLOOM FILTERS

Chose $k$ independent random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \to [m]$.

- Maintain an array $A$ containing $m$ bits, all initially 0.
- *insert*$(x)$: set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*$(x)$: return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.

**Insertions:**

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| m bit array **A** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Queries:**

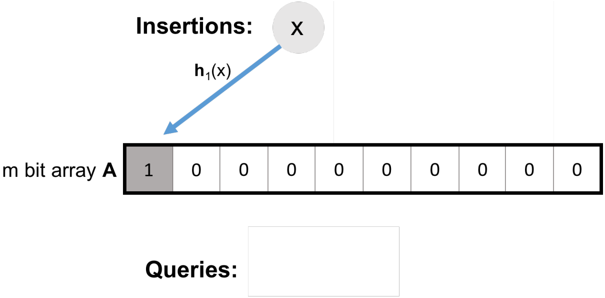
# BLOOM FILTERS

Chose $k$ independent random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \rightarrow [m]$.

- Maintain an array $A$ containing $m$ bits, all initially 0.
- *insert*$(x)$: set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*$(x)$: return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.

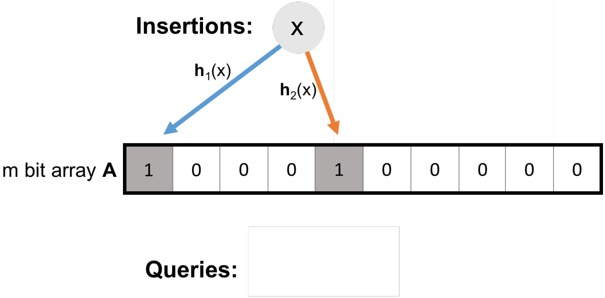
Chose $k$ independent random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \to [m]$.

- Maintain an array $A$ containing $m$ bits, all initially 0.
- *insert*$(x)$: set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*$(x)$: return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.



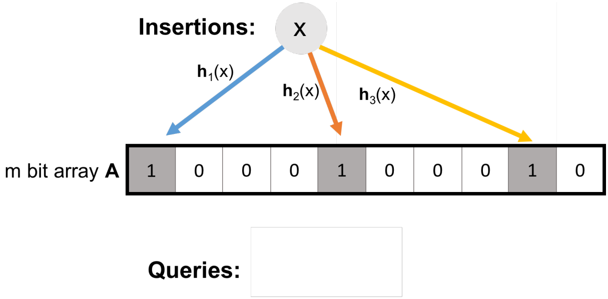
Chose $k$ independent random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \to [m]$.

- Maintain an array $A$ containing $m$ bits, all initially 0.
- *insert*$(x)$: set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*$(x)$: return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.



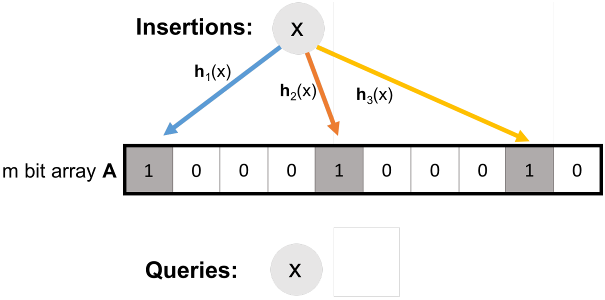
Chose $k$ independent random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \to [m]$.

- Maintain an array $A$ containing $m$ bits, all initially 0.
- *insert*$(x)$: set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*$(x)$: return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.



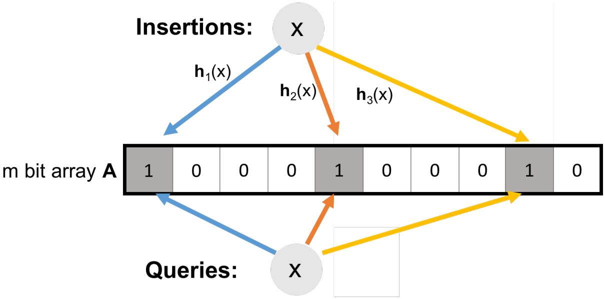
Chose $k$ independent random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \rightarrow [m]$.

- Maintain an array $A$ containing $m$ bits, all initially 0.
- *insert*$(x)$: set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*$(x)$: return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.



