CS2020
Data Structures and Algorithms
Today: Hashing (Part 3)


BIG
DATA
USE
ONE
SOFTWARE
USING
WORKING
GARTNER
MASSIVELY
BIOLOGICAL
PERFORMANCE
ALSO
HUNDREDS
BUSINESS
MOVING
ANALYTICS
SAN
PARALLEL
TOLERABLE
SIZE
NEEDED
PETABYTES
PROCESSING
LOGS
INTERNET
TECHNOLOGIES
DISK
GENOMICS
TOOLS
SET
SHARED
TIME
RELATIONAL
WITHIN
PROCESS
INCLUDE
SYSTEMS
DEFINITION
NETWORKS
SEARCH
INFORMATION
TENS
CAPACITY
PRACTITIONERS
DISTRIBUTED
CAPTURE
MAY
MANAGEMENT
MANAGE
GROW
SOCIAL
LARGE
STORE
RECORDS
COST
CONTINUES
TERABYTES
MPP
RESEARCH
STORAGE
EVERY
CASE
LARGER
CITATION
COMPLEX
DATABASES
DIFFICULTY
TARGET
ABILITY
SETS
EXAMPLES
ELAPSED
CURRENT
APPLIED
SENSOR
ARCHIVES

Problem Set 5

Big data analysis:

- How do we analyze a big database?
- Solve a simple data mining problem as fast as you can!

Problem Set 5

Speed Demon Competition

- How do we analyze a big database?
- Solve a simple data mining problem as fast as you can!
- Fastest solutions will win… eternal glory.

Problem Set 5

Speed Demon Competition

- Due: Thursday, March 17, midnight.

Today: Hash Tables (continued)

- Table (re)sizing
  - Proper hash table size
  - Amortized analysis
- Sets
  - Hash table sets
  - Bloom Filters

- Application: DNA analysis

Quick Review

Symbol Table

```
public interface  SymbolTable<Key, Value>

           void  insert(Key k, Value v)  insert (k,v) into table
          Value  search(Key k)           get value paired with k
           void  delete(Key k)           remove key k (and value)
        boolean  contains(Key k)         is there a value for k?
            int  size()                  number of (k,v) pairs
```

Note: no successor / predecessor queries.

Quick Review

Hash Table

- Implements a symbol table.
- Goal:
  - O(1) insert
  - O(1) lookup
- Idea:
  - Store data in a large array.
  - Hash function maps key to slot in the array.
  - Challenge: choosing a good hash function.

Quick Review

Hash Table with Chaining

- Each array slots stores a linked list.
- All items mapped to the same slot are stored in the linked list.

Open addressing:

- Each array slot stores one element.
- On collision, continue probing.
- Probe sequence specifies order in which cells are examined.

Table Size

How large should the table be?

- Assume: Hashing with Chaining
- Assume: Simple Uniform Hashing
- Expected search time: $O(1 + n/m)$
- Optimal size: $m = \Theta(n)$
  - if $(m < 2n)$: too many collisions.
  - if $(m > 10n)$: too much wasted space.
- Problem: we don't know n in advance.

Table Size

Idea:

- Start with small (constant) table size.
- Grow (and shrink) table as necessary.

Example:

- Initially, $m = 10$.
- After inserting 6 items, table too small! Grow…
- After deleting n-1 items, table too big! Shrink…

Table Size

Time complexity of growing the table:

- Assume:
  - Let m_1 be the size of the old hash table.
  - Let m_2 be the size of the new hash table.
  - Let n be the number of elements in the hash table.
- Costs:
  - Scanning old hash table: $O(m_1)$
  - Creating new hash table: $O(m_2)$
  - Inserting each element in new hash table: $O(1)$
  - Total: $O(m_1 + m_2 + n)$

How fast to grow?

Idea 1: Increment table size by 1

- if ($n == m$): $m = m+1$

- Cost of resize:
  - Size $m_1 = n$.
  - Size $m_2 = n+1$.

How fast to grow?

Idea 1: Increment table size by 1

- When $(n == m)$: $m = m+1$
- Cost of each resize: $O(n)$

Table size	8	8	9	10	11	12	…	n+1
Number of items	0	7	8	9	10	11	…	n
Number of inserts		7	1	1	1	1	…	1
Cost		7	8	9	10	11		n

- Total cost: $(7 + 8 + 9 + 10 + 11 + \ldots + n) = O(n^2)$

How fast to grow?

Idea 2: Double table size

- if $(n == m)$: $m = 2m$

- Cost of resize:
  - Size $m_1 = n$.
  - Size $m_2 = 2n$.
  - Total: $O(n)$

How fast to grow?

Idea 2: Double table size

- When ($n == m$): $m = 2m$
- Cost of each resize: $O(n)$

Table size	8	8	16	16	16	16	16	16	16	16	32	32	32	…	2n
# of items	0	7	8	9	10	11	12	13	14	15	16	17	18	…	n
# of inserts		7	1	1	1	1	1	1	1	1	1	1	1	…	1
Cost		7	8	1	1	1	1	1	1	1	16	1	1		n

- Total cost: $(8 + 16 + 32 + \ldots + n) = O(n)$

How fast to grow

Idea 2: Double table size

Cost of Resizing:

Table size	Total Resizing Cost
8	8
16	(8 + 16)
32	(8 + 16 + 32)
64	(8 + 16 + 32 + 64)
128	(8 + 16 + 32 + 64 + 128)
…	…
m	$<(1+2+4+8+\ldots+m) \leq 2m$

How fast to grow?

Idea 2: Double table size

- if $(n == m)$: $m = 2m$

  - Cost of resize: $O(n)$
  - Cost of inserting n items + resizing: $O(n)$

- Most insertions: $O(1)$
- Some insertions: linear cost (expensive)
- Average cost: $O(1)$

How fast to grow?

Idea 3: Square table size

- When ($n == m$): $m = m^2$

Table size	Total Resizing Cost
8	?
64	?
4,096	?
16,777,216	?
…	…
m	?

How fast to grow?

Idea 3: Square table size

– if $(n == m)$: $m = m^2$

– Cost of resize:

  • Total: $O(n^2)$

– Cost of inserts:

  • Total: $O(n)$

Deleting Elements

Basic procedure: (chained hash tables)

Delete(*key*)

1. Calculate hash of *key*.
2. Let L be the linked list in the specified bucket.
3. Search for item in linked list L.
4. Delete item from linked list L.

Cost:

- Total: $O(1 + n/m)$

Deleting Elements

What happens if too many items are deleted?

– Table is too big!
– Shrink the table…

– Try 1:
  - If $(n == m)$, then $m = 2m$.
  - If $(n < m/2)$ then $m = m/2$.

Deleting Elements

Rules for shrinking and growing:

- Try 1:
  - If $(n == m)$, then $m = 2m$.
  - If $(n < m/2)$ then $m = m/2$.

- Example problem:
  - Start: n=100, m=200
  - Delete: n=99, m=200 → shrink to m=100
  - Insert: n=100, m=100 → grow to m=200
  - Repeat…

Deleting Elements

Example execution:

- Start: n=100, m=200
- cost=100 • Delete: n=99, m=200 → shrink to m=100
- cost=100 • Insert: n=100, m=100 → grow to m=200
- cost=100 • Delete: n=99, m=200 → shrink to m=100
- cost=100 • Insert: n=100, m=100 → grow to m=200
- cost=100 • Delete: n=99, m=200 → shrink to m=100
- cost=100 • Insert: n=100, m=100 → grow to m=200
- Repeat…

Deleting Elements

Rules for shrinking and growing:

- Try 2:
  - If $(n == m)$, then $m = 2m$.
  - If $(n < m/4)$, then $m = m/2$.

- Claim:
  - Every time you double a table of size m, at least $m/2$ new items were added.
  - Every time you shrink a table of size m, at least $m/4$ items were deleted.

Deleting Elements

Example execution:

- Start: n=100, m=200

cost=350 • Delete 50: n=50, m=200 → shrink to m=100

cost=350 • Insert 50: n=100, m=100 → grow to m=200

cost=20 • Delete 20: n=80, m=200 → unchanged

cost=720 • Insert 120: n=200, m=200 → grow to m=400

cost=100 • Insert 100: n=300, m=400 → unchanged

Amortized Analysis

Technique for analyzing "average" cost:

- Common in data structure analysis
- Like paying rent:
  - You don't pay rent every day!
  - Pay \$900/month = \$30/day.

Definition:

- Operation has <u>amortized cost</u> $\mathrm{T}(n)$ if for every integer k, the cost of k operations is $\leq k\,\mathrm{T}(n)$

Amortized Analysis

Definition:

– Operation has <u>amortized cost</u> $T(n)$ if for every integer k, the cost of k operations is $\leq k\, T(n)$

Example: amortized cost = 7

1. insert: 5
2. insert: 5
3. insert: 5
4. insert: 13
5. insert: 7

5 <= 7

5+5 <= 2*7 = 14

5+5+5 <= 3*7 = 21

5+5+5+13 <= 4*7 = 28

5+5+5+13+7 <= 5*7 = 35

Amortized Analysis

Definition:

- Operation has <u>amortized cost</u> $\mathrm{T}(n)$ if for every integer k, the cost of k operations is $\leq k\,\mathrm{T}(n)$

Example: amortized cost **NOT** 7

1. insert: 13
2. insert: 5
3. insert: 5
4. insert: 5
5. insert: 7

13 > 7

13+5 > 2*7 = 14

13+5+5 > 3*7 = 21

13+5+5+5 <= 4*7 = 28

5+5+5+13+7 <= 5*7 = 35

Amortized Analysis

Definition:

- Operation has amortized cost $T(n)$ if for every integer k, the cost of k operations is $\leq k\, T(n)$

Example: (Hash Tables)

- Inserting k elements into a hash table takes time $O(k)$.
- Conclusion:

  The insert operation has amortized cost $O(1)$.

Amortized Analysis

Accounting Method (paying rent)

- Imagine a bank account **B**.
- Each operation adds money to the bank account.
- Every step of the algorithm spends money:
  - Immediate money: to perform the operation.
  - Deferred money: from the bank account.
- Total cost execution = total money
  - Average time / operation = money / num. ops

Amortized Analysis

Accounting Method Example (Hash Table)

- Each table has a bank account.
- Each time an element is added to the table, it adds O(1) dollars to the bank account, uses O(1) dollars to insert element.
- A table with *k* new elements since last resize has *k* dollars in bank.

Bank account
$2 dollars

0	null
1	null
2	**(k_1, A)**
3	null
4	null
5	null
6	null
7	null
8	**(k_2, B)**
9	null

Amortized Analysis

Accounting Method Example (Hash Table)

- The table has a bank account.
- Each time an element is added to the table, it adds O(1) dollars to the bank account.

- Claim:
  - Growing a table of size m: $(m + m + 2m) = O(m)$ time.
  - Shrinking a table of size m: $(m + m/4 + m/2) = O(m)$ time.
  - If you resize a table of size m, then:
    - at least $m/2$ added OR $m/4$ deleted since last resize.
    - bank account has $\Theta(m)$ dollars.

Amortized Analysis

Accounting Method Example (Hash Table)

- Each table has a bank account.
- Each time an element is added to the table, it adds O(1) dollars to the bank account.
- Pay for resizing from the bank account!
- Strategy:
  - Analyze inserts ignoring cost of resizing.
  - Ensure that bank account always is big enough to pay for resizing.

Amortized Analysis

Total cost: Inserting k elements costs:

- Deferred dollars: $O(k)$ (to pay for resizing)
- Immediate dollars: $O(k)$ for inserting elements in table
- Total (Deferred + Immediate): $O(k)$

Amortized Analysis

Total cost: Inserting k elements costs:

- Deferred dollars: $O(k)$ (to pay for resizing)
- Immediate dollars: $O(k)$ for inserting elements in table
- Total (Deferred + Immediate): $O(k)$

Average cost per operation:

- Deferred dollars: $O(1)$
- Immediate dollars: $O(1)$
- Total: $O(1)$ / per operation

Example: Binary Counter

Counter ADT:

- increment()
- read()

0	0	0	0	0	0	0	0	0	0

Example: Binary Counter

Counter ADT:

- increment()
- read()

increment()

0	0	0	0	0	0	0	0	0	1

Example: Binary Counter

Counter ADT:

- increment()
- read()

increment(), increment()

0	0	0	0	0	0	0	0	1	0

Example: Binary Counter

Counter ADT:

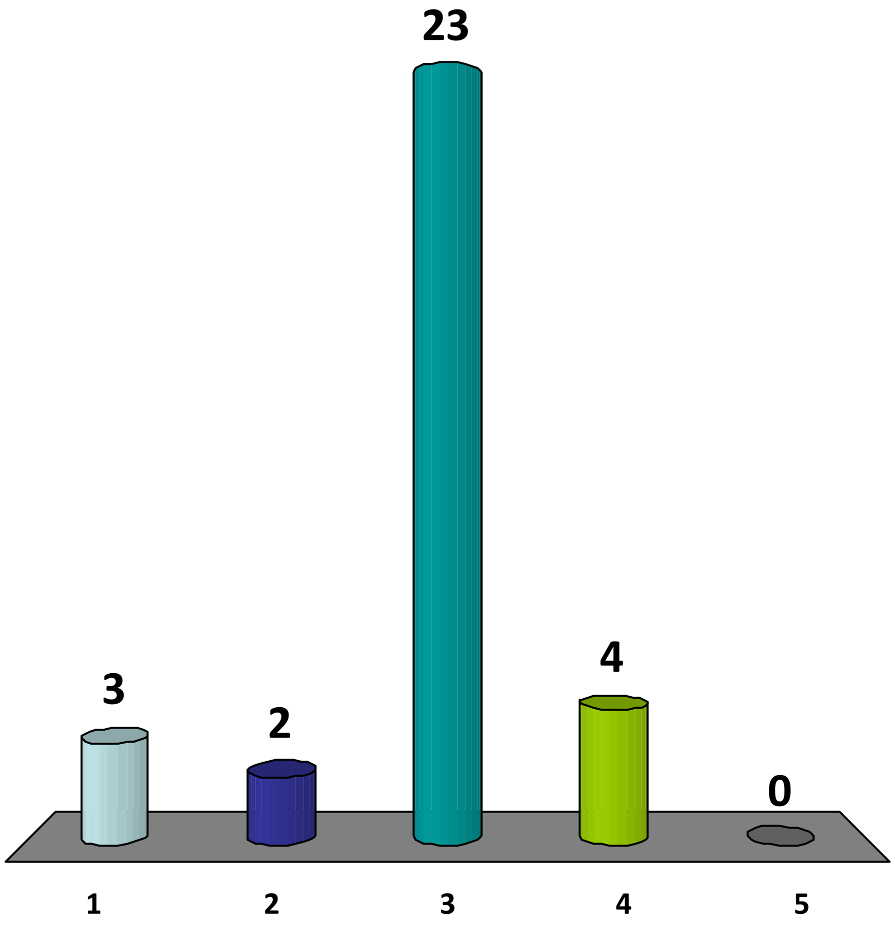
- increment()
- read()

increment(), increment(), increment()

0	0	0	0	0	0	0	0	1	1

What is the worst-case cost of incrementing the counter?

