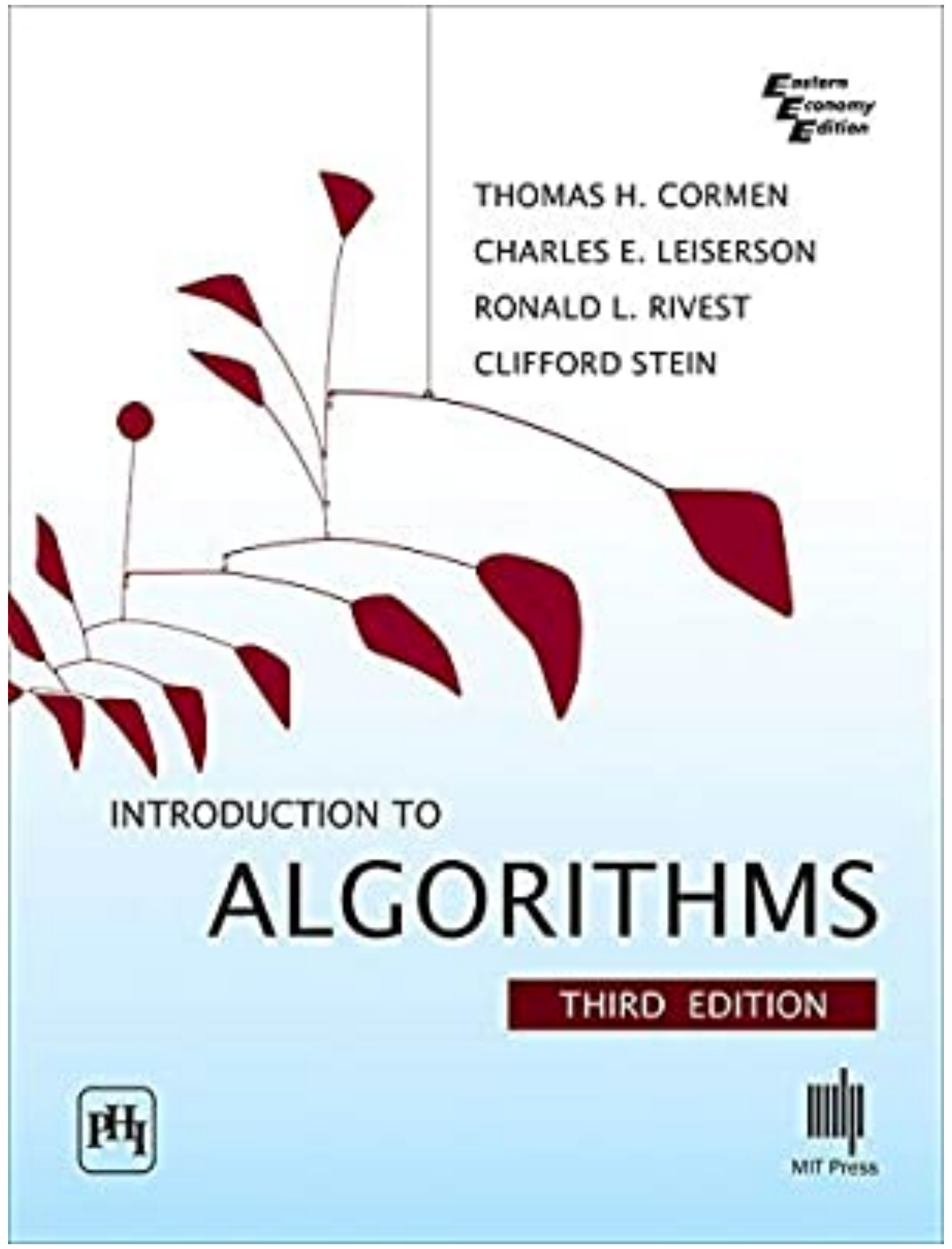
Lecture 10: Hashing
Bloom Filter

Readings



CLRS Chapter 11.3.3

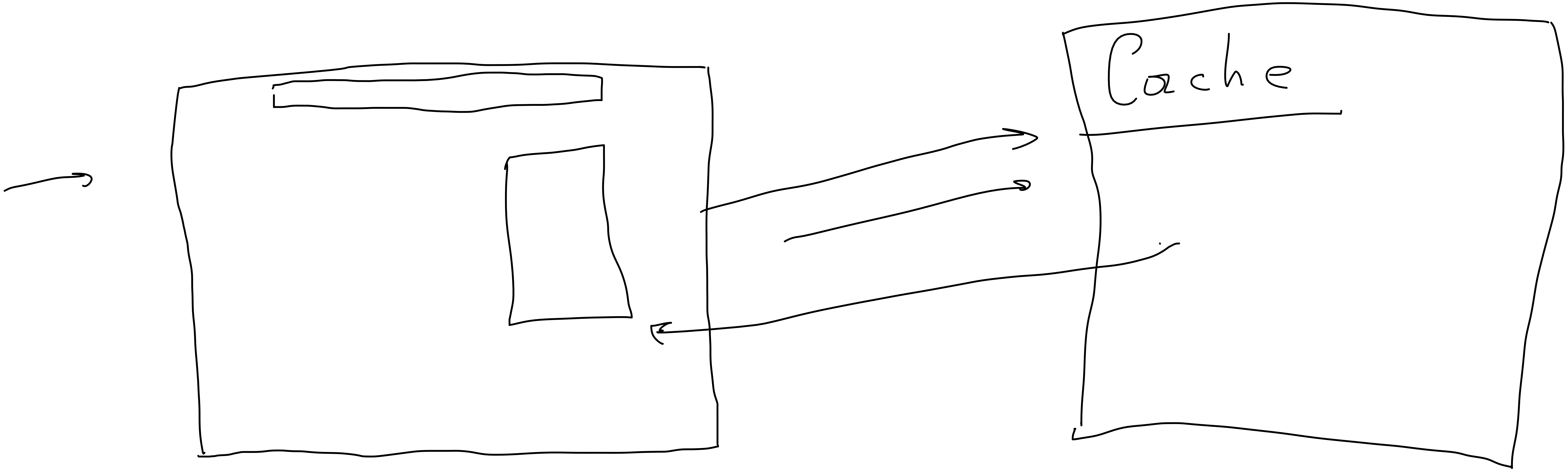
Introduction

- Imagine we would like to build a web cache application. We would like to store URLs in some space-efficient way such that we can check membership in the cache very efficiently.

- Ideally, we would like to use $O(n)$ space to store n keys (i.e. URLs) picked from a universe of size U, where U is much bigger than n, and would like to be able to check membership in the cache in time $O(1)$.

- These are all the operations we care about: that is, instead of supporting Insert, Delete, Find and Successor operations, we will just want to support Insert and Member.

- The data structure maintains a subset $S \subseteq U$ of keys. The operation Member(k) should just return whether or not the supplied key k is contained within S.



set → O(log n)

O(1)

Introduction

Bloom filters are a randomised data structure which achieve this goal. However, they have some important caveats:

- Bloom filters do not support deletion; they only support Insert and Member.
- They are not deterministic but have some risk of false positives.
- That is, when we query the Bloom filter with some key k, if $k \notin S$ there is some small chance (say 1%) that the answer is “yes” when it should be “no”. On the other hand, if $k \in S$ the answer is always “yes” .

