CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Lossy Sets and Bloom Filters

Announcements

- Recitations are "open office hours" for the remainder of the semester
  - No attendance required, but TAs will still be there to answer questions and help review PA4/WA4/Final Exam material

Recall the `Set` ADT

Set has **`add`**, **`apply`** and **`remove`**...

What if we didn't need apply to be perfect?

What would that entail? Why would that be useful? What would we gain?

Motivation

Let's say a hot new restaurant just opened up, but it's an hour away. You want to eat there, but don't want to make the drive only to find out they have no tables…*what do you do?*

Motivation

Let's say a hot new restaurant just opened up, but it's an hour away. You want to eat there, but don't want to make the drive only to find out they have no tables…*what do you do?*

- **Call the restaurant**
  - **If they say they have no tables, don't go. You've saved yourself the trip**
  - **If they say they do…drive there. By the time you get there, they might have run out of tables, but you would have made the drive anyways**

Motivation

Another Example: Reading from disk is expensive

Even when using a **B+ Tree**, if you ask for an element that doesn't exist you will need to do $\mathbf{log_c}(\boldsymbol{n})$ disk reads

Idea: Keep an in-memory summary of the data

- If the summary says the key is in a particular layer, access the layer
- If not, skip the layer and end the search early

What guarantees do we need for this to work?

Motivation

- If our summary incorrectly says the key exists (false positive) ✓
  - We will read the layer only to find out it's not there
  - Extra work, but doesn't break anything…we would have done that work anyways

Motivation

- If our summary incorrectly says the key exists (false positive) ✓
  - We will read the layer only to find out it's not there
  - Extra work, but doesn't break anything...we would have done that work anyways
- If our summary incorrectly says the key doesn't exist (false negative) ✗
  - We will stop the algorithm and return an incorrect answer

Motivation

- If our summary incorrectly says the key exists (false positive) ✓
  - We will read the layer only to find out it's not there
  - Extra work, but doesn't break anything…we would have done that work anyways
- If our summary incorrectly says the key doesn't exist (false negative) ✗
  - We will stop the algorithm and return an incorrect answer

False positives are OK (not ideal though), false negatives are not

Lossy Sets

`LossySet[A]`

`add(a: A):` Insert a into the set

`apply(a: A):`

- If **a** is in the set **ALWAYS** return true
- If **a** is not in the set **USUALLY** return false (returning true is OK)

Lossy Set

What does this gain for us?

Idea: If apply doesn't always need to be right, we don't need to store everything

A Trivial Example

```
class TrivialLossySet[A] extends LossySet[A] {
  def add(a: A): Unit = { /* do nothing */ }
  def apply(a: A): Boolean = true
}
```

Does this work?

A Trivial Example

```scala
class TrivialLossySet[A] extends LossySet[A] {
  def add(a: A): Unit = { /* do nothing */ }
  def apply(a: A): Boolean = true
}
```

Does this work? **Yes…but not useful**

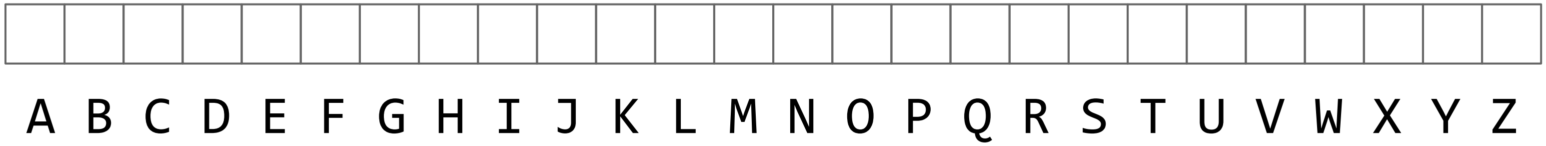
An Improvement

Idea: Take inspiration from HashTables

- Bucketize the keys in some way
- Keep **one bit** per bucket
- **`add(a: A)`:** Set **a**'s bucket to 1
- **`apply(a: A)`:** Return true if the bit for **a**'s bucket is set

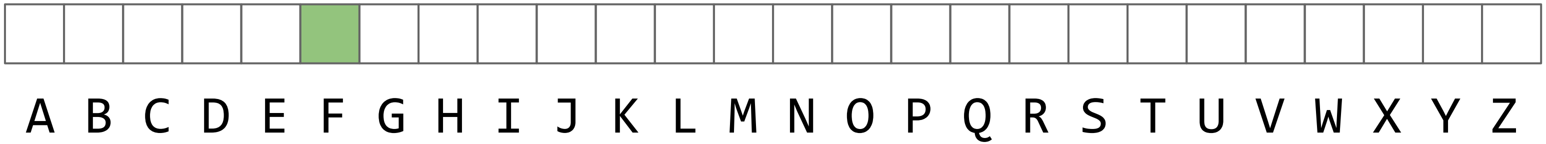
Setting Bins

```
add("Frankenstein")
```