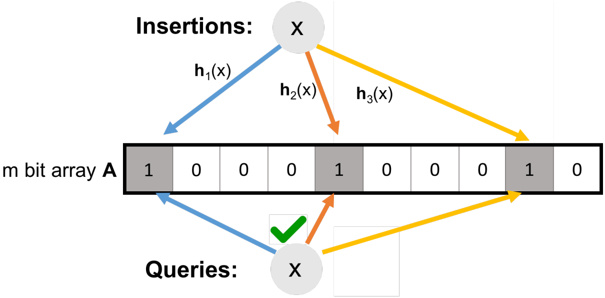
Chose $k$ independent random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \rightarrow [m]$.

- Maintain an array $A$ containing $m$ bits, all initially 0.
- *insert*$(x)$: set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*$(x)$: return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.



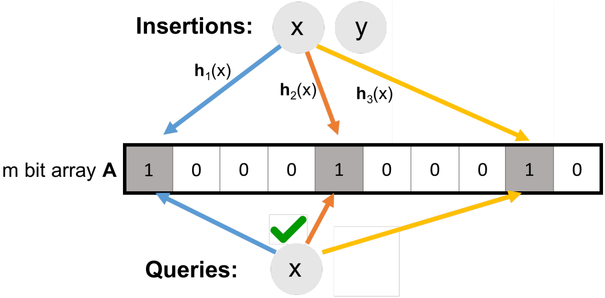
Chose $k$ independent random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \rightarrow [m]$.

- Maintain an array $A$ containing $m$ bits, all initially 0.
- $insert(x)$: set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- $query(x)$: return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.



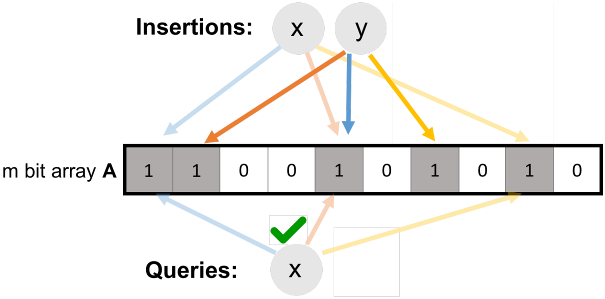
Chose $k$ independent random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \rightarrow [m]$.

- Maintain an array $A$ containing $m$ bits, all initially 0.
- *insert*$(x)$: set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*$(x)$: return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.



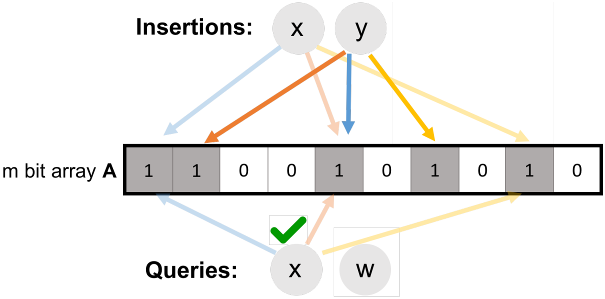
Chose $k$ independent random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \to [m]$.

- Maintain an array $A$ containing $m$ bits, all initially 0.
- $insert(x)$: set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- $query(x)$: return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.



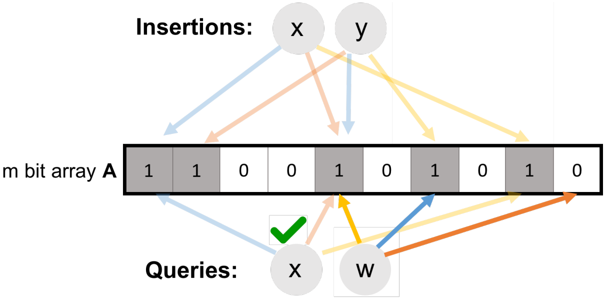
Chose $k$ independent random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \to [m]$.

- Maintain an array $A$ containing $m$ bits, all initially 0.
- *insert*$(x)$: set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*$(x)$: return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.



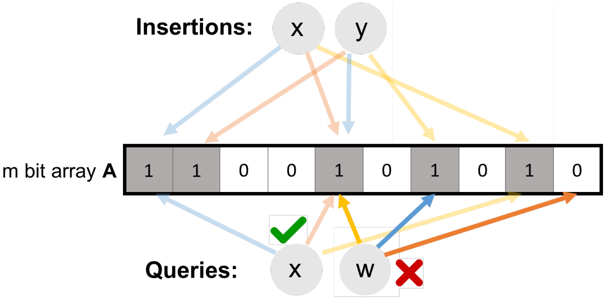
Chose $k$ independent random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \to [m]$.