This is reasonable for applications like a web cache:

- If we incorrectly think that a page is in the cache, this is not a disaster: we check the cache first, find it is not there, and download it directly.
- However, if we incorrectly decide that a page is not in the cache, this is undesirable because we download the page unnecessarily.

Example

The following sequence of operations illustrates what can happen using a Bloom filter.

Operation	Returns
Insert(`www.bbc.co.uk`)	

Example

The following sequence of operations illustrates what can happen using a Bloom filter.

Operation	Returns
Insert(`www.bbc.co.uk`)	
Insert(`twitter.com`)	

Example

The following sequence of operations illustrates what can happen using a Bloom filter.

Operation	Returns
Insert(`www.bbc.co.uk`)	
Insert(`twitter.com`)	
Member(`cs.bristol.ac.uk`)	No

Example

The following sequence of operations illustrates what can happen using a Bloom filter.

Operation	Returns
Insert(www.bbc.co.uk)	
Insert(twitter.com)	
Member(cs.bristol.ac.uk)	No
Member(www.bbc.co.uk)	Yes

Example

The following sequence of operations illustrates what can happen using a Bloom filter.

Operation	Returns
Insert(`www.bbc.co.uk`)	
Insert(`twitter.com`)	
Member(`cs.bristol.ac.uk`)	No
Member(`www.bbc.co.uk`)	Yes
Insert(`facebook.com`)	

Example

The following sequence of operations illustrates what can happen using a Bloom filter.