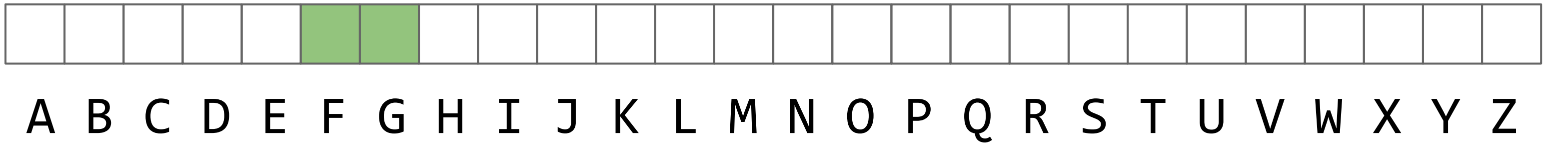
Setting Bins

```
add("Frankenstein")
```



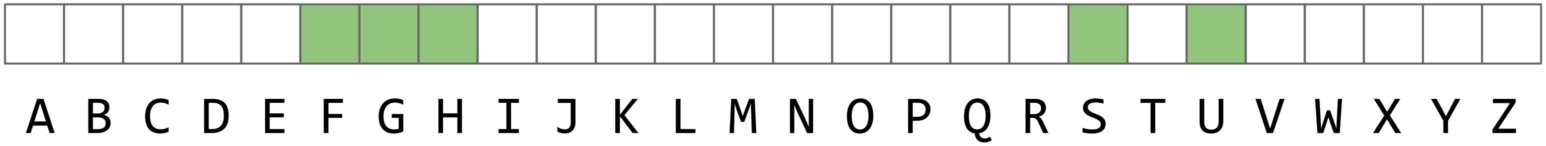
Setting Bins

```
add("Frankenstein")
add("Get Out")
```



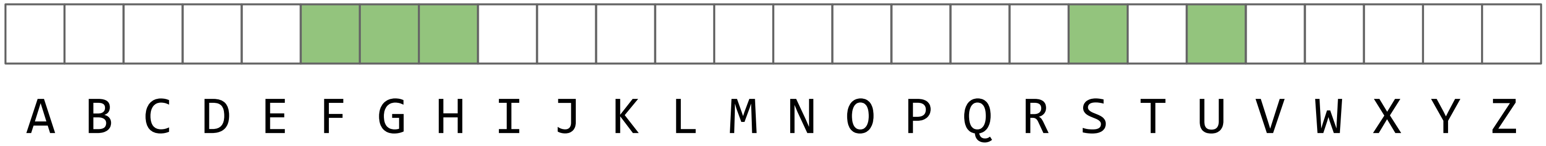
Setting Bins

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
```



Setting Bins

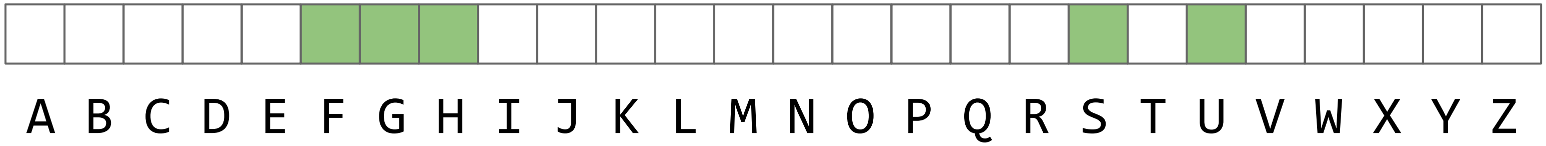
```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```