1. O(1)
2. O(log n)
3. O(n)
4. O(n^2)
5. I have no idea.



Response Counter

Example: Binary Counter

Counter ADT:

- increment()
- read()

Some increments are expensive…

0	1	1	1	1	1	1	1	1	1

$\Downarrow$ O(log n)

1	0	0	0	0	0	0	0	0	0

Example: Binary Counter

Question: If we increment the counter to n, what is the average cost per operation?

- Easy answer: O(log n)
- More careful analysis….

0	1	1	1	1	1	1	1	1	1

$\Downarrow$ O(log n)

1	0	0	0	0	0	0	0	0	0

Example: Binary Counter

Observation:

During each increment, only <u>one</u> bit is changed from: 0 → 1

0	1	1	1	1	1	1	1	1	1

⇩

1	0	0	0	0	0	0	0	0	0

Example: Binary Counter

Observation:

During each increment, <u>many</u> bits may be changed from: 1 → 0

0	1	1	1	1	1	1	1	1	1

⇓

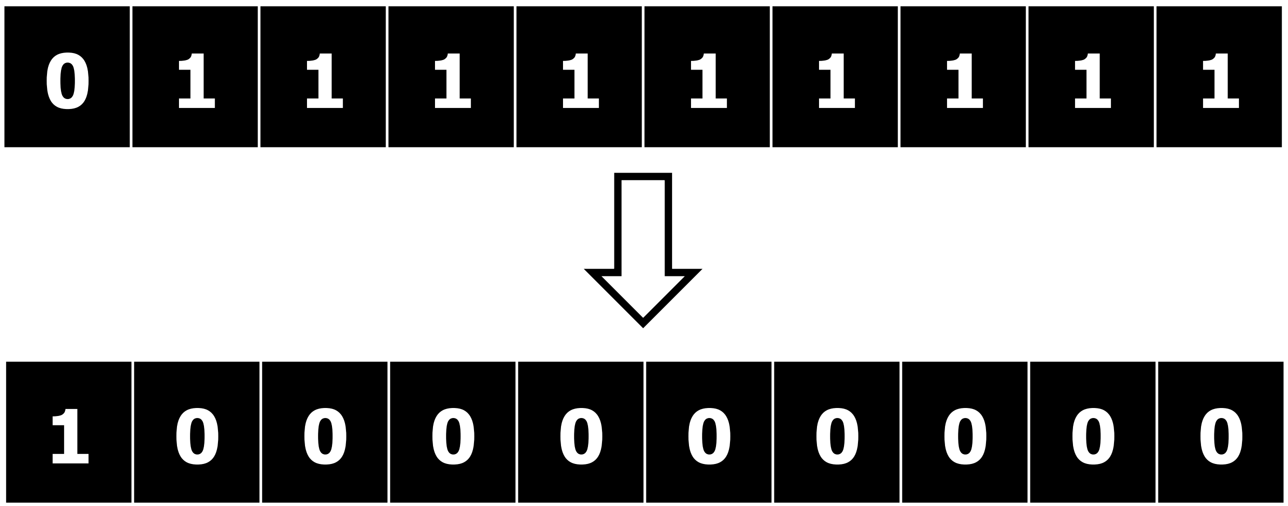
1	0	0	0	0	0	0	0	0	0

Example: Binary Counter

Observation:

Accounting method: each bit has a bank account.

Whenever you change it from 0→1, add one dollar.



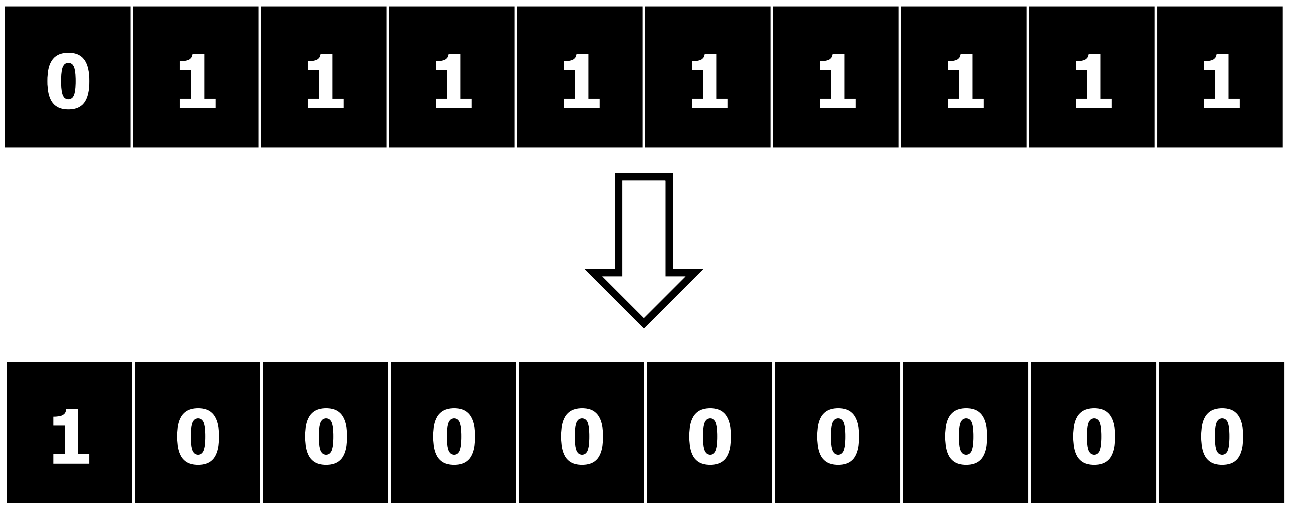
Example: Binary Counter

Observation:

Accounting method: each bit has a bank account.

Whenever you change it from 0→1, add one dollar.

Whenever you change it from 1→0, pay one dollar.



Example: Binary Counter

Counter ADT

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	0	0	0	0	0	0	0	0	0

Example: Binary Counter

Counter ADT

increment()

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

⇓

0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	1

Example: Binary Counter

Counter ADT

increment(), increment()

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**
0	0	0	0	0	0	0	0	0	1

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**
0	0	0	0	0	0	0	0	1	0

Example: Binary Counter

Counter ADT

increment(), increment(), increment()

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**

0 0 0 0 0 0 0 0 1 0

⇩

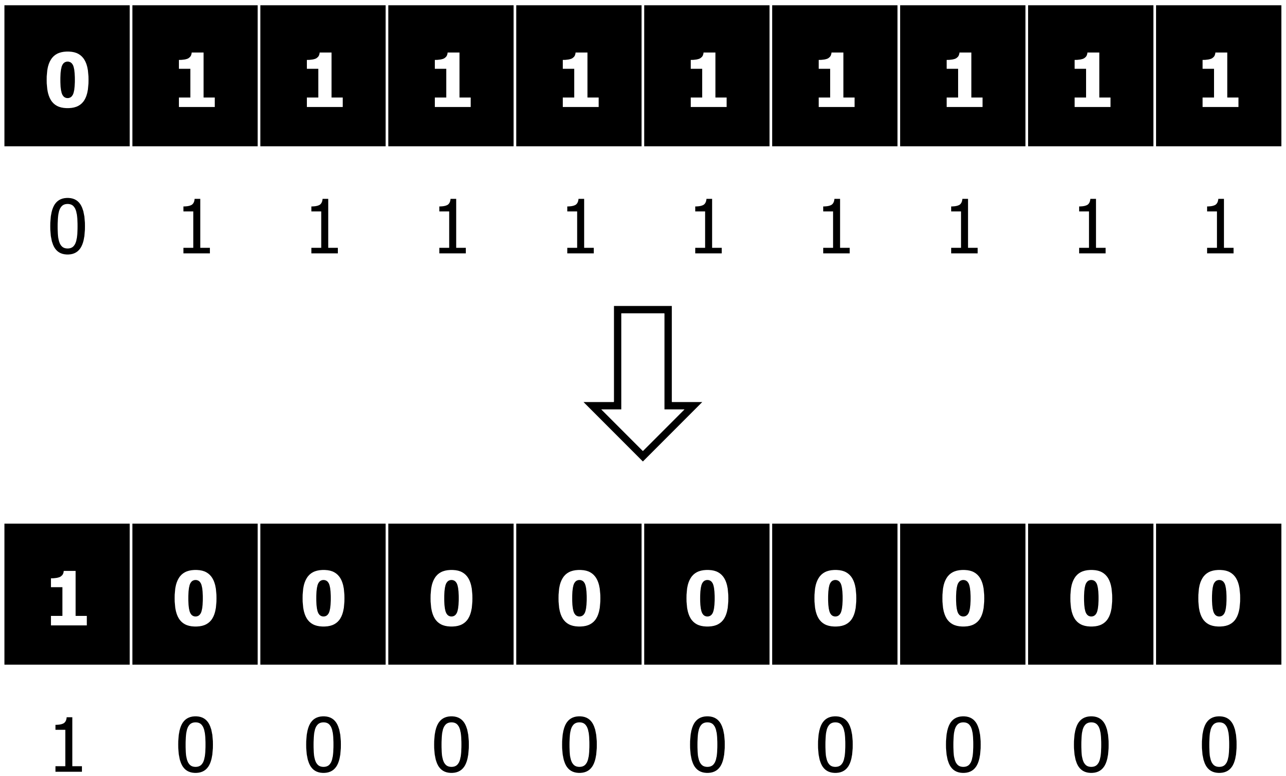
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**

0 0 0 0 0 0 0 0 1 1

Example: Binary Counter

Counter ADT

increment()



Example: Binary Counter

Observation:

Average cost of increment: 2

- One operation to switch <u>one</u> 0→1
- One dollar (for bank account of switched bit).

(All switches from 1→0 paid for by bank account.)

0	1	1	1	1	1	1	1	1	1

⇩

1	0	0	0	0	0	0	0	0	0

Today: Hash Tables (continued)

- Table (re)sizing
  - Proper hash table size
  - Amortized analysis
- Sets
  - Hash table sets
  - Bloom Filters

A few examples

Facebook:

- I have a list of (names) of friends:
  - John
  - Mary
  - Bob
- Some are online, some are offline.
- How do I determine which are on-line and which are off-line?

A few examples

Spam filter:

- I have a list bad e-mail addresses:
  - @ mxkp322ochat.com
  - @ info.dhml212oblackboard.net
  - @ transformationalwellness.com
- I have a list of good e-mail addresses:
  - My mom.
  - *.nus.edu.sg

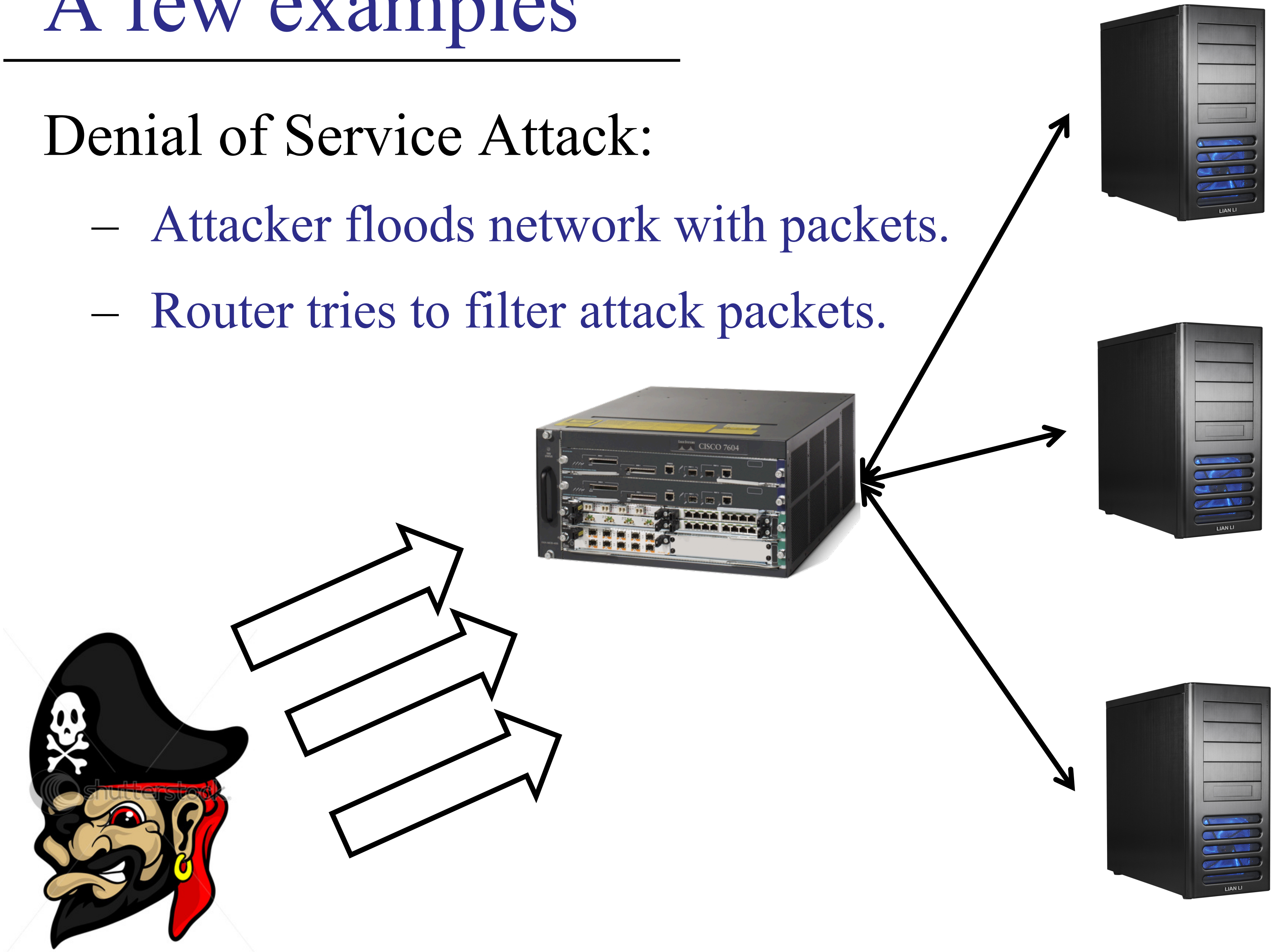
- How do I quickly check for spam?