Operation	Returns
Insert(`www.bbc.co.uk`)	
Insert(`twitter.com`)	
Member(`cs.bristol.ac.uk`)	No
Member(`www.bbc.co.uk`)	Yes
Insert(`facebook.com`)	
Member(`cs.bristol.ac.uk`)	Yes

~1%

The last “Yes” is an example of a false positive.

How can we do it?
Your ideas…

Naive approach

- The simplest thing we could do to implement the web cache is to maintain a string B of U bits in an array, where bit $B[k]$ is set to 0 or 1 depending on whether $k \in S$.

- For example, if the universe is the integers between 1 and 10, after inserting 3, 6 and 8 we have:

len(B)

1	2	3	4	5	6	7	8	9	10
0	0	1	0	0	1	0	1	0	0

- If we would like the storage space used not to depend on U, we will need to compress this string somehow.

Insert("aaa")

Insert(3)

h("aaa") = x % 10

Yes ← Member("abc")

h("abc") = x

Member("aaa")

Hashing

- One way to do this is by hashing. We maintain an m-bit string B in our structure, for some m to be determined. Assume we have access to a hash function h which maps each key k to an integer $h(k)$ between 1 and m.

- Our structure will set bit number $h(k)$ of B to 1 when key k is inserted.

- If we would like the storage space used not to depend on U, we will need to compress this string somehow.

Example

Imagine m = 3 and we have h(www.bbc.co.uk) = 2,
h(facebook.com) = 3,
h(cs.bristol.ac.uk) = 3.

Start	0	0	0

Example

Imagine m = 3 and we have h(www.bbc.co.uk) = 2,
h(facebook.com) = 3,
h(cs.bristol.ac.uk) = 3.

Start	0	0	0
Insert(`www.bbc.co.uk`)	0	1	0

Example

Imagine m = 3 and we have h(www.bbc.co.uk) = 2,
h(facebook.com) = 3,
h(cs.bristol.ac.uk) = 3.

Start	0	0	0
Insert(`www.bbc.co.uk`)	0	1	0
Insert(`facebook.com`)	0	1	1

Example

Imagine m = 3 and we have h(www.bbc.co.uk) = 2,
h(facebook.com) = 3,
h(cs.bristol.ac.uk) = 3.

Start	0	0	0
Insert(`www.bbc.co.uk`)	0	1	0
Insert(`facebook.com`)	0	1	1
Member(`cs.bristol.ac.uk`) returns Yes	0	1	1

Hashing

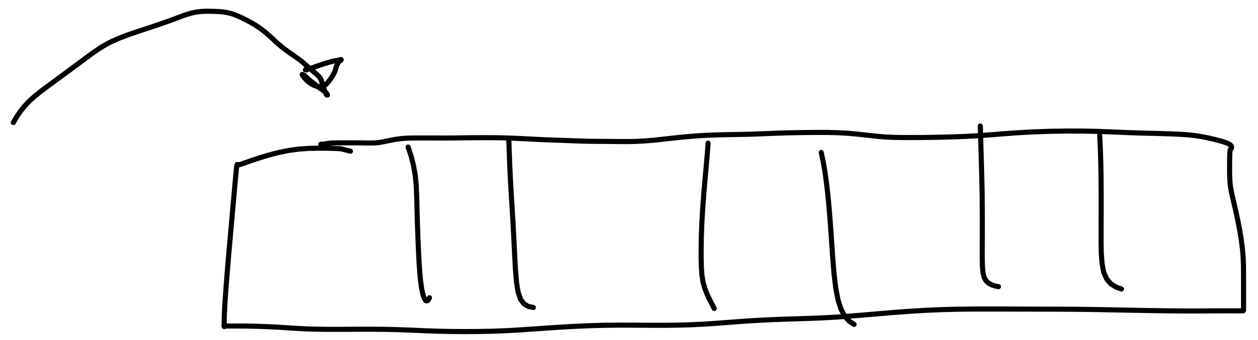
- A problem with this idea: if $m < U$, there will be some keys that hash to the same positions (collisions).

- If we call Member(k) for some k *not in* S , if $h(k) = h(k')$ for some $k' \in S$, we will incorrectly output “yes”.

- To make the probability of collisions low for the worst-case input, we pick our hash function $h(k)$ at random.