Calling Apply

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

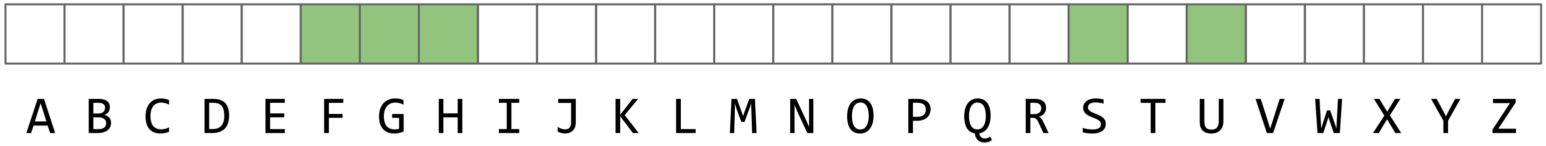
```
apply("Scream")?
```



Calling Apply

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

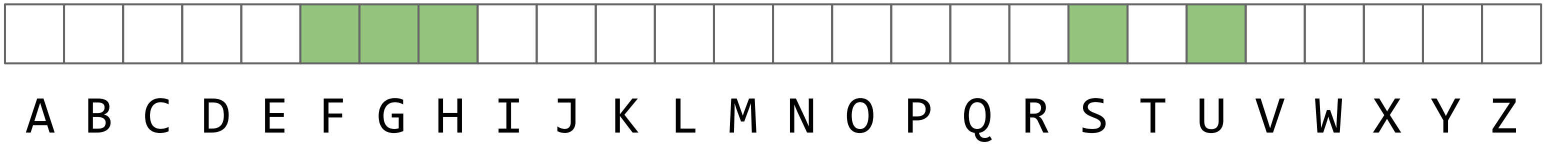
```
apply("Scream")? TRUE
```



Calling Apply

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

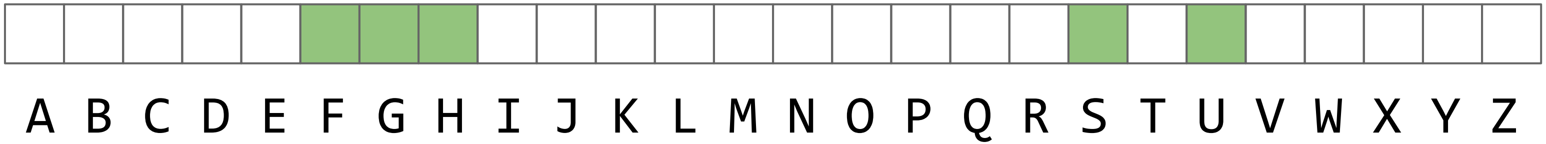
```
apply("Scream")? TRUE
apply("Saw")?
```



Calling Apply

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

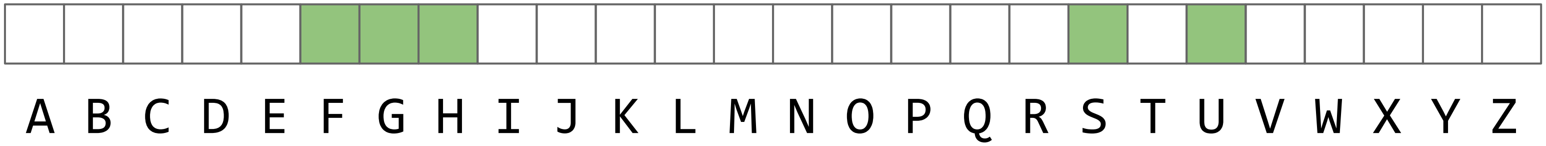
```
apply("Scream")? TRUE
apply("Saw")? TRUE
```



Calling Apply

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

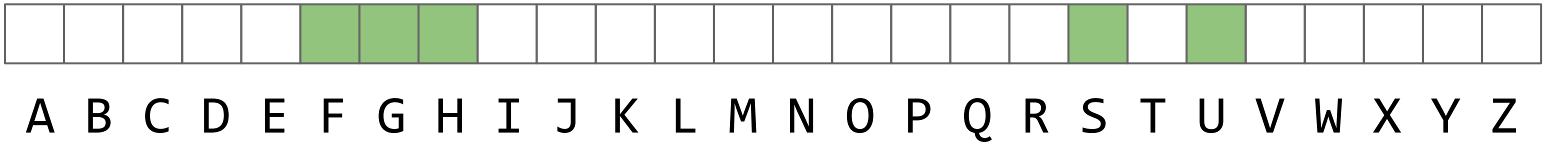
```
apply("Scream")? TRUE
apply("Saw")? TRUE
apply("The Candyman")?
```



Calling Apply

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

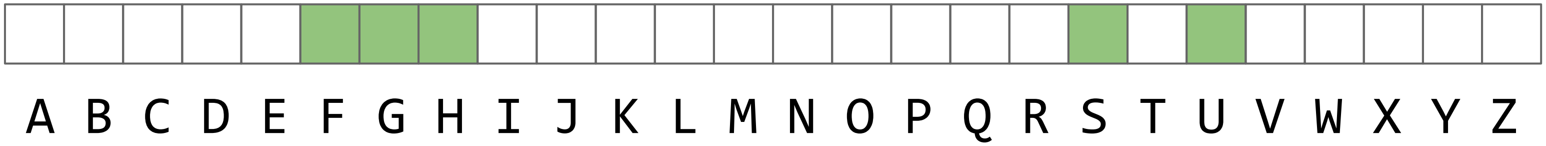
```
apply("Scream")? TRUE
apply("Saw")? TRUE
apply("The Candyman")? FALSE
```