A few examples

Denial of Service Attack:

– Attacker floods network with packets.

– Router tries to filter attack packets.



A few examples

Denial of Service Attack:

- Attacker floods network with packets.
- Router tries to filter attack packets.

1. Keep list of bad IP addresses. (Same as spam solution.)

2. Only allow 100 packets/second from each IP address.

Abstract Data Type

Set

public class	**Set<Key>**	
void	insert(Key k)	*Insert k into set*
boolean	contains(Key k)	*Is k in the set?*
void	delete(Key k)	*Remove key k from the set*
void	intersect(Set<Key> s)	*Take the intersection.*
void	union(Set<Key> s)	*Take the union.*

Properties:

- No defined ordering.
- Speed is critical.
- Space is critical.

Abstract Data Type

Set

`public class Set<Key>`

`void`	`insert(Key k)`	*Insert k into set*
`boolean`	`contains(Key k)`	*Is k in the set?*
`void`	`delete(Key k)`	*Remove key k from the set*
`void`	`intersect(Set<Key> s)`	*Take the intersection.*
`void`	`union(Set<Key> s)`	*Take the union.*

Java: HashSet<…> implements Set<…>

Abstract Data Type

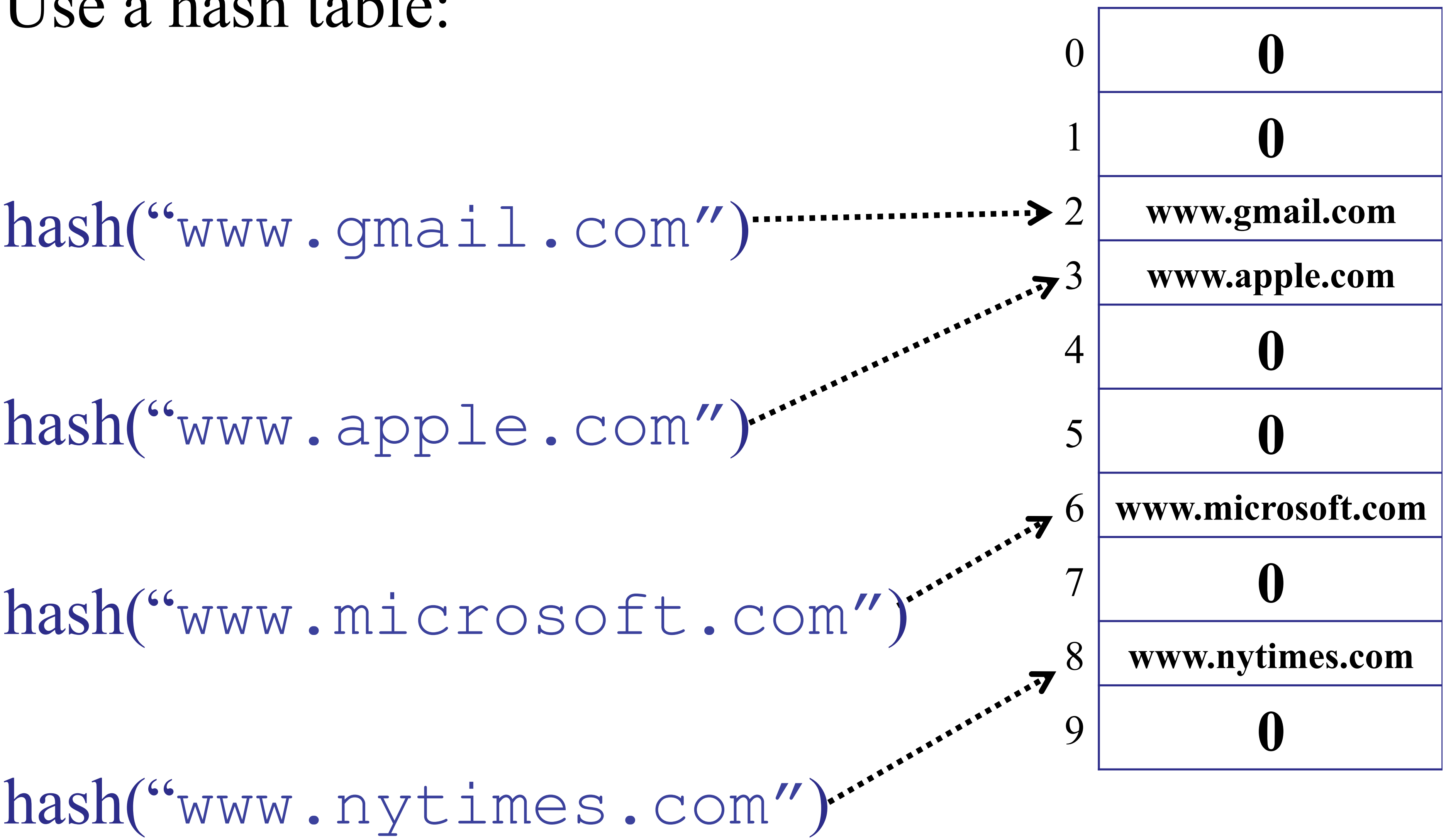
Set

public class	**Set<Key>**	
void	insert(Key k)	*Insert k into set*
boolean	contains(Key k)	*Is k in the set?*
void	delete(Key k)	*Remove key k from the set*
void	intersect(Set<Key> s)	*Take the intersection.*
void	union(Set<Key> s)	*Take the union.*

Solution 1: Implement using a Hash Table

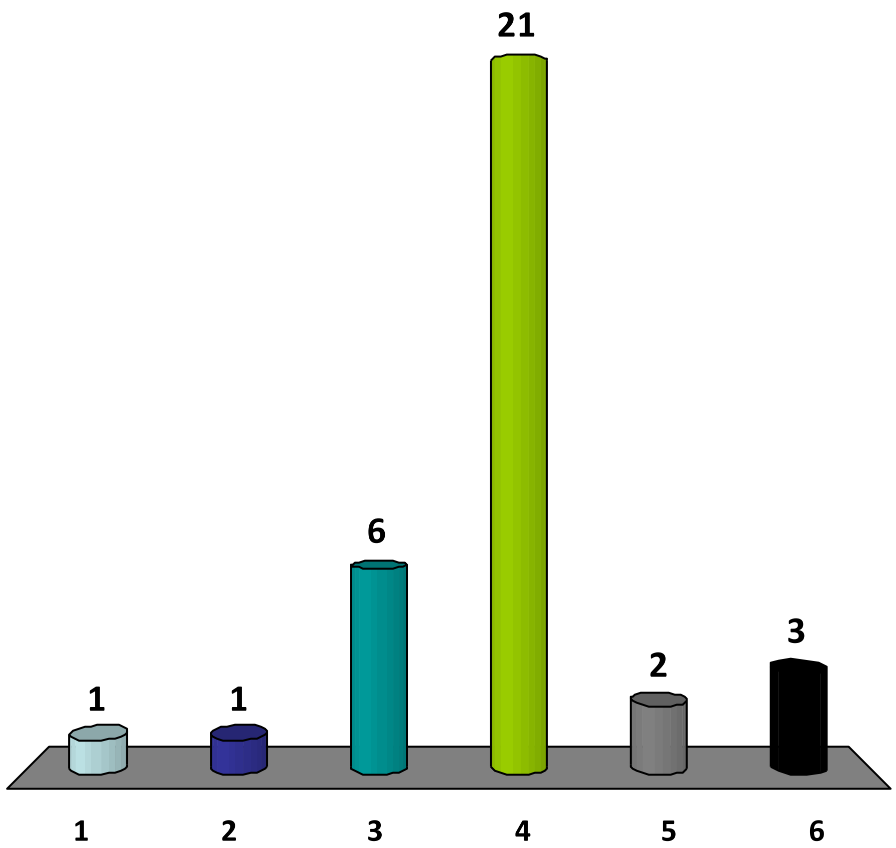
Implementing a Set

Use a hash table:



Which problem does a hash table not solve?

1. Fast insertion
2. Fast deletion
3. Fast lookup
4. Small space
5. All of the above
6. None of the above



A hash table takes **more** space than a simple list!

Response Counter

Implementing a Set

Use a hash table:

hash(“www.gmail.com”)

hash(“www.apple.com”)

hash(“www.microsoft.com‘

hash(“www.nytimes.com”)

0	**0**
1	**0**
2	**www.gmail.com**
3	**www.apple.com**
4	**0**
5	**0**
6	**www.microsoft.com**
7	**0**
8	**www.nytimes.com**
9	**0**

Abstract Data Type

Set

public class	**Set<Key>**	
`void`	`insert(Key k)`	*Insert k into set*
`boolean`	`contains(Key k)`	*Is k in the set?*
`void`	`delete(Key k)`	*Remove key k from the set*
`void`	`intersect(Set<Key> s)`	*Take the intersection.*
`void`	`union(Set<Key> s)`	*Take the union.*

Solution 2: Implement using a Fingerprint Hash Table

Implementing a Set

Use a fingerprint:

- Only store/send *m* bits!

hash(“`www.gmail.com`”)

hash(“`www.apple.com`”)

hash(“`www.microsoft.com`”)

hash(“`www.nytimes.com`”)

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

Fingerprints

Set Abstract Data Type

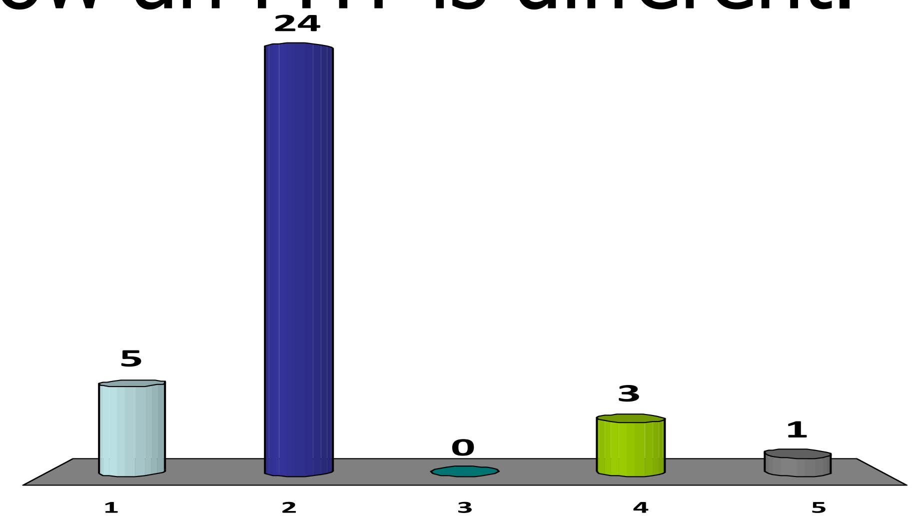
– Maintain a vector of 0/1 bits.

```
insert(key)
1. h = hash(key);
2. m_table[h] = 1;

lookup(key)
1. h = hash(key);
2. return (m_table[h] == 1);
```

The key difference of a Fingerprint Hash Table (FHT) is:

1. A FHT prevents collisions.
2. A FHT does not store the key in the table.
3. A FHT works with simpler hash functions.
4. A FHT saves time calculating hashes.
5. I don't understand how an FHT is different.



Response Counter

Implementing a Set

Use a fingerprint:

hash(“`www.gmail.com`”)

hash(“`www.apple.com`”)

hash(“`www.microsoft.com`”)

hash(“`www.nytimes.com`”)

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

Implementing a Set

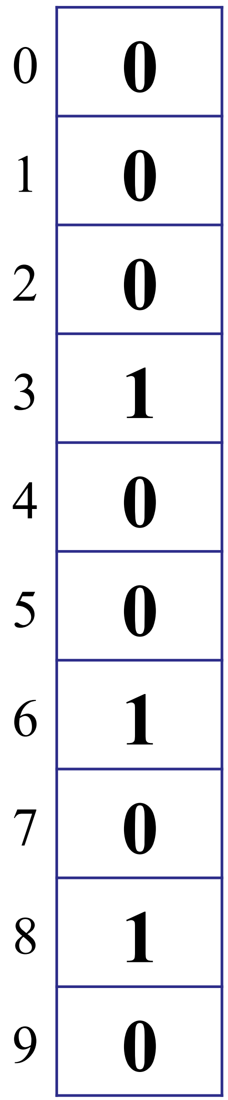
What happens on collision?

hash(“`www.gmail.com`”)

hash(“`www.apple.com`”)

hash(“`www.microsoft.com`”)

hash(“`www.nytimes.com`”)



Implementing a Set

Lookup operation:

hash(“`www.microsoft.com`”)

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

If the URL is in the web cache, it will always report **true**.
(No false negatives.)

Fingerprint Hash Table

Insert operation:

hash(“`www.microsoft.com`”)

Lookup operation:

hash(“`www.rugby.com`”)

Even if the URL is NOT in the set,

it may *sometimes* report **true**.