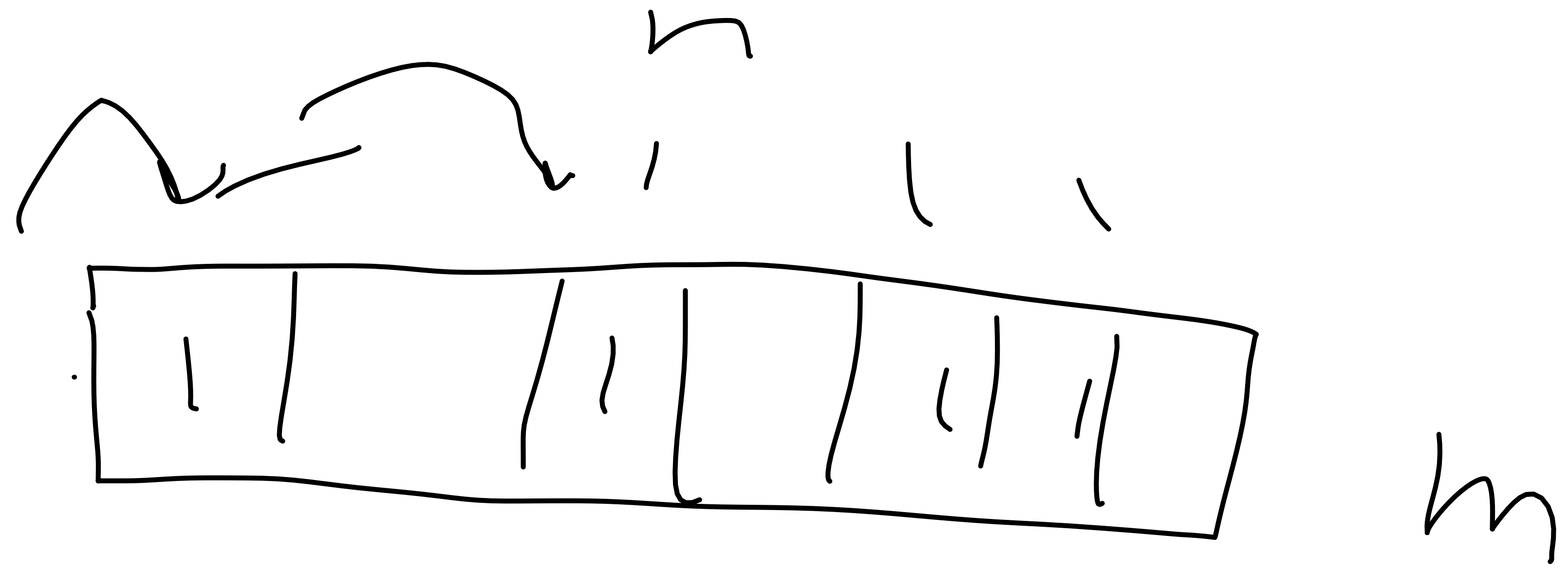
- For each key k, the value of $h(k)$ is uniformly random: that is, the probability that $h(k) = j$ is equal to $1/m$ for all j between 1 and m.

$\frac{1}{m}$



m

Hashing



What is the probability of a collision?

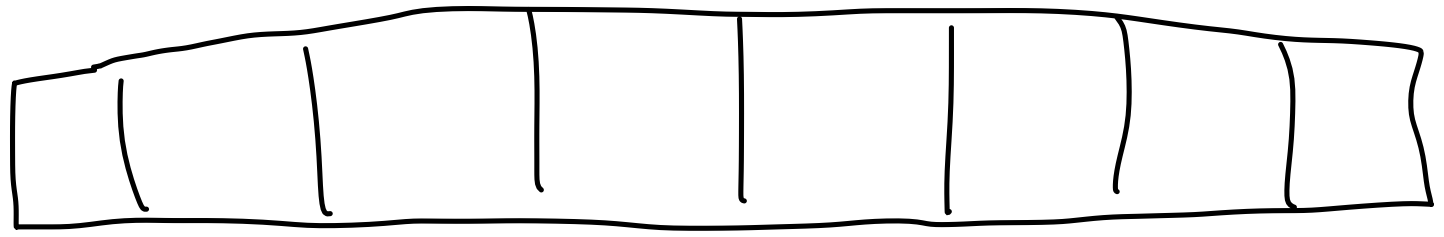
- Assume we have already inserted n keys into the structure and we would like to check whether some other key k *not in* S is contained in S (so the output should be “no”).
- The bit-string B contains at most n 1’s, and the value $h(k)$ is uniformly random; so the probability that $B[h(k)] = 1$ is at most n/m.
- That is, when we query the Bloom filter with some key k, if $k \notin S$ there is some small chance (say 1%) that the answer is “yes” when it should be “no”. On the other hand, if $k \in S$ the answer is always “yes”.
- So the probability that we incorrectly output “yes” for this key is at most n/m, and we never incorrectly output “no” for any key.
- So it suffices (for example) to take $m = 100n$ to achieve a failure probability of at most 1%. Note that m does not depend on the universe size U.