Calling Apply

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

```
apply("Scream")? TRUE
apply("Saw")? TRUE
apply("The Candyman")? FALSE
apply("Dracula")? FALSE
apply("Friday the 13th")? TRUE
```

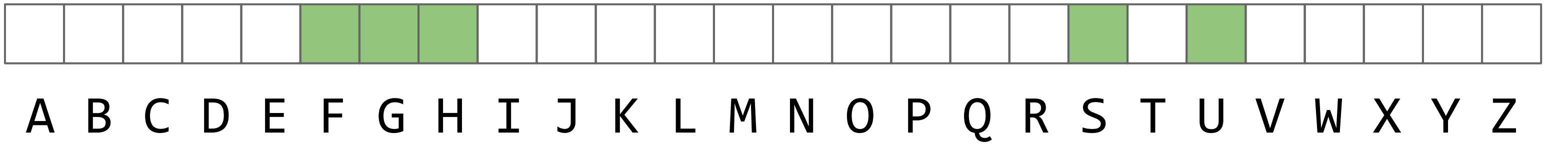


Calling Apply

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

```
apply("Scream")? TRUE
apply("Saw")? TRUE
apply("The Candyman")? FALSE
apply("Dracula")? FALSE
apply("Friday the 13th")? TRUE
```

So this works...how can we improve its accuracy?



Motivating Example

By show of hands, who was born in:

- ≤ 2000
- 2001
- 2002
- 2003
- 2004
- ≥ 2005

Take note of who raised their hand the same time as you

Motivating Example

By show of hands, what color is your shirt?

- White?
- Black?
- Red?
- Green?
- Blue?

Take note of who raised their hand the same time as you…how much overlap was there with the previous question?

Lossy Sets

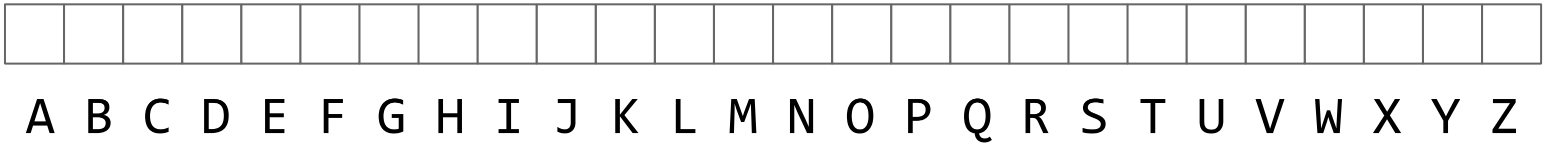
Observation: We have fewer collisions (less overlap) with TWO features instead of one…but now we need to store info for 2 features…? or do we?

Idea: Use the same set of buckets for both features

- ie store movies by first AND last letter in the title

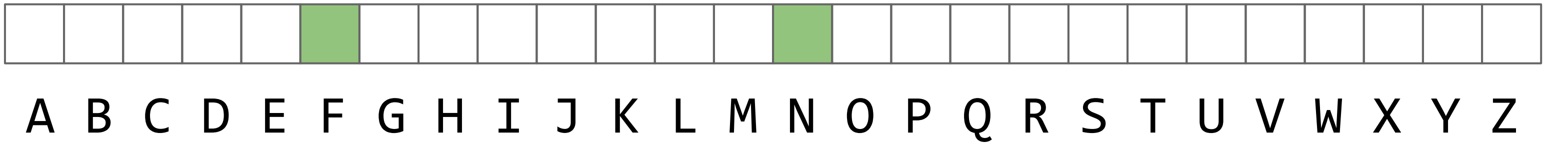
Setting Bins

```
add("Frankenstein")
```



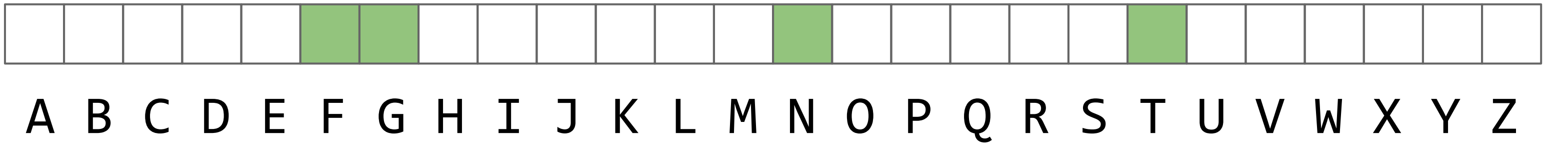
Setting Bins

```
add("Frankenstein")
```