- Maintain an array $A$ containing $m$ bits, all initially 0.
- *insert*$(x)$: set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*$(x)$: return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.



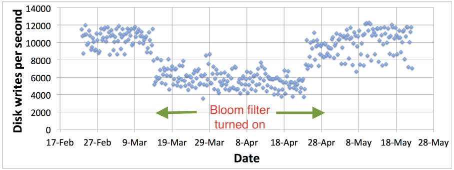
Chose $k$ independent random hash functions $\mathbf{h}_1, \ldots, \mathbf{h}_k$ mapping the universe of elements $U \to [m]$.

- Maintain an array $A$ containing $m$ bits, all initially 0.
- *insert*$(x)$: set all bits $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] := 1$.
- *query*$(x)$: return 1 only if $A[\mathbf{h}_1(x)] = \ldots = A[\mathbf{h}_k(x)] = 1$.

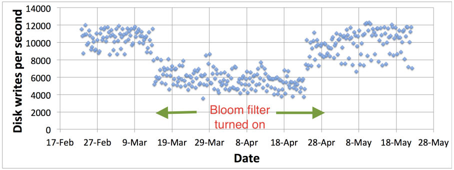



No false negatives. False positives more likely with more insertions.

Akamai (Boston-based company serving 15 – 30% of all web traffic) applies bloom filters to prevent caching of 'one-hit-wonders' – pages only visited once fill over 75% of cache.

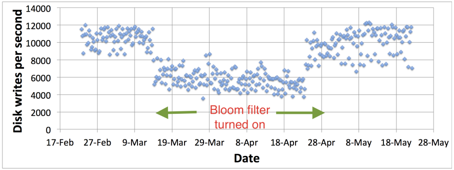


Akamai (Boston-based company serving 15 – 30% of all web traffic) applies bloom filters to prevent caching of 'one-hit-wonders' – pages only visited once fill over 75% of cache.




- When url $x$ comes in, if $query(x) = 1$, cache the page at $x$. If not, run $insert(x)$ so that if it comes in again, it will be cached.

Akamai (Boston-based company serving 15 – 30% of all web traffic) applies bloom filters to prevent caching of 'one-hit-wonders' – pages only visited once fill over 75% of cache.




- When url $x$ comes in, if $query(x) = 1$, cache the page at $x$. If not, run $insert(x)$ so that if it comes in again, it will be cached.
- **False positive:** A new url (possible one-hit-wonder) is cached. If the bloom filter has a false positive rate of $\delta = .05$, the number of cached one-hit-wonders will be reduced by at least 95%.

For a bloom filter with $m$ bits and $k$ hash functions, the insertion and query time is $O(k)$.

For a bloom filter with $m$ bits and $k$ hash functions, the insertion and query time is $O(k)$. How does the false positive rate $\delta$ depend on $m$, $k$, and the number of items inserted?

For a bloom filter with $m$ bits and $k$ hash functions, the insertion and query time is $O(k)$. How does the false positive rate $\delta$ depend on $m$, $k$, and the number of items inserted?

**Step 1**: What is the probability that after inserting $n$ elements, the $i^{th}$ bit of the array $A$ is still 0?

For a bloom filter with $m$ bits and $k$ hash functions, the insertion and query time is $O(k)$. How does the false positive rate $\delta$ depend on $m$, $k$, and the number of items inserted?

**Step 1**: What is the probability that after inserting $n$ elements, the $i^{th}$ bit of the array $A$ is still 0? $n \times k$ total hashes must not hit bit $i$.

$$\Pr(A[i] = 0) = \Pr\big(\mathbf{h}_1(x_1) \neq i \cap \ldots \cap \mathbf{h}_k(x_1) \neq i \\ \cap\, \mathbf{h}_1(x_2) \neq i \ldots \cap \mathbf{h}_k(x_2) \neq i \cap \ldots\big)$$

For a bloom filter with $m$ bits and $k$ hash functions, the insertion and query time is $O(k)$. How does the false positive rate $\delta$ depend on $m$, $k$, and the number of items inserted?