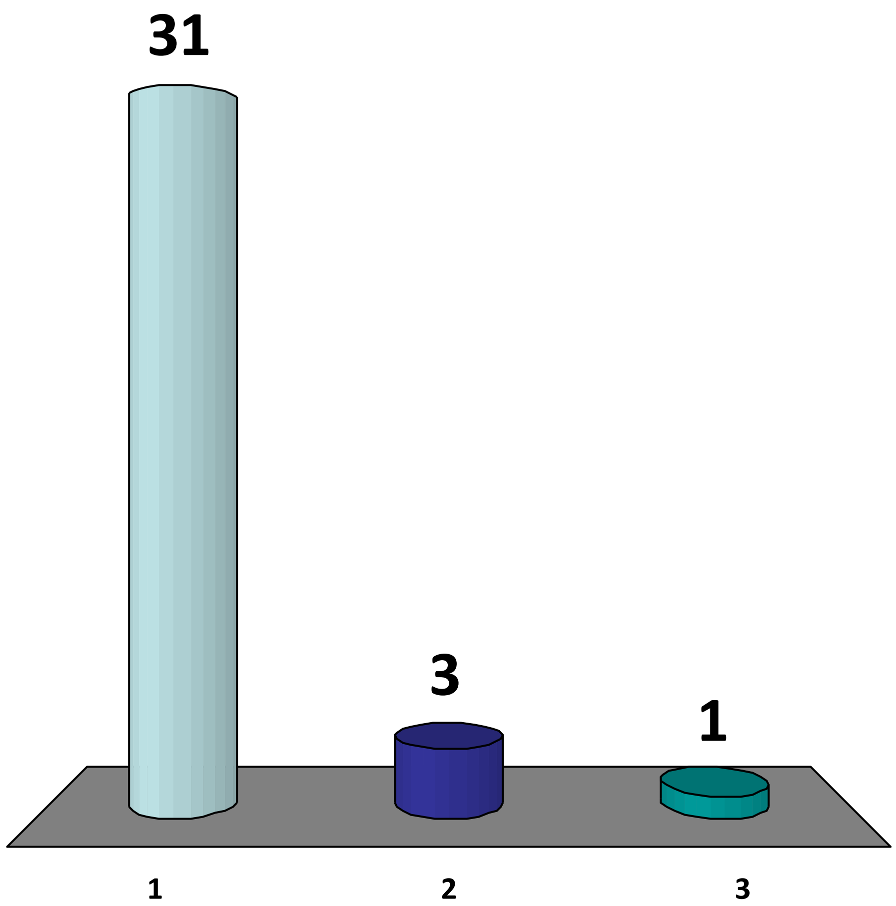
(False positives.)

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

Facebook example: if the FHT stores the set of online users, then you might:

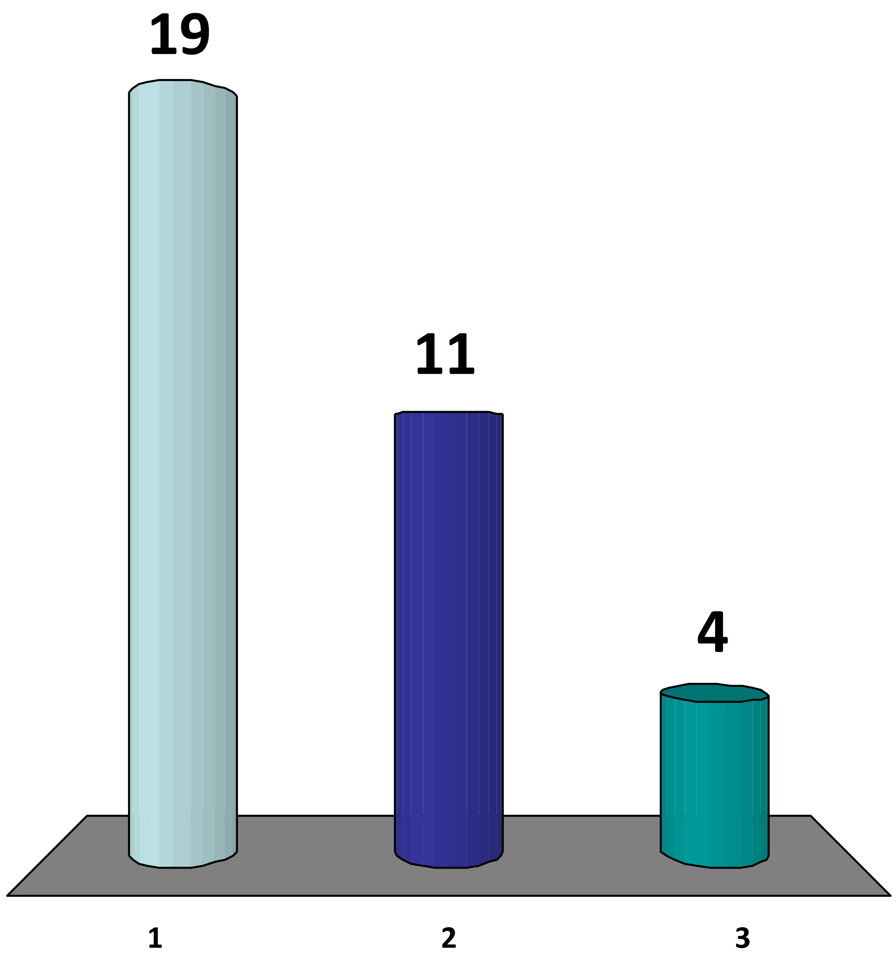
1. Believe Fred is on-line, when he is not.
2. Believe Fred is offline, when is not.
3. Never make any mistakes.

Response Counter



Spam example: it is better to store in the Fingerprint Hash Table:

1. The set of **good** e-mail addresses.
2. The set of **bad** e-mail addresses
3. It does not matter.



I think it is better to mistakenly accept a few SPAM e-mails than to accidently reject an e-mail from my mother!

Response Counter

Fingerprint Analysis

Probability of a false negative: 0

Fingerprint Analysis

Probability of a false negative: 0

On lookup in a table of size m with n elements,

Probability of **no** false positive:

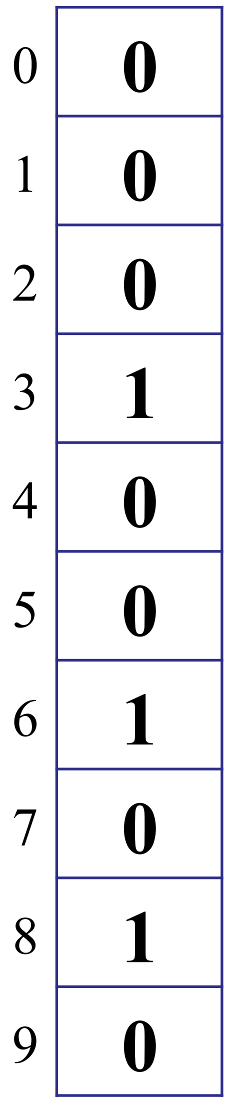
$$\left(1 - \frac{1}{m}\right)^{n} \approx \left(\frac{1}{e}\right)^{n/m}$$

chance of no collision

Fingerprint Analysis

Probability of collision?

hash(“`www.gmail.com`”)



What is the probability that no other URL is in slot 3?

Fingerprint Analysis

Probability of a false negative: 0

Probability of **no** false positive: (simple uniform hashing assumption)

$$\left(1 - \frac{1}{m}\right)^n \approx \left(\frac{1}{e}\right)^{n/m}$$

Probability of a false positive, at most:

$$1 - \left(\frac{1}{e}\right)^{n/m}$$

Fingerprint Analysis

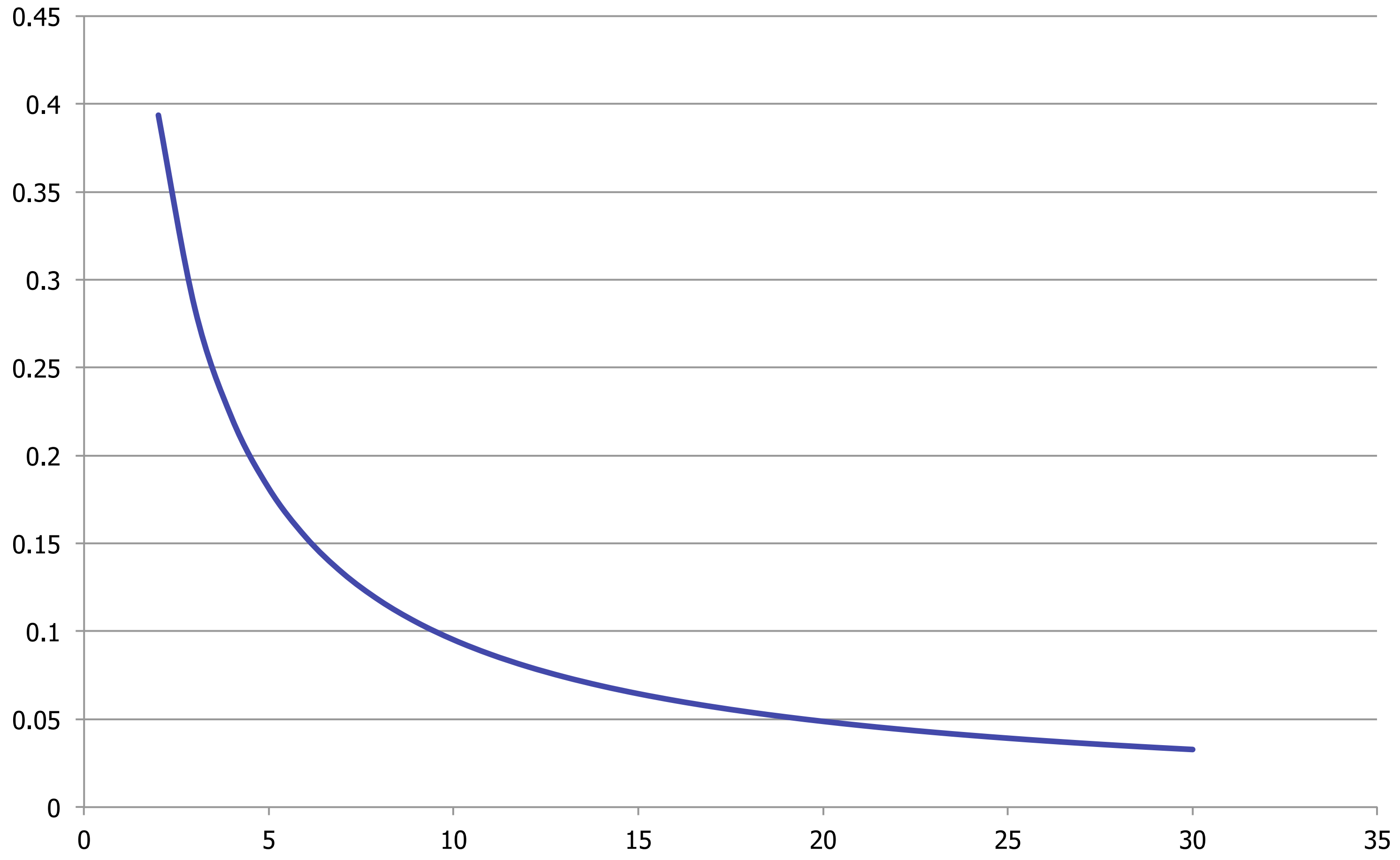
Assume you want:

- probability of false positives $< p$
  - Example: at most 1% of queries return false positive.

    $p = .01$

- Need: $\frac{n}{m} \leq \log\left(\frac{1}{1-p}\right)$
  - Example: $m >= (68.97)n$

Fingerprint Analysis



probability of false positive vs (m/n)

Summary So Far

Fingerprint Hash Functions

- Don't store the key.
- Only store 0/1 vector.

Summary So Far

Fingerprint Hash Functions

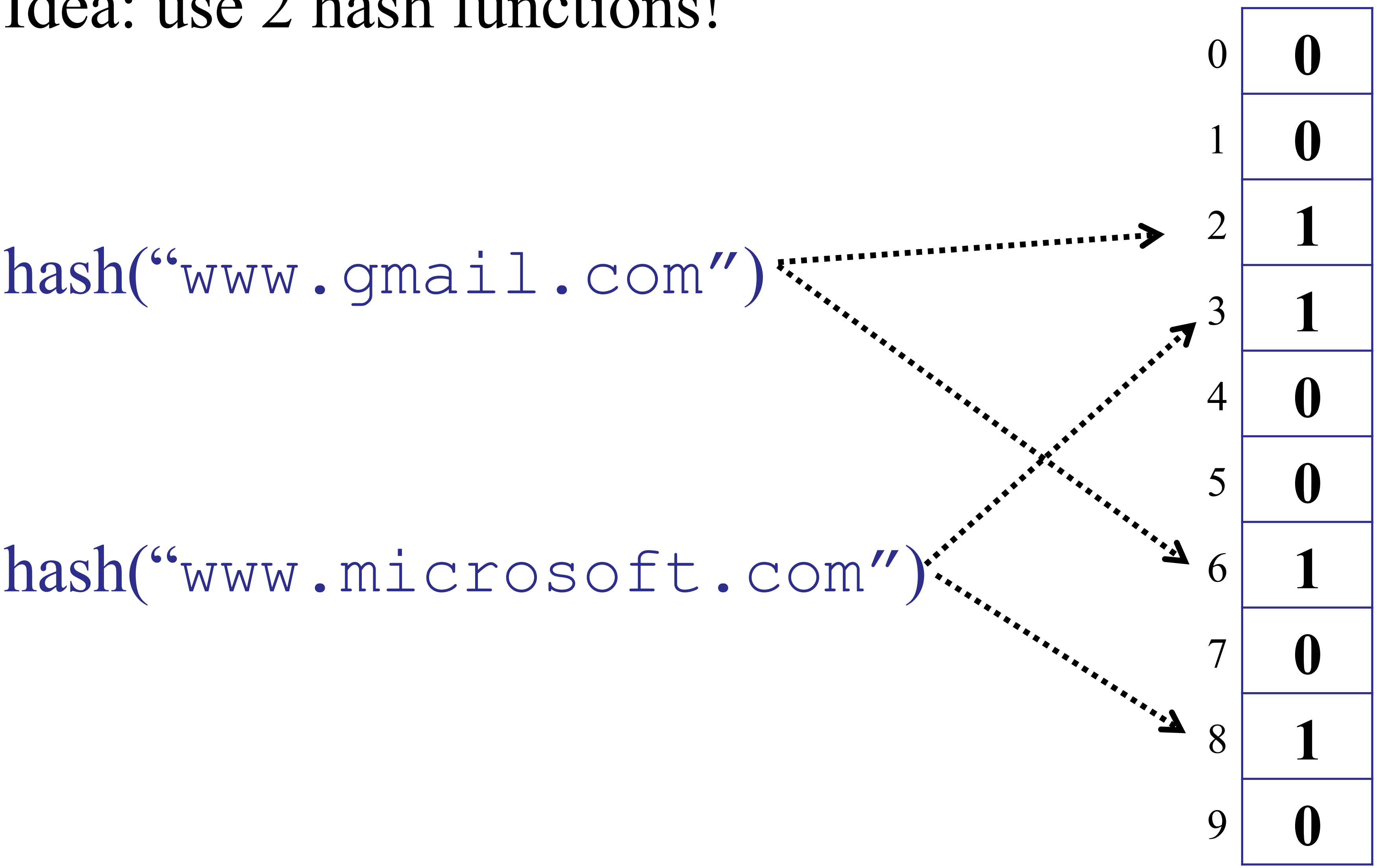
- Don't store the key.
- Only store 0/1 vector.
- Trade-off:
  - Reduced space: only 1-bit per slot
  - Increase space: bigger table to avoid collisions

Fingerprint Hash Table

Can we do better?