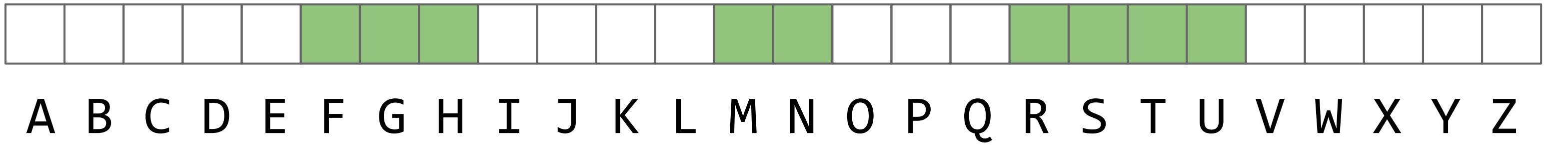
Setting Bins

```
add("Frankenstein")
add("Get Out")
```



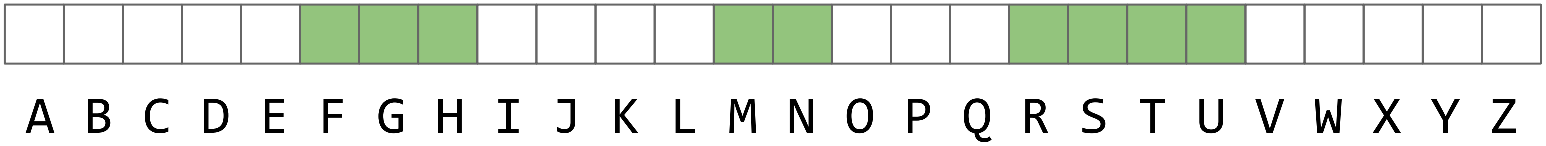
Setting Bins

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
```



Setting Bins

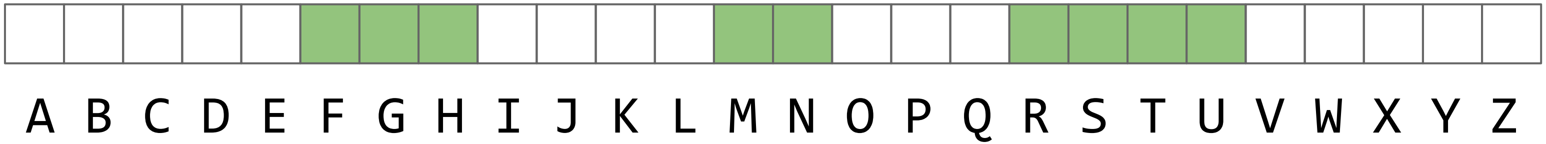
```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```



Calling Apply

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

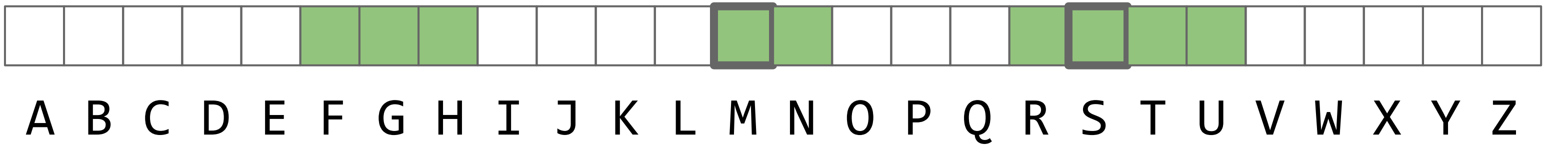
```
apply("Scream")?
```



Calling Apply

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

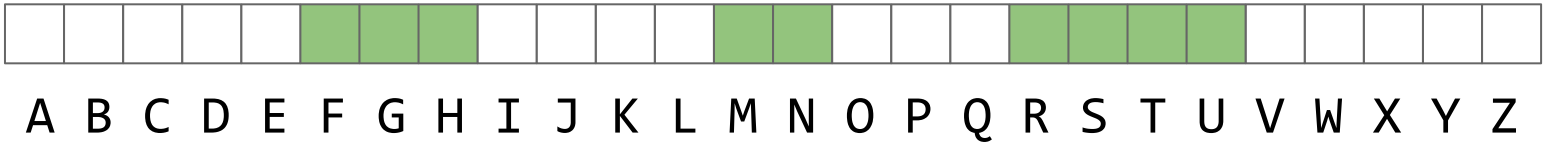
```
apply("Scream")? TRUE
```



Calling Apply

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

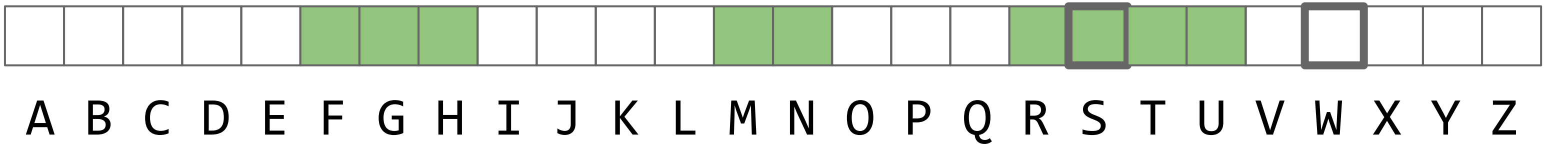
```
apply("Scream")? TRUE
apply("Saw")?
```