**Step 1**: What is the probability that after inserting $n$ elements, the $i^{th}$ bit of the array $A$ is still 0? $n \times k$ total hashes must not hit bit $i$.

$$\begin{aligned}\Pr(A[i] = 0) &= \Pr\big(\mathbf{h}_1(x_1) \neq i \cap \ldots \cap \mathbf{h}_k(x_1) \neq i \\ &\qquad \cap \mathbf{h}_1(x_2) \neq i \ldots \cap \mathbf{h}_k(x_2) \neq i \cap \ldots\big) \\ &= \underbrace{\Pr\big(\mathbf{h}_1(x_1) \neq i\big) \times \ldots \times \Pr\big(\mathbf{h}_k(x_1) \neq i\big) \times \Pr\big(\mathbf{h}_1(x_2) \neq i\big) \ldots}_{k \cdot n \text{ events each occuring with probability } 1-1/m}\end{aligned}$$

# ANALYSIS

For a bloom filter with $m$ bits and $k$ hash functions, the insertion and query time is $O(k)$. How does the false positive rate $\delta$ depend on $m$, $k$, and the number of items inserted?

**Step 1**: What is the probability that after inserting $n$ elements, the $i^{th}$ bit of the array $A$ is still 0? $n \times k$ total hashes must not hit bit $i$.

$$\begin{aligned}
\Pr(A[i] = 0) &= \Pr\big(\mathbf{h}_1(x_1) \neq i \cap \ldots \cap \mathbf{h}_k(x_1) \neq i \\
&\qquad \cap\, \mathbf{h}_1(x_2) \neq i \ldots \cap \mathbf{h}_k(x_2) \neq i \cap \ldots\big) \\
&= \underbrace{\Pr\big(\mathbf{h}_1(x_1) \neq i\big) \times \ldots \times \Pr\big(\mathbf{h}_k(x_1) \neq i\big) \times \Pr\big(\mathbf{h}_1(x_2) \neq i\big) \ldots}_{k \cdot n \text{ events each occuring with probability } 1 - 1/m} \\
&= \left(1 - \frac{1}{m}\right)^{kn}
\end{aligned}$$

How does the false positive rate $\delta$ depend on $m$, $k$, and the number of items inserted?

What is the probability that after inserting $n$ elements, the $i^{th}$ bit of the array $A$ is still 0?

$$\Pr(A[i] = 0) = \left(1 - \frac{1}{m}\right)^{kn}$$

$n$: total number items in filter, $m$: number of bits in filter, $k$: number of random hash functions, $\mathbf{h}_1, \ldots \mathbf{h}_k$: hash functions, $A$: bit array, $\delta$: false positive rate.

How does the false positive rate $\delta$ depend on $m$, $k$, and the number of items inserted?

What is the probability that after inserting $n$ elements, the $i^{th}$ bit of the array $A$ is still 0?

$$\Pr(A[i] = 0) = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-\frac{kn}{m}}$$

$n$: total number items in filter, $m$: number of bits in filter, $k$: number of random hash functions, $\mathbf{h}_1, \ldots \mathbf{h}_k$: hash functions, $A$: bit array, $\delta$: false positive rate.

How does the false positive rate $\delta$ depend on $m$, $k$, and the number of items inserted?

What is the probability that after inserting $n$ elements, the $i^{th}$ bit of the array $A$ is still 0?

$$\Pr(A[i] = 0) = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-\frac{kn}{m}}$$

Let $T$ be the number of zeros in the array after $n$ inserts. Then,

$$E[T] = m\left(1 - \frac{1}{m}\right)^{kn} \approx me^{-\frac{kn}{m}}$$

$n$: total number items in filter, $m$: number of bits in filter, $k$: number of random hash functions, $\mathbf{h}_1, \ldots \mathbf{h}_k$: hash functions, $A$: bit array, $\delta$: false positive rate.

# CORRECT ANALYSIS SKETCH

If $T$ is the number of 0 entries, for a non-inserted element $w$:

$$\begin{aligned}
&\Pr(A[\mathbf{h}_1(w)] = \ldots = A[\mathbf{h}_k(w)] = 1) \\
&= \Pr(A[\mathbf{h}_1(w)] = 1) \times \ldots \times \Pr(A[\mathbf{h}_k(w)] = 1) \\
&= (1 - T/m) \times \ldots \times (1 - T/m) \\
&= (1 - T/m)^k
\end{aligned}$$