Bloom Filter

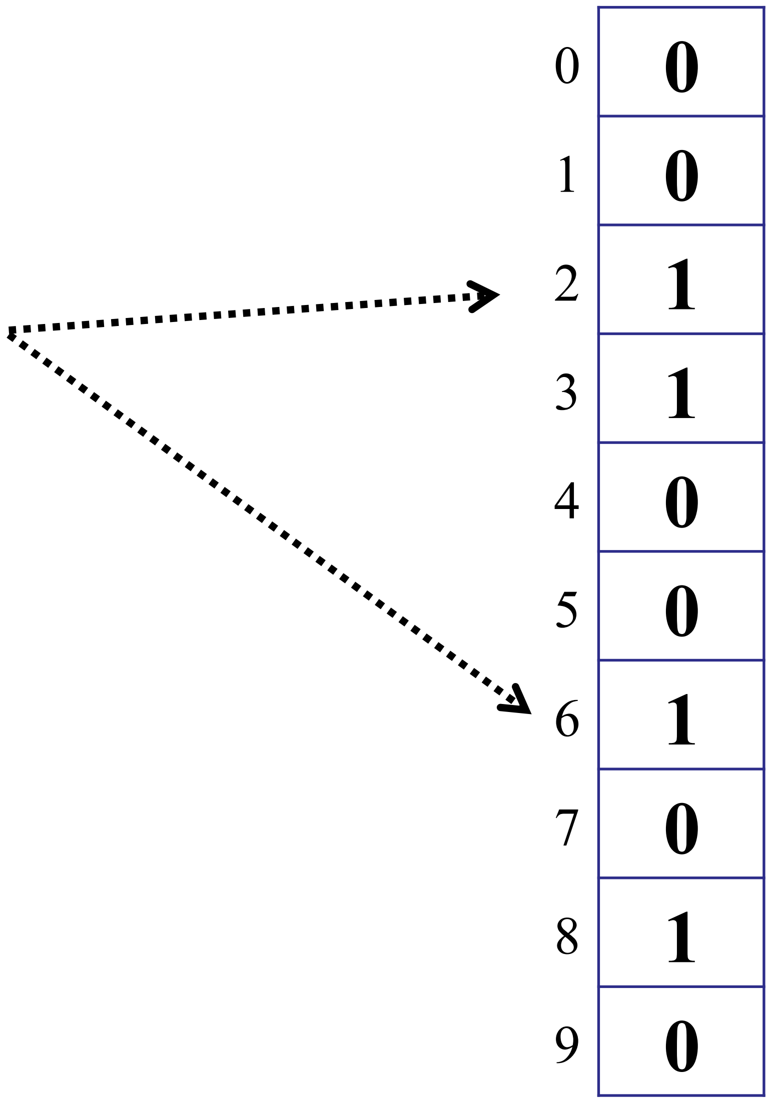
Idea: use 2 hash functions!



Bloom Filter

Idea: use 2 hash functions!

hash(“`www.gmail.com`”)



insert(URL)

$k_1 = \text{hash}_1(\text{URL});$

$k_2 = \text{hash}_2(\text{URL});$

$\text{T}[k_1] = 1;$

$\text{T}[k_2] = 1;$

Bloom Filter

Idea: use 2 hash functions!

query(URL)

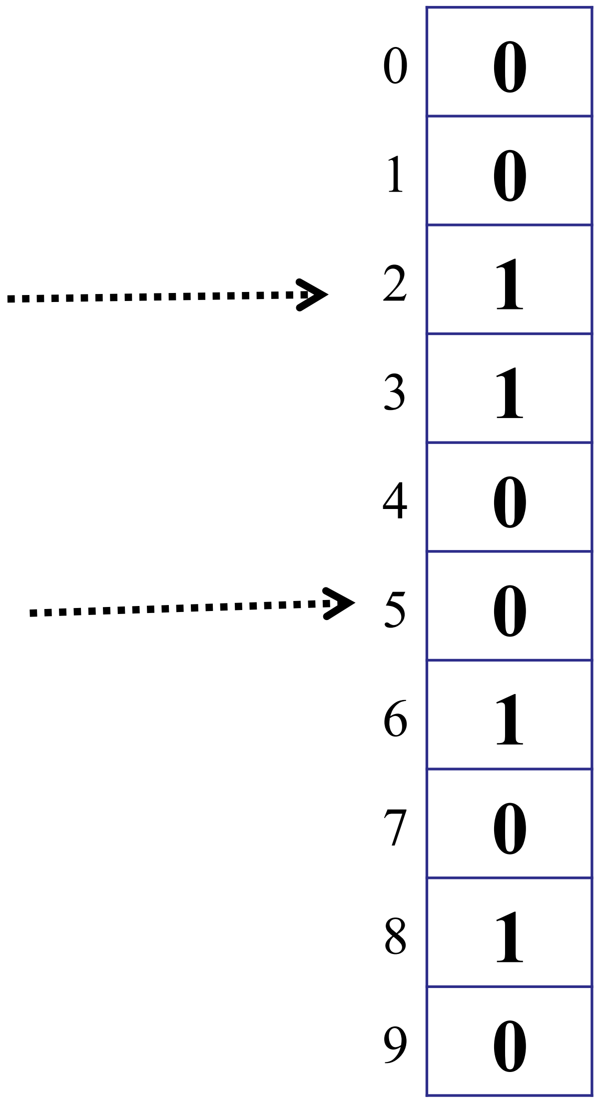
k_1 = hash$_1$(URL);

k_2 = hash$_2$(URL);

if (T[k_1] && T[k_2])

return true;

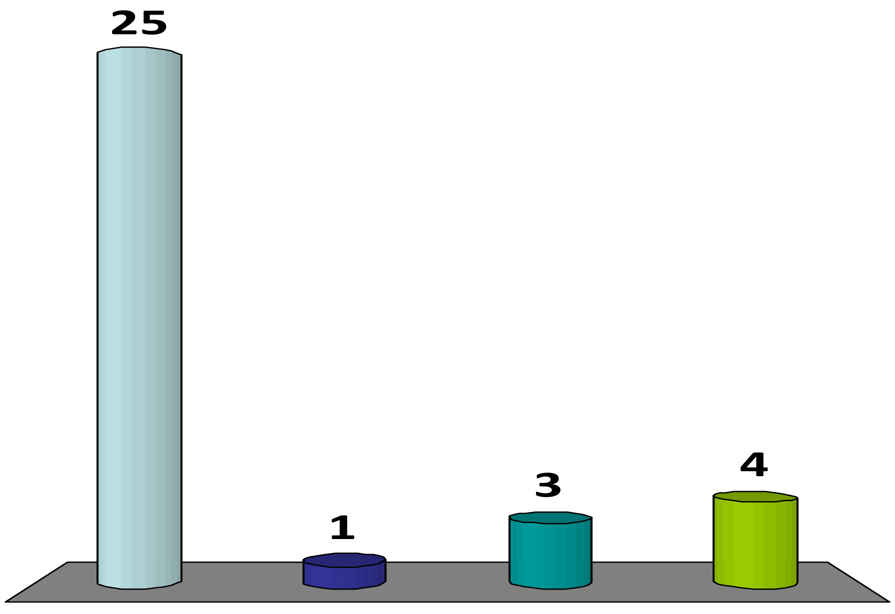
else return false;



A Bloom Filter can have:

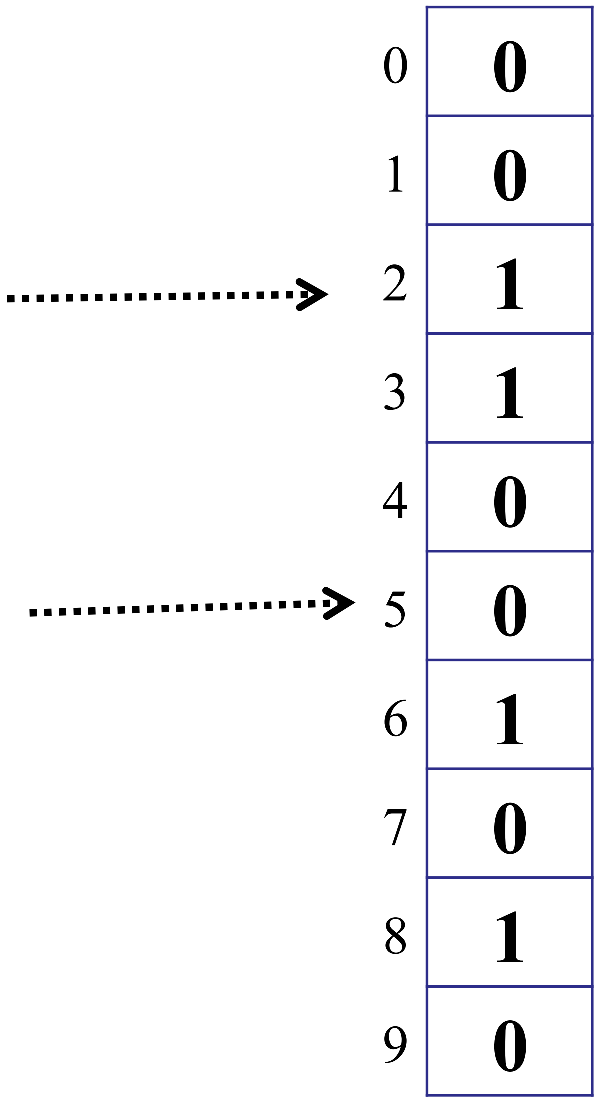
1. Only false positives.
2. Only false negatives.
3. Both false positives and negatives.
4. Wait, which is which again?

Response Counter



Bloom Filter

Idea: use 2 hash functions!



query(URL)

k_1 = hash$_1$(URL);

k_2 = hash$_2$(URL);

if (T[k_1] && T[k_2])

return true;

else return false;

Bloom Filter

Idea: use 2 hash functions!

Trade-off:

- Each item takes more "space" in the table.
- Requires <u>two</u> collisions for a false positive.

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

Bloom Filter Analysis

Probability a given bit is 0:

$$\left(1 - \frac{1}{m}\right)^{2n} \approx \left(\frac{1}{e}\right)^{2n/m}$$

chance hash does not choose this bit

each of n items sets 2 bits in the table

Bloom Filter Analysis

Probability a given bit is 0:

$$\left(1 - \frac{1}{m}\right)^{2n} \approx \left(\frac{1}{e}\right)^{2n/m}$$

Probability of a false positive: (1 set in both slots)

$$\left(1 - \left(\frac{1}{e}\right)^{2n/m}\right)^{2}$$

Bloom Filter Analysis

Probability a given bit is 0:

$$\left(1-\frac{1}{m}\right)^{2n} \approx \left(\frac{1}{e}\right)^{2n/m}$$

Probability of a false positive: (1 set in both slots)

$$\left(1-\left(\frac{1}{e}\right)^{2n/m}\right)^{2}$$

* Assuming BOGUS fact that each table slot is independent…

Bloom Filter Analysis

Assume you want:

– probability of false positives $< p$

  • Example: at most 1% of queries return false positive.