Calling Apply

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

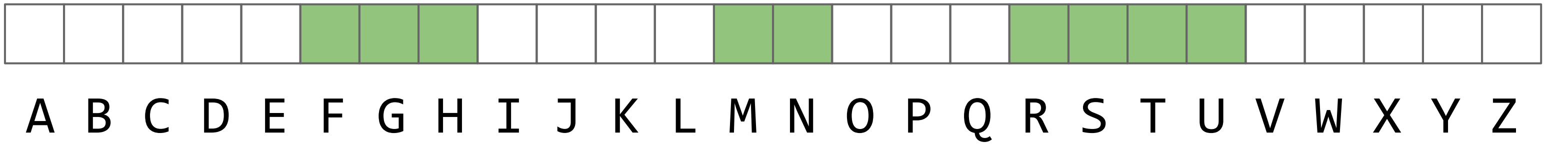
```
apply("Scream")? TRUE
apply("Saw")? FALSE
```



Calling Apply

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

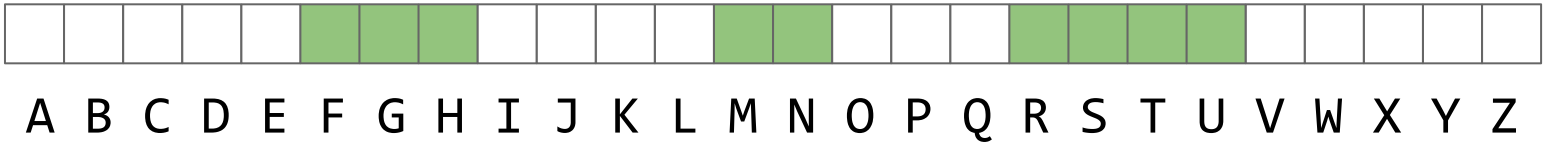
```
apply("Scream")? TRUE
apply("Saw")? TRUE
apply("Dracula")? FALSE
apply("Friday the 13th")? TRUE
```



Calling Apply

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

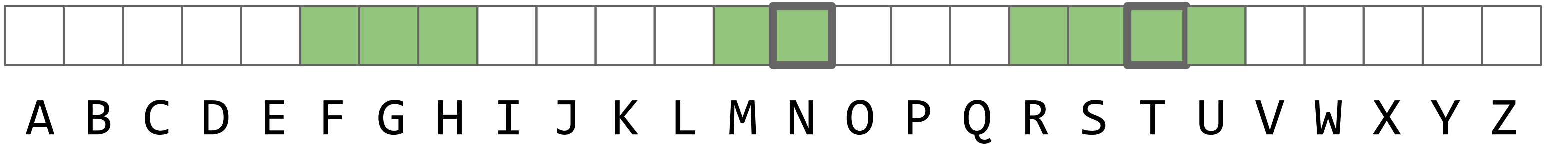
```
apply("Scream")? TRUE
apply("Saw")? TRUE
apply("Dracula")? FALSE
apply("Friday the 13th")? TRUE
apply("The Candyman")?
```



Calling Apply

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

```
apply("Scream")? TRUE
apply("Saw")? TRUE
apply("Dracula")? FALSE
apply("Friday the 13th")? TRUE
apply("The Candyman")? TRUE
```



Hash Functions

Observation: Our current example is not very uniform…movie titles are much more likely to start with some letters and not others

Idea: Use hash functions!

Lossy Hash Set

```
class LossyHashSet[A](_size: Int) extends LossySet[A] {
  val bits = new Array[Boolean](_size)
  def add(a: A): Unit = {
    val bucket = a.hashCode % _size
    bits(bucket) = true
  }
  def apply(a: A): Boolean = {
    val bucket = a.hashCode % _size
    return bits(bucket)
  }
}
```

Lossy Hash Set

Assume we **`add(a)`** then **`apply(b)`**

What does **`apply(b)`** return, and when?

- **True:** When **`hash(a) == hash(b) mod _size`**
- **False:** When **`hash(a) != hash(b) mod _size`**

What is the probability of each, with ***N*** buckets?