# CORRECT ANALYSIS SKETCH

If $T$ is the number of 0 entries, for a non-inserted element $w$:

$$\begin{aligned}
&\Pr(A[\mathbf{h}_1(w)] = \ldots = A[\mathbf{h}_k(w)] = 1)\\
&= \Pr(A[\mathbf{h}_1(w)] = 1) \times \ldots \times \Pr(A[\mathbf{h}_k(w)] = 1)\\
&= (1 - T/m) \times \ldots \times (1 - T/m)\\
&= (1 - T/m)^k
\end{aligned}$$

- How small is $T/m$? Note that $\frac{T}{m} \geq \frac{m-nk}{m} \approx e^{-\frac{kn}{m}}$ when $kn \ll m$. More generally, it can be shown that $T/m = \Omega\left(e^{-\frac{kn}{m}}\right)$ via Theorem 2 of:

  `cglab.ca/~morin/publications/ds/bloom-submitted.pdf`

# FALSE POSITIVE RATE

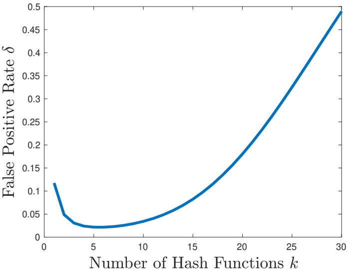
**False Positive Rate:** with $m$ bits of storage, $k$ hash functions, and $n$ items inserted $\delta \approx \left(1 - e^{\frac{-kn}{m}}\right)^k$.

# FALSE POSITIVE RATE

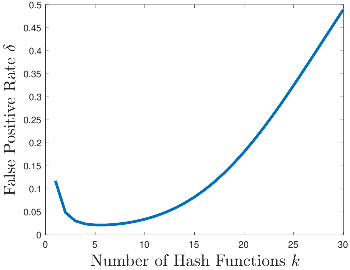
**False Positive Rate:** with $m$ bits of storage, $k$ hash functions, and $n$ items inserted $\delta \approx \left(1 - e^{\frac{-kn}{m}}\right)^k$.

**False Positive Rate:** with $m$ bits of storage, $k$ hash functions, and $n$ items inserted $\delta \approx \left(1 - e^{\frac{-kn}{m}}\right)^k$.

**False Positive Rate:** with $m$ bits of storage, $k$ hash functions, and $n$ items inserted $\delta \approx \left(1 - e^{\frac{-kn}{m}}\right)^k$.



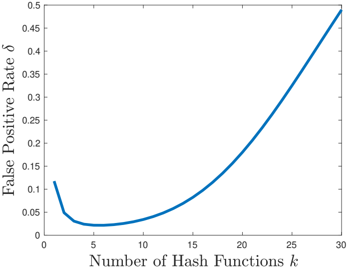
**False Positive Rate:** with $m$ bits of storage, $k$ hash functions, and $n$ items inserted $\delta \approx \left(1 - e^{\frac{-kn}{m}}\right)^k$.




- Can differentiate to show optimal number of hashes is $k = \ln 2 \cdot \frac{m}{n}$.

# FALSE POSITIVE RATE

**False Positive Rate:** with $m$ bits of storage, $k$ hash functions, and $n$ items inserted $\delta \approx \left(1 - e^{\frac{-kn}{m}}\right)^k$.




- Can differentiate to show optimal number of hashes is $k = \ln 2 \cdot \frac{m}{n}$.
- Balances between filling up the array with too many hashes and having enough hashes so that even when the array is pretty full, a new item is unlikely to have all its bits set (yield a false positive)

**Stream Processing:** Have a massive dataset $X$ with $n$ items $x_1, x_2, \ldots, x_n$ that arrive in a continuous stream. Not nearly enough space to store all the items (in a single location).

- Still want to analyze and learn from this data.

**Stream Processing:** Have a massive dataset $X$ with $n$ items $x_1, x_2, \ldots, x_n$ that arrive in a continuous stream. Not nearly enough space to store all the items (in a single location).

- Still want to analyze and learn from this data.
- Typically must compress the data on the fly, storing a data structure from which you can still learn useful information.

**Stream Processing:** Have a massive dataset $X$ with $n$ items $x_1, x_2, \ldots, x_n$ that arrive in a continuous stream. Not nearly enough space to store all the items (in a single location).

- Still want to analyze and learn from this data.
- Typically must compress the data on the fly, storing a data structure from which you can still learn useful information.
- Often the compression is randomized. E.g., bloom filters.