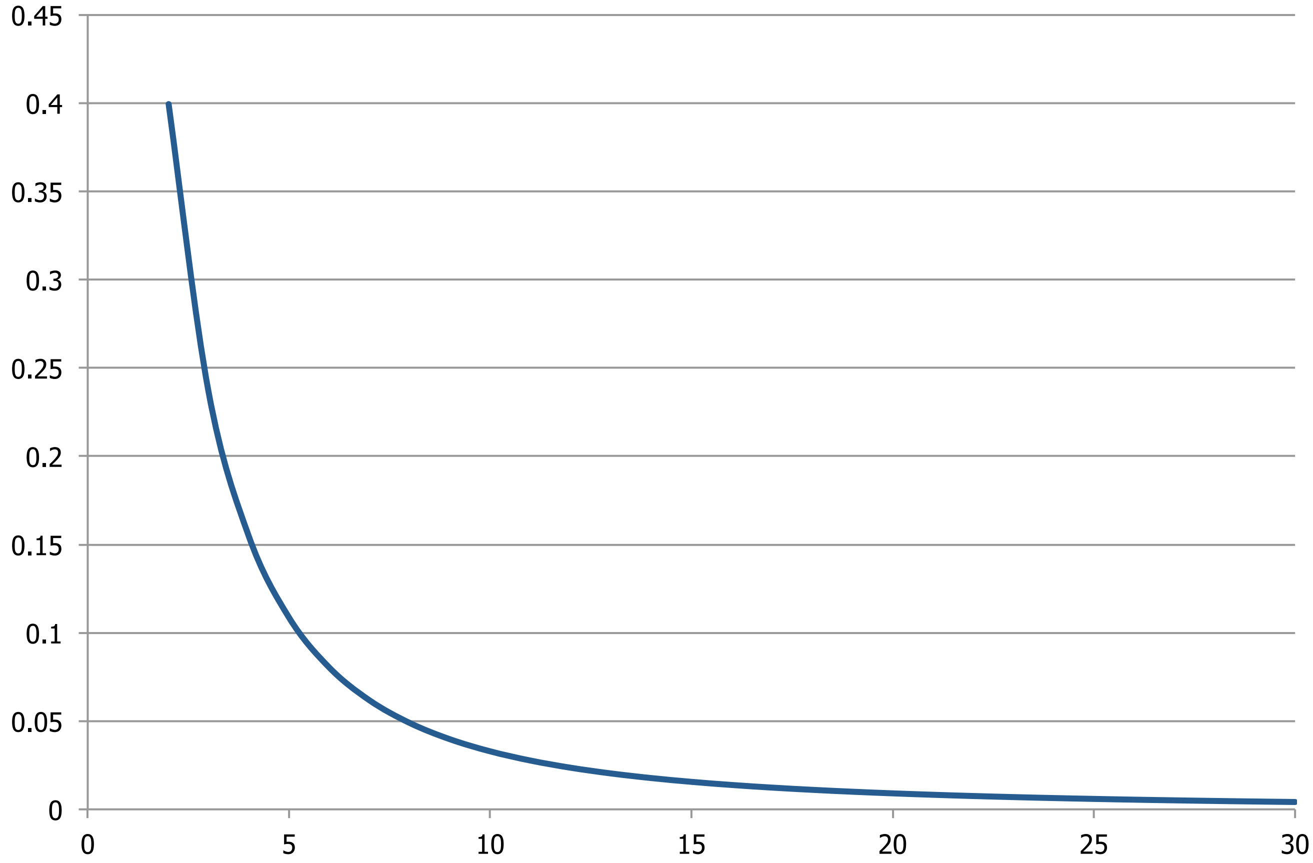
$$p = .01$$

– Need: $\frac{n}{m} \leq \frac{1}{2}\log\left(\frac{1}{1-p^{1/2}}\right)$

  • Example: $m >= 19n$

* Assuming BOGUS fact that each table slot is independent…

Bloom Filter

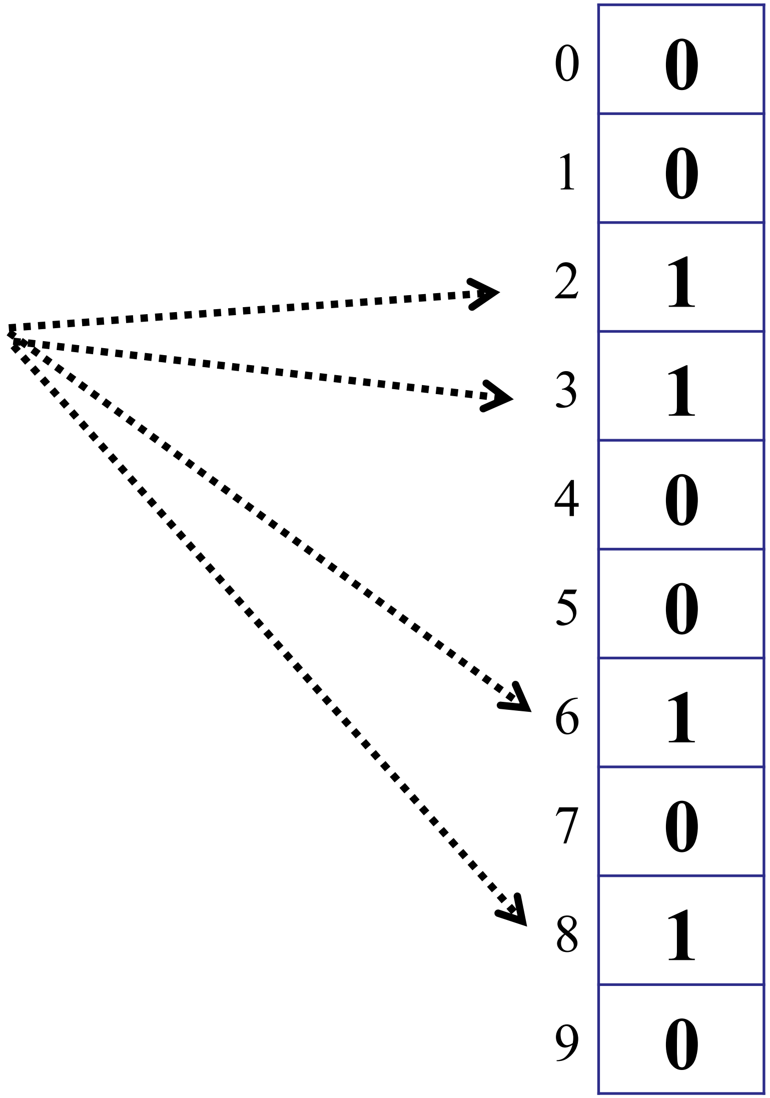


False positives rate vs. (m/n)

Bloom Filters

Use *k* hash functions!

hash(“`www.gmail.com`”)



Bloom Filter Analysis

Probability a given bit is 0:

$$\left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}$$

Bloom Filter Analysis

Probability a given bit is 0:

$$\left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}$$

Probability of a collision at one spot:

$$1 - e^{-kn/m}$$

* Assuming BOGUS fact that each table slot is independent…

Bloom Filter Analysis

Probability of a collision at one spot:

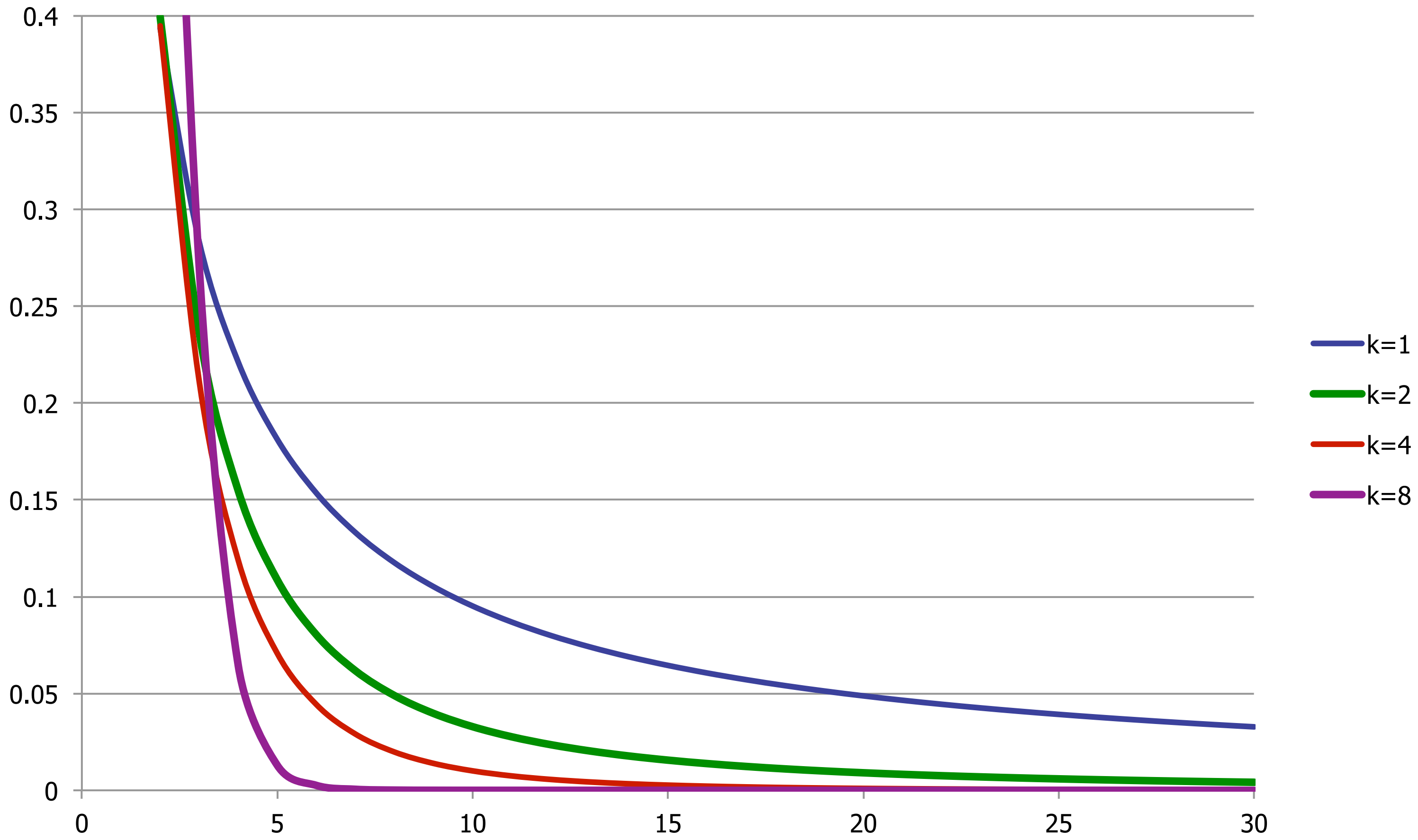
$$1 - e^{-kn/m}$$

Probability of a collision at all k spots:

$$\left(1 - e^{-kn/m}\right)^k$$

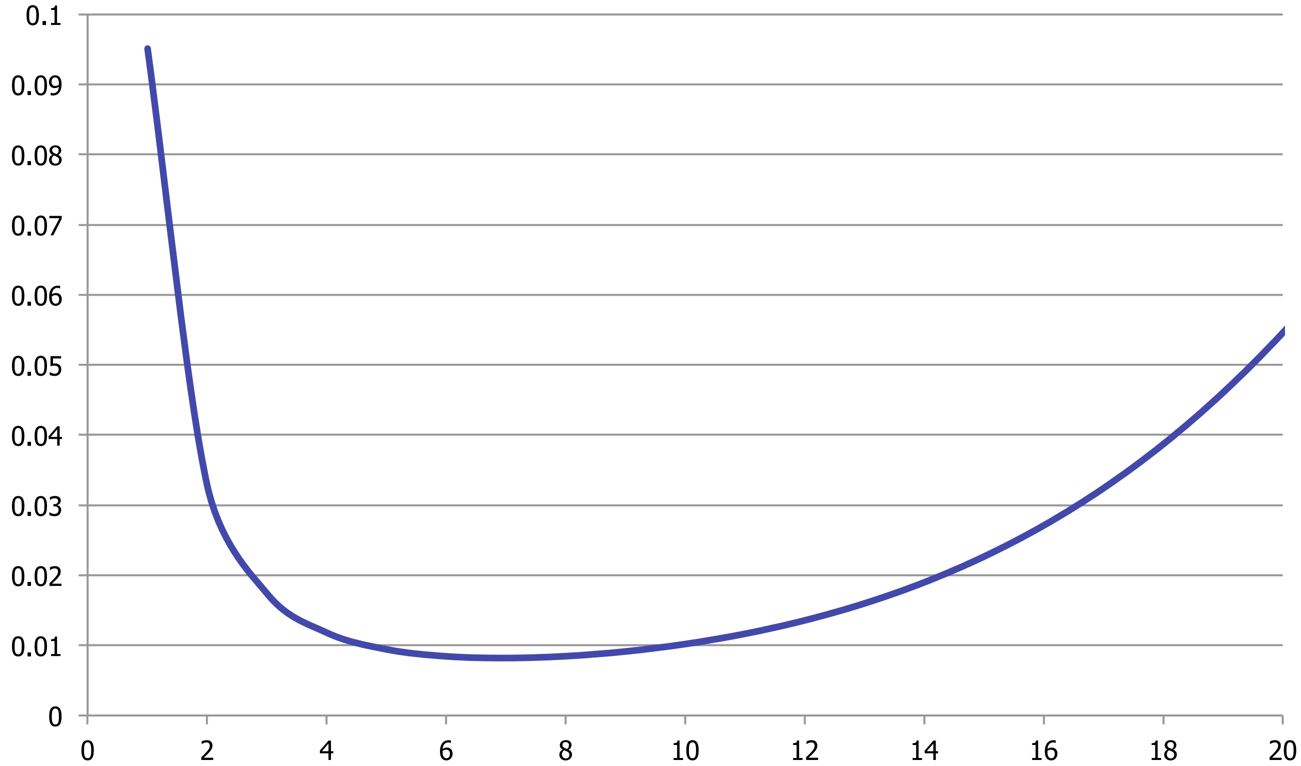
* Assuming BOGUS fact that each table slot is independent…

Bloom Filter



false positive rate vs. (m/n)

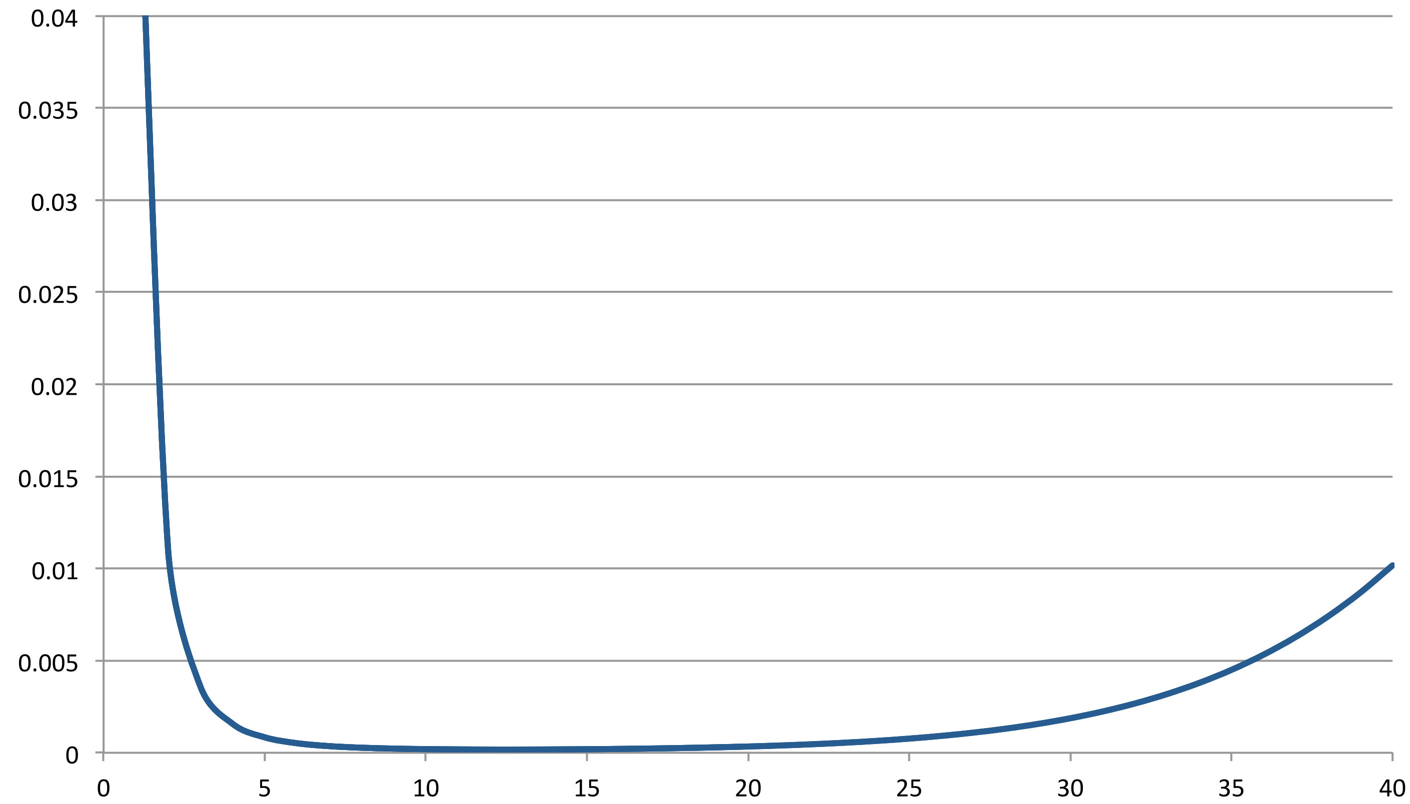
Bloom Filter



false positive rate vs *k*

m = 10n

Bloom Filter



false positive rate vs *k*

$m = 18n$

Bloom Filter

What is the optimal value of *k*?