- **True:** 1/N ← only wrong 1/N of the time
- **False:** (N-1)/N

Lossy (Double) Hash Set

```
class LossyDoubleHashSet[A](_size: Int) extends LossySet[A] {
  val bits = new Array[Boolean](_size)
  def hash1(a: A): Int = ???
  def hash2(a: A): Int = ???
  def add(a: A): Unit = {
    bits( hash1(a) % _size ) = true
    bits( hash2(a) % _size ) = true
  }
  def apply(a: A): Boolean = {
    return bits( hash1(a) % _size ) && bits( hash2(a) % _size )
  }
}
```

Lossy (Double) Hash Set

Assume we **`add(a)`** then **`apply(b)`**

What does **`apply(b)`** return, and when?

- **True:** When $\texttt{hash}_1\texttt{(a) == hash}_1\texttt{(b)}$ AND $\texttt{hash}_2\texttt{(a) == hash}_2\texttt{(b)}$ (mod _size)
- **False:** Otherwise

What is the probability of each, with ***N*** buckets?

- **True:** ~$(1/N)^2$ ← only wrong $1/N^2$ of the time!
- **False:** $!((N-1)/N)^2$

How do we get 2 hash functions?

```
val SEED1 = 123104912035
val SEED2 = 406923456234

def hash1[A](a: A) =
  hash( SEED1 + a.hashCode )

def hash2[A](a: A) =
  hash( SEED2 + a.hashCode )
```

Don’t use sequentially adjacent values

How do we get 2 hash functions?

```
val SEED1 = 123104912035
val SEED2 = 406923456234

def hash1[A](a: A) =
  hash( SEED1 ^ a.hashCode )

def hash2[A](a: A) =
  hash( SEED2 ^ a.hashCode )
```

Use bitwise-XOR instead of +

How do we get K hash functions?

```
val SEED1 = 123104912035
def hash1[A](a: A) =
  hash( SEED1 ^ a.hashCode )

val SEED2 = 406923456234
def hash2[A](a: A) =
  hash( SEED2 ^ a.hashCode )

val SEED3 = 908057230543
def hash3[A](a: A) =
  hash( SEED3 ^ a.hashCode )
```

Generate as many hash functions as needed

How do we get K hash functions?

```
val SEEDS = Seq(123104912035, 406923456234, ...)
def ithHash[A](a: A, i: Int) =
  hash( SEEDS(i) ^ a.hashCode )
```

Bloom Filters

- Overall Structure
  - _**size** bits
  - _**k** hash functions

Bloom Filters

```
class BloomFilter[A](_size: Int, _k: Int) extends LossySet[A]
{
  val bits = new Array[Boolean](_size)

  def add(a: A): Unit = {
    for(i <- 0 until _k) { bits( ithHash(a, i) % _size ) = true }
  }

  def apply(a: A): Boolean = ???
}
```

Bloom Filters

```
class BloomFilter[A](_size: Int, _k: Int) extends LossySet[A]
{
  val bits = new Array[Boolean](_size)

  def add(a: A): Unit = {
    for(i <- 0 until _k) { bits( ithHash(a, i) % _size ) = true }
  }

  def apply(a: A): Boolean = {
    for(i <- 0 until _k) {
      if( !bits( ithHash(a, i) % _size ) { return false; }
    }
    return true
  }
}
```

Bloom Filters

```
class BloomFilter[A](_size: Int, _k: Int) extends LossySet[A]
{
  val bits = new Array[Boolean](_size)

  def add(a: A): Unit = {
    for(i <- 0 until _k) { bits( ithHash(a, i) % _size ) = true }
  }

  def apply(a: A): Boolean = {
    return (0 until _k).foreach { i => bits(ithHash(a, i) % _size) }
  }
}
```

Bloom Filter Parameters

- _size
  - Intuitively: More space, fewer collisions
- _k
  - Intuitively: more hash functions means…
    - …more chances for one of **b**’s bits to be unset.
    - …more bits set = higher chance of collisions.

Bloom Filters: Analysis

$$\frac{1}{N}$$

The probability that 1 bit is set by 1 hash function

Bloom Filters: Analysis

$$1 - \frac{1}{N}$$

The probability that 1 bit is **not** set by 1 hash function

Bloom Filters: Analysis

$$\left(1 - \frac{1}{N}\right)^k$$

The probability that 1 bit is **not** set by k hash functions

Bloom Filters: Analysis

$$\left(1 - \frac{1}{N}\right)^{kn}$$

The probability that 1 bit is **not** set by k hash functions

... over n distinct calls to **add**

Bloom Filters: Analysis

$$1 - \left(1 - \frac{1}{N}\right)^{kn}$$

The probability that 1 bit is set by **at least one** of k hash functions

... over n distinct calls to **add**

Bloom Filters: Analysis

$$\approx \left(1 - \left(1 - \frac{1}{N}\right)^{kn}\right)^{k}$$

The probability that all k randomly selected bits of element **b**

... are set by **<u>at least one</u>** of k hash functions

... over n distinct calls to **add**

Bloom Filters: Analysis

The chance of collision in a Bloom filter with parameters k, N after n distinct elements have been added