$n = 100 \rightarrow m = 100 \cdot 100 = 10000 \Rightarrow 1\%$

$n = 5000, \ m = 10000 \Rightarrow 50\%$

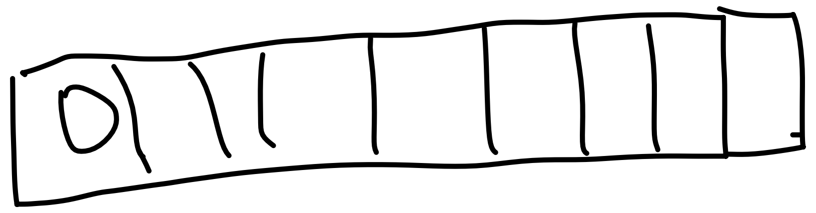
4bytes = 32 bits

int



2bytes → 16

$-2^{31} \ldots 2^{31}$



0, 1

bool 1byte → 8bits → 0 ... 255

$0 \ldots 2^{8}-1$

char

Can we do better?

We can achieve superior performance by using multiple hash functions.

- A Bloom filter consists of a string B of m bits, and a set of r hash functions $h_1,\ldots,h_r$.
- Each hash function maps a key k to an integer between 1 and m.
- For each i , we assume as before that $h_i(k)$ is uniformly random: that is, for each key k, the probability that $h_i(k) = j$ is equal to $1/m$ for all j between 1 and m.
- We will choose the parameters m and r later.

Inserting into a Bloom filter

To insert into a Bloom filter, we use the following simple procedure.

Insert(k)

1. for $i \leftarrow 1$ to r
2. $\quad B[h_i(k)] \leftarrow 1$

To check membership, we just check the bits of B that should be set to 1.

Member(k)

1. for $i \leftarrow 1$ to r
2. $\quad$ if $B[h_i(k)] = 0$
3. $\quad\quad$ return false
4. return true

Example

Imagine m = 4, r = 2, and we randomly pick the following hash functions:
h_1(www.bbc.co.uk) = 2, h_1(facebook.com) = 3,
h_1(cs.bristol.ac.uk) = 3.

h_2(www.bbc.co.uk) = 1, h_2(facebook.com) = 2,
h_2(cs.bristol.ac.uk) = 4.

Start	0	0	0	0
Insert(`www.bbc.co.uk`)	1	1	0	0
Insert(`facebook.com`)	1	1	1	0
Member(`cs.bristol.ac.uk`) returns No	1	1	1	0

Parameters

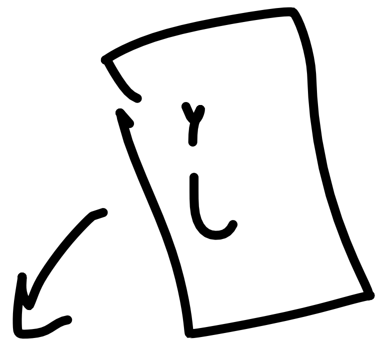
What is the probability of a collision?

- The probability that we incorrectly output 1 is at most $(nr/m)^r$
- By taking the derivative, we find that the minimum of $(nr/m)^r$ is achieved when $r = (m/n) \cdot \ln(2)$, where $e = 2.7818\ldots$.
- So, to achieve failure probability p, we can choose any m such that $m \geq -en \ln p$
- For small p, this is much better than using one hash function. For example, to achieve $p = 0.01$ (i.e. a 1% failure probability), we can take $m \approx 12.52n$.

Practical considerations

- We made the unrealistic assumption that each hash function h_i maps a key k to a uniformly random integer between 1 and m.

- In practice, we would pick each hash function h_i randomly from a fixed set of hash functions. One way of doing this for integer keys k (see CLRS §11.3.3) is to do the following for each i:
- 1) Pick a prime number $p > U$.
- 2) Pick random integers $a \in \{1,\ldots,p-1\}$, $b \in \{0,\ldots,p-1\}$.



- 3) Let h_i be defined by $h_i(k) = 1 + ((ak + b) \bmod p) \bmod m$.
- Some number theory can be used to prove that this set of hash functions is "pseudorandom" in some sense; however, technically they are not "random enough" for our analysis above to go through.
- Nevertheless, in practice hash functions like this are very effective.