- Probability of false positive:

$$\left(1 - e^{-kn/m}\right)^k$$

- Choose: $k = \frac{m}{n} \ln 2$

- Error probability: 2^{-k}

Summary So Far

- Fingerprint Hash Functions
  - Don't store the key.
  - Only store 0/1 vector.
- Bloom Filter
  - Use more than one hash function.
  - Redundancy reduces collisions.
- Probability of Error
  - False positives
  - False negatives

Fingerprint Hash Table

What about deletion?

insert("www.gmail.com")

insert("www.apple.com")

delete("www.gmail.com")

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

Bloom Filters

What about deleting an element?

- Store counter instead of 1 bit.
- On insert: increment.
- On delete: decrement.

Beware:

- If counter is big, then no space saving.
- If collisions are rare, counter is small: only a few bits.

Bloom Filters

Implementation of Set ADT:

- insert: O(k)
- delete: O(k)
- query: O(k)

Bloom Filters

Implementation of Set ADT:

- intersection
  - Bitwise AND of two Bloom filters
  - Time: O(m)

0	**0**	**&**	**0**
1	**0**	**&**	**1**
2	**0**	**&**	**0**
3	**1**	**&**	**1**
4	**0**	**&**	**0**
5	**0**	**&**	**0**
6	**1**	**&**	**0**
7	**0**	**&**	**0**
8	**1**	**&**	**1**
9	**0**	**&**	**0**

Bloom Filters

Implementation of Set ADT:

- intersection
  - Bitwise AND of two Bloom filters: O(m)
- union
  - Bitwise OR of two Bloom filters: O(m)

Other applications

- Chrome browser safe-browsing
  - Maintains list of “bad” websites.
  - Occasionally retrieves updates from google server.

- Spell-checkers
  - Storing all words takes a lot of space.
  - Instead, store a Bloom filter of the words.

- Weak password dictionaries

Summary

When to use Bloom Filters?

- Storing a set of data.
- Space is important.
- False positives are ok.

Interesting trade-offs:

- Space
- Time
- Error probability

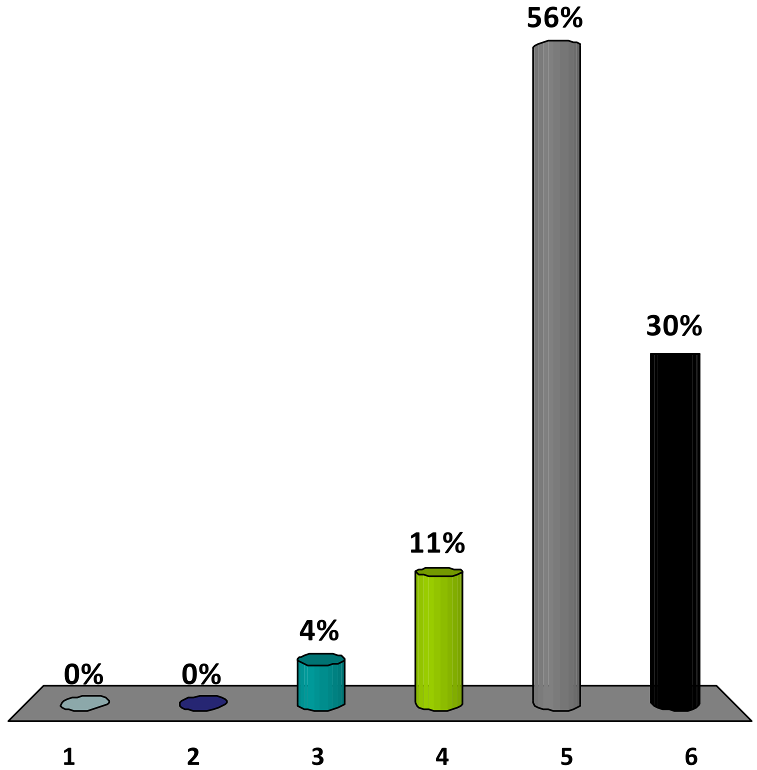
Today: Hash Tables (continued)

- Table (re)sizing
  - Proper hash table size
  - Amortized analysis
- Sets
  - Hash table sets
  - Bloom Filters

Application: DNA Analysis

How similar is Chimpanzee DNA to Human DNA?

1. 20-50%
2. 70-79%
3. 80-90%
4. 80-95%
5. 96-99%
6. Who are you calling a chimp, chump?



Response Counter

DNA Analysis

Question:

- How similar is chimp DNA to human DNA?
- Problem:
  - Given human DNA string: ACAAGCGGTAA
  - Given chimp DNA string: CCAAGGGGTAA
  - How similar are they?

- Similarity = longest common substring
  - Implies a gene that is shared by both.
  - Count genes that are shared by both.

DNA Analysis

How similar is Chimp and Human DNA?

- Longest common substring (text):

  ALGORITHM vs. ARITHMETIC

DNA Analysis

Naïve Algorithm: strings *A* and *B*

L = length(*A*);

for (L = n down to 1):

for every substring X1 of A of length L:

for every substring X2 of B of length L:

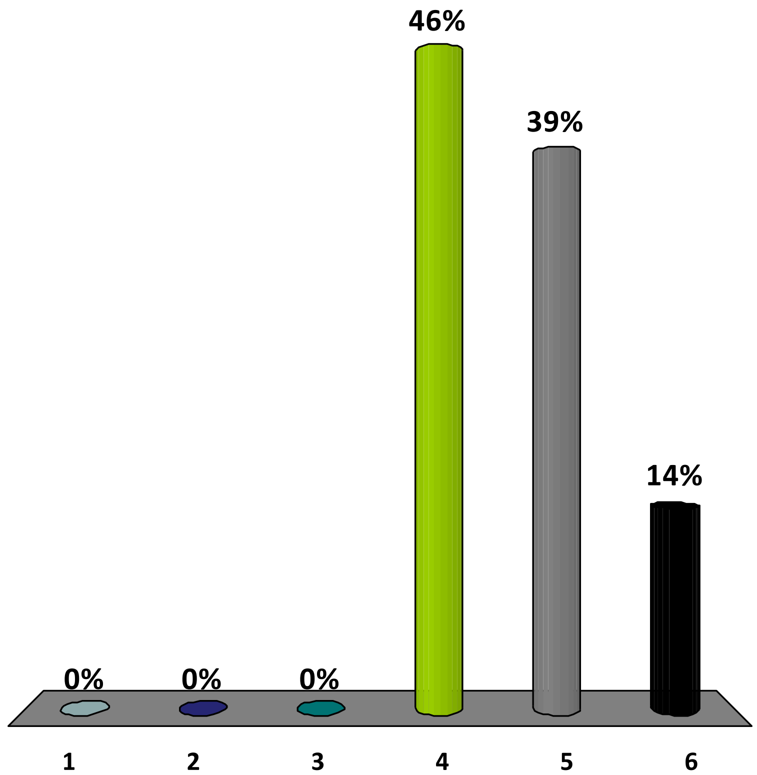
if (X1==X2) then return X1;

Example: ALGORITHM   ARITHMETIC

– L=3 : X1= ALG → compare to ARI, RIT, ITH, …

What is the running time?

1. $O(\log n)$
2. $O(n)$
3. $O(n \log n)$
4. $O(n^2)$
5. $O(n^3)$
6. $O(n^4)$



DNA Analysis

Naïve Algorithm: strings *A* and *B*

L = length(*A*);

for (L = n down to 1)

Loop *n* times.

for every substring X1 of A of length L:

n-L substrings

for every substring X2 of B of length L:

if (X1==X2) then return X1;

comparison costs: *O(L)*

n-L substrings

Total cost: $O(n^4)$

DNA Analysis

How similar is Chimp and Human DNA?

- Longest common substring (text):

  ALGORITHM vs. ARITHMETIC

- Another idea:
  - Binary search!
  - Don't search every length L.
  - Start with L = length(A) / 2.
  - Search until you find a match for some length L.

DNA Analysis

Binary Search Algorithm: strings *A* and *B*

```
L = length(A) /2;
repeat until done:
    for every substring X1 of A of length L:
        for every substring X2 of B of length L:
            if (X1==X2) then found=true;
    if (found) then increase L
    else decrease L
```

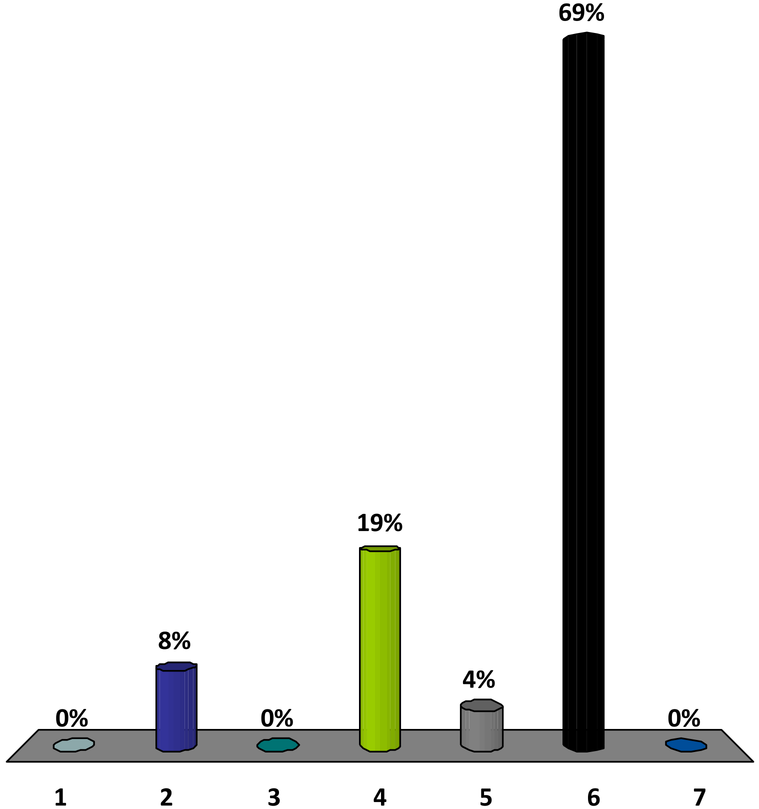
DNA Analysis

Binary Search Algorithm: strings *A* and *B*

```
low = 0;
high = length(A)+1;
repeat until (low >= high-1):
        L = low + (high-low)/2;
        found = substring(A, B, L);
        if (found) then low = L;
        else high = L;
return low;
```

What is the running time?

1. $O(n)$
2. $O(n \log n)$
3. $O(n^2)$
4. $O(n^2 \log n)$
5. $O(n^3)$
6. $O(n^3 \log n)$
7. $O(n^4)$



DNA Analysis

Binary Search Algorithm: strings *A* and *B*

substring(A, B, L)

for every substring X1 of A of length L: *n-L* substrings

for every substring X2 of B of length L: *n-L* substrings

if (X1==X2) then return true;

comparison costs*: O(L)*

return false

Cost: $O(n^3)$

DNA Analysis

Binary Search Algorithm: strings *A* and *B*

```
low = 0;
high = length(A)+1;
repeat until (low >= high-1):
        L = low + (high-low)/2;
        found = substring(A, B, L);
        if (found) then low = L;
        else high = L;
return low;
```

Cost: $O(n^3 \log n)$

DNA Analysis

How similar is Chimp and Human DNA?

- Longest common substring (text):

  ALGORITHM vs. ARITHMETIC

- Another idea:
  - Put every substring from first string into a symbol table.
  - Lookup every substring from second string in the symbol table.

DNA Analysis

How similar is Chimp and Human DNA?

- Long common substring (text):

  ALGORITHM vs. ARITHMETIC

- Add to symbol table:
  - A, AL, ALG, ALGO, ALGOR, ALGORI, ALGORIT, ALGORITH, …
  - L, LG, LGO, LGOR, LGORI, LGORIT, LGORITH, LGORITHM
  - G, GO, GOR, GORI, GORIT, GORITH, GORITHM
  - …

DNA Analysis

How similar is Chimp and Human DNA?

– Long common substring (text):

  ALGORITHM vs. ARITHMETIC

– Search in symbol table:

  - A, AR, ARI, ARIT, ARITH, ARITHM, ARITHME, ARITHMET, …
  - R, RI, RIT, RITH, RITHM, RITHME, RITHMET, RITHMETI, …
  - I, IT, ITH, ITHM, ITHME, ITHMET, ITHMETI, ITHMETIC
  - …

DNA Analysis

How similar is Chimp and Human DNA?

- Long common substring (text):

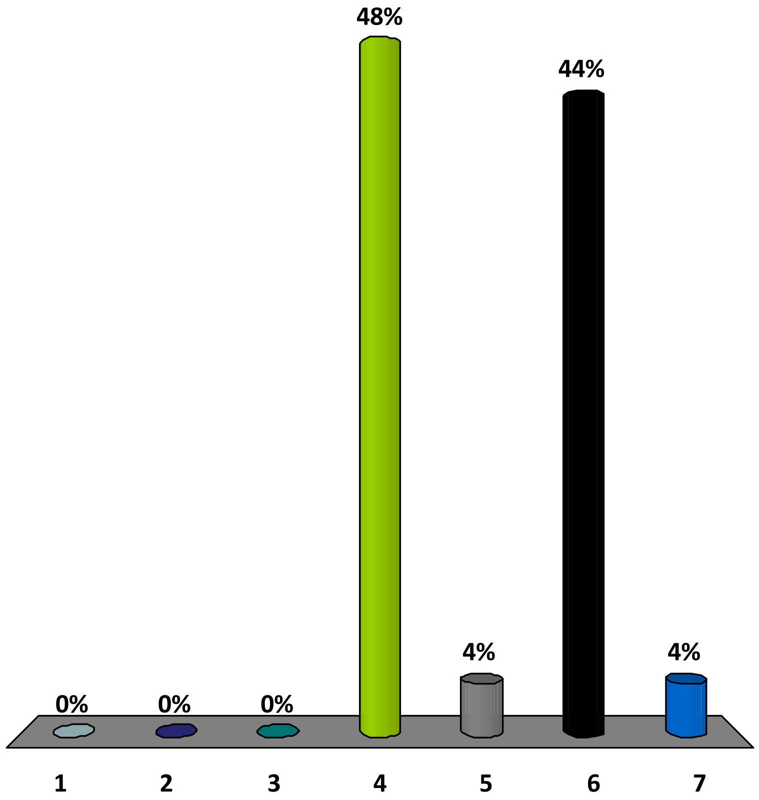
  ALGORITHM vs. ARITHMETIC

- Search in symbol table:
  - A, AR, ARI, ARIT, ARITH, ARITHM, ARITHME, ARITHMET, …
  - R, RI, RIT, RITH, RITHM, RITHME, RITHMET, RITHMETI, …
  - I, IT, ITH, ITHM, ITHME, ITHMET, ITHMETI, ITHMETIC
  - …

Assume insert/search are O(1). What is the running time of this algorithm?