**Stream Processing:** Have a massive dataset $X$ with $n$ items $x_1, x_2, \ldots, x_n$ that arrive in a continuous stream. Not nearly enough space to store all the items (in a single location).

- Still want to analyze and learn from this data.
- Typically must compress the data on the fly, storing a data structure from which you can still learn useful information.
- Often the compression is randomized. E.g., bloom filters.
- Compared to traditional algorithm design, which focuses on minimizing runtime, the big question here is how much space is needed to answer queries of interest.

# SOME EXAMPLES

- **Sensor data:** images from telescopes (15 terabytes per night from the Large Synoptic Survey Telescope), readings from seismometer arrays monitoring and predicting earthquake activity, traffic cameras and travel time sensors (Smart Cities), electrical grid monitoring.

- **Sensor data:** images from telescopes (15 terabytes per night from the Large Synoptic Survey Telescope), readings from seismometer arrays monitoring and predicting earthquake activity, traffic cameras and travel time sensors (Smart Cities), electrical grid monitoring.

# SOME EXAMPLES

- **Sensor data:** images from telescopes (15 terabytes per night from the Large Synoptic Survey Telescope), readings from seismometer arrays monitoring and predicting earthquake activity, traffic cameras and travel time sensors (Smart Cities), electrical grid monitoring.
- **Internet Traffic**: 500 million Tweets per day, 5.6 billion Google searches, billions of ad-clicks and other logs from instrumented webpages, IPs routed by network switches, ...

- **Sensor data:** images from telescopes (15 terabytes per night from the Large Synoptic Survey Telescope), readings from seismometer arrays monitoring and predicting earthquake activity, traffic cameras and travel time sensors (Smart Cities), electrical grid monitoring.
- **Internet Traffic**: 500 million Tweets per day, 5.6 billion Google searches, billions of ad-clicks and other logs from instrumented webpages, IPs routed by network switches, ...
- **Datasets in Machine Learning:** When training e.g. a neural network on a large dataset (ImageNet with 14 million images), the data is typically processed in a stream due to storage limitations.

# SOME EXAMPLES

- **Sensor data:** images from telescopes (15 terabytes per night from the Large Synoptic Survey Telescope), readings from seismometer arrays monitoring and predicting earthquake activity, traffic cameras and travel time sensors (Smart Cities), electrical grid monitoring.
- **Internet Traffic**: 500 million Tweets per day, 5.6 billion Google searches, billions of ad-clicks and other logs from instrumented webpages, IPs routed by network switches, ...
- **Datasets in Machine Learning:** When training e.g. a neural network on a large dataset (ImageNet with 14 million images), the data is typically processed in a stream due to storage limitations.

**Distinct Elements (Count-Distinct) Problem:** Given a stream $x_1, \ldots, x_n$, output the number of distinct elements in the stream.

**Distinct Elements (Count-Distinct) Problem:** Given a stream $x_1, \ldots, x_n$, output the number of distinct elements in the stream. E.g.,

$$1, 5, 7, 5, 2, 1 \rightarrow 4 \text{ distinct elements}$$

**Distinct Elements (Count-Distinct) Problem:** Given a stream $x_1, \ldots, x_n$, estimate the number of distinct elements in the stream. E.g.,

$$1, 5, 7, 5, 2, 1 \rightarrow 4 \text{ distinct elements}$$

**Distinct Elements (Count-Distinct) Problem:** Given a stream $x_1, \ldots, x_n$, estimate the number of distinct elements in the stream. E.g.,

$$1, 5, 7, 5, 2, 1 \to 4 \text{ distinct elements}$$

**Applications:**

- Distinct IP addresses clicking on an ad or visiting a site.
- Distinct values in a database column (for estimating sizes of joins and group bys).
- Number of distinct search engine queries.
- Counting distinct motifs in large DNA sequences.

**Distinct Elements (Count-Distinct) Problem:** Given a stream $x_1, \ldots, x_n$, estimate the number of distinct elements in the stream. E.g.,

$$1, 5, 7, 5, 2, 1 \rightarrow 4 \text{ distinct elements}$$

**Applications:**

- Distinct IP addresses clicking on an ad or visiting a site.
- Distinct values in a database column (for estimating sizes of joins and group bys).
- Number of distinct search engine queries.
- Counting distinct motifs in large DNA sequences.

Google Sawzall, Facebook Presto, Apache Drill, Twitter Algebird