$$\approx \left(1 - e^{-\frac{kn}{N}}\right)^k$$

The probability that all k randomly selected bits of element **b**

... are set by **at least one** of k hash functions

... over n distinct calls to **add**

Bloom Filters: Analysis

$$\approx \left(1 - e^{-\frac{kn}{N}}\right)^k$$

As $e^{kn/N}$ grows, the chance of collision shrinks

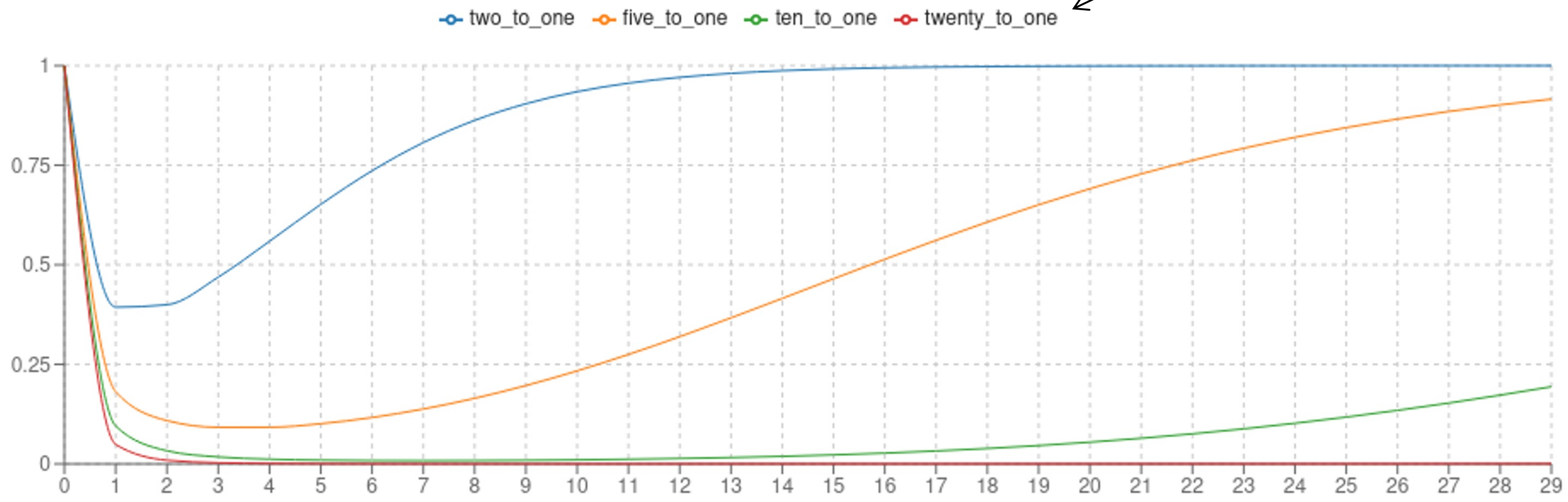
Bloom Filters: Analysis

$$\left(1 - e^{-\frac{kn}{N}}\right)^k$$

- **Ideal**: Pick N, k that minimize collision chance:
  - N
    - Smaller N, more opportunities for collisions
    - Bigger N, more space used
  - k
    - Smaller k, fewer tests, more chance of collisions
    - Bigger k, more bits set, more chance of collisions
    - Sweet spot in the middle

Bloom Filters: Analysis

$$\frac{N}{n}$$



Optimum at: $k = c \cdot \frac{N}{n}$

Bloom Filters: Analysis

$$k = c \cdot \frac{N}{n}$$

$$n = c\frac{N}{k}$$

N and n are linearly related
O(n) buckets required

Bloom Filters: Analysis

- N/n = 5 → ~10% collision chance
- N/n = 10 → ~1% collision chance

- 10 bits vs
  - 32 bits for one Int (3 to 1 savings)
  - 64 bits for a Double/Long (6 to 1 savings)
  - ~8000 bits for a full record (800 to 1 savings)

Bloom Filters: Analysis

- vs B+Tree or Binary Search Tree implementing Set
  - $O(k \cdot \textbf{cost}_{hash}) \approx O(1)$ **vs** $O(\log(n) \cdot \textbf{cost}_{compare})$ runtime
  - No directory pages (constant factor extra memory required)
- vs Hash Table implementing Set
  - Guaranteed $O(k \cdot \textbf{cost}_{hash}) \approx O(1)$ **vs** Expected $O(\textbf{cost}_{hash})$
  - No ‘fill factor’ (constant factor extra memory required)
- vs Array implementing Set
  - $O(k \cdot \textbf{cost}_{hash}) \approx O(1)$ **vs** $O(n \cdot \textbf{cost}_{compare})$