1. O(1)
2. O(log n)
3. O(n log n)
4. O(n^2)
5. O(n^2 log n)
6. O(n^3)
7. O(n^3log n)

Response Counter



DNA Analysis

How similar is Chimp and Human DNA?

- Long common substring (text):

  ALGORITHM vs. ARITHMETIC

- There are $O(n^2)$ substrings.
- To add a substring of length k takes time $O(k)$:
  - To add the substring to the symbol table, you have to at least read the whole string!
- Total running time: $O(n^3)$

DNA Analysis

How similar is Chimp and Human DNA?

- Longest common substring (text):

  ALGORITHM vs. ARITHMETIC

- Focus on one length at a time:

  For a given length L:

  - Put every substring from first string into a symbol table.
  - Lookup every substring from second string in the symbol table.

  Binary search on length L.

DNA Analysis

Binary Search Algorithm: strings *A* and *B*

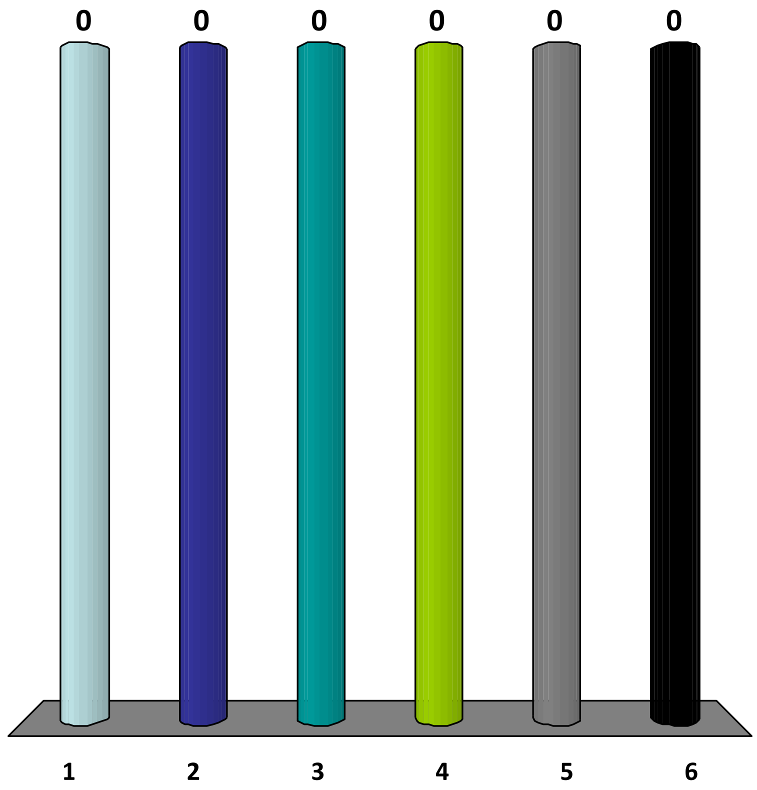
```
substring(A, B, L)
    for every substring X1 of A of length L:
        Add X1 to the symbol table.
    for every substring X2 of B of length L:
        if X2 is in the symbol table then return true;
    return false;
```

The performance of

`substring(X1, X2, L)`

on strings of length n is:

1. O(1)
2. O(log n)
3. O(n)
4. O(n^2)
5. O(n^2log(n))
6. O(n^3)



Response Counter

DNA Analysis

Binary Search Algorithm: strings *A* and *B*

substring(A, B, L)

for every substring X1 of A of length L: *n* substrings

Add X1 to the symbol table. cost to add: *n*

for every substring X2 of B of length L: *n* substrings

if X2 is in the symbol table **then** return true;

return false;

cost to search: *n*

Cost: $O(Ln) = O(n^2)$

DNA Analysis

How similar is Chimp and Human DNA?

- Longest common substring (text):

  ALGORITHM vs. ARITHMETIC

- Now, binary search again:
  - For $\log n$ values of length L:
    - Add all $O(n)$ substrings of length L from A.
    - Search all $O(n)$ substrings of length L from B.
    - Adjust L and continue.
  - Running time: $O(n^2 \log n)$.

DNA Analysis

Two problems:

1. Calculating hash of string of length L:
   - Takes O(L) time.
   - There are n-L substrings of length L.
   - Time: O(nL)

DNA Analysis

Two problems:

2. Searching for string of length L in hash table:
   - Entry e = T.lookup(X2)
   - Is entry e for string X2?
   - Or is entry e for a string X3 where h(X2) = h(X3)?
   - Verifying Entry e takes O(L) time.

Each false positive costs O(L)!

DNA Analysis

In order to speed up `substring`:

1. Reduce false positives
   – If the hash is in the table, then it is very likely that the string is in the hash table.

2. Compute hash faster
   – It is too slow to re-compute the hash function (n – L) times.

Faster substring matching

Reduce false positives:

- Use two different hash functions.
  - $h_1 : U \rightarrow \{1..m\}, m < 4n$.
  - $h_2 : U \rightarrow \{1..n^2\}$.

- Using a hash function as a *signature.*
  - A hash of a large data structure gives a small signature.
  - Example:
    - Are two databases identical?
    - Compare hash!
  - A hash as a fingerprint.

Faster substring matching

Reduce false positives:

– Use two different hash functions.

- $h_1 : U \rightarrow \{1..m\}$, $m < 4n$.
- $h_2 : U \rightarrow \{1..n^2\}$.

hash-insert(s):

Table[$h_1(s)$].LinkListInsert($h_2(s)$, s)

Faster substring matching

Reduce false positives:

- Use two different hash functions.
  - $h_1 : U \rightarrow \{1..m\}, m < 4n$.
  - $h_2 : U \rightarrow \{1..n^2\}$.

hash-lookup(s):

if (Table[$h_1(s)$] != null) then

(sig, t) = Table[$h_1(s)$]

if ($h_2(s) == sig$) then

if ($s == t$) then return true;

Faster substring matching

Analysis: hash-lookup(s)

- Case 1: string s is in table: O(L)
- Case 2: Table[$h_1(s)$] == null: O(1)
- Case 3: Table[$h_1(s)$] != null: ??

hash-lookup(s):

if (Table[$h_1(s)$] != null) then

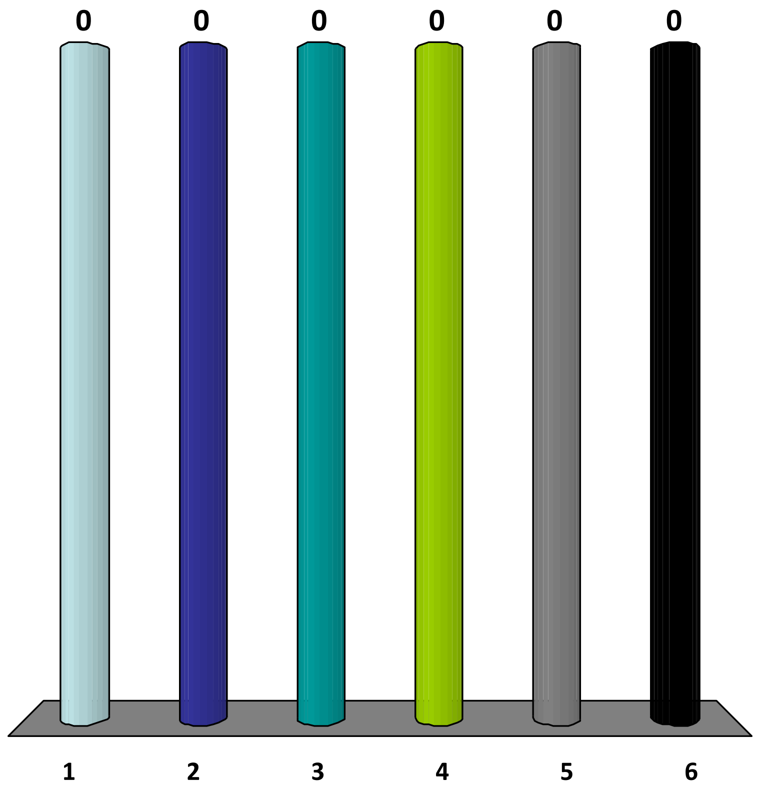
(sig, t) = Table[$h_1(s)$]

if ($h_2(s) == sig$) then

if ($s == t$) then return true;

Let $h_2 : U \rightarrow \{1..n^2\}$ be a hash function. For strings s and t, what is the probability that $h_2(s) == h_2(t)$?

1. $1/n$
2. $2/n$
3. $1/n^2$
4. $1/\sqrt{n}$
5. $1/2$
6. None of the above.



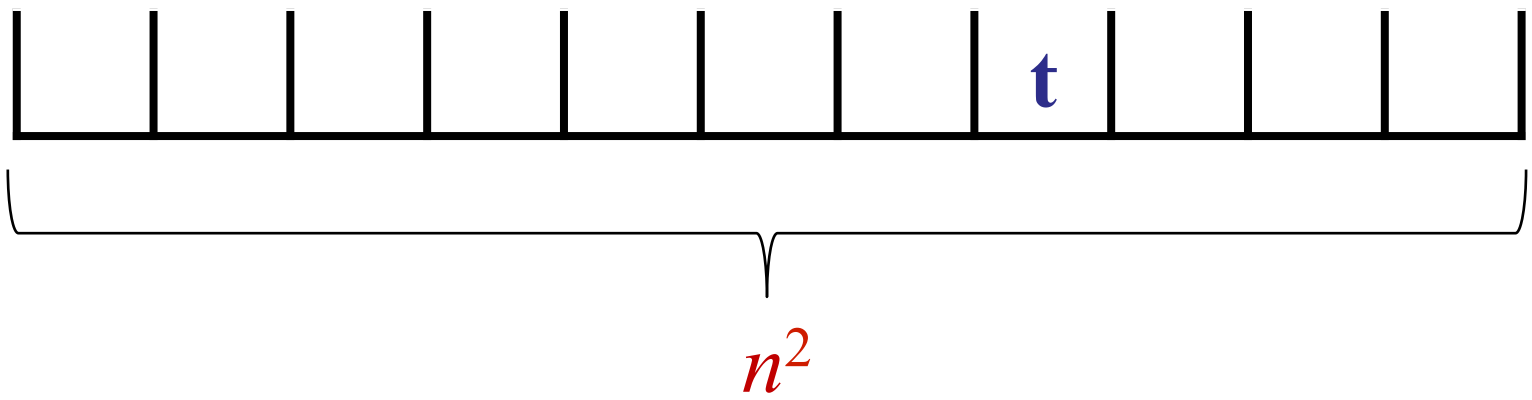
Response Counter

Faster substring matching

Analysis: hash-lookup(s) (Assume SUHA.)

- $h_2 : U \rightarrow \{1..n^2\}$
- For two strings s and t:

  Probability($h_2(s) == h_2(t)$): $1/n^2$



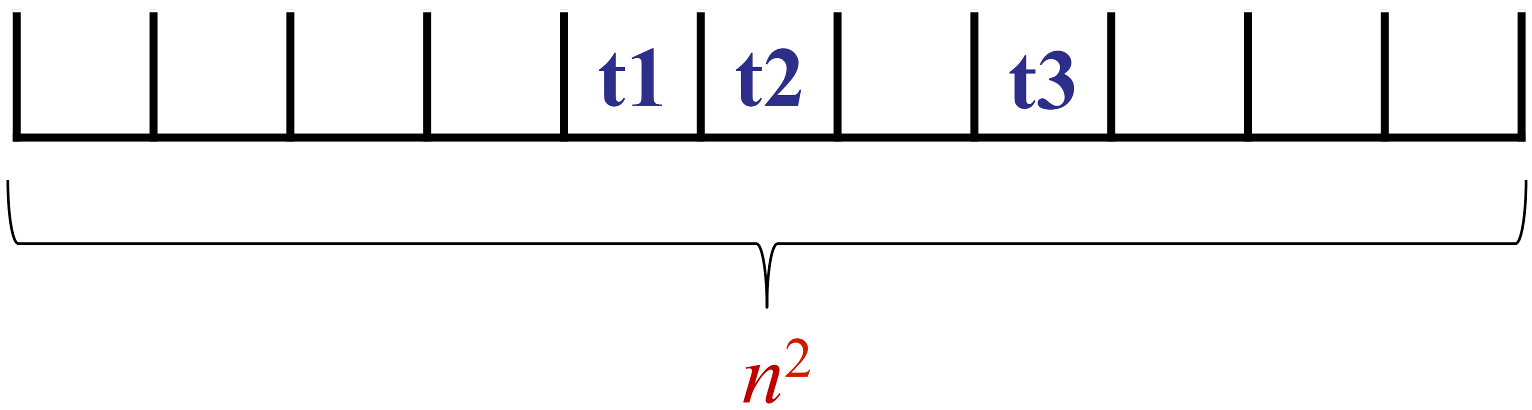
Faster substring matching

Analysis: hash-lookup(s) (Assume SUHA.)

- $h_2 : U \rightarrow \{1..n^2\}$

- For string s:

  Probability($h_2(s) == h_2(t)$ for any string t): $n/n^2 \leq 1/n$



Faster substring matching

Analysis: hash-lookup(s)

- Case 1: string s is in table: O(L)
- Case 2: Table[$h_1(s)$] == null: O(1)
- Case 3: Table[$h_1(s)$] != null: O(1 + L/n)

hash-lookup(s):

if (Table[$h_1(s)$] != null) then

(*sig*, t) = Table[$h_1(s)$]

if ($h_2(s)$ == *sig*) then

with probability ≤ 1/n

if (s == t) then return true;

Cost: O(L).

Faster substring matching

Analysis: hash-lookup(s)

- Case 1: string s is in table: O(L)
- Case 2: Table[$h_1(s)$] == null: O(1)
- Case 3: Table[$h_1(s)$] != null: O(1 + L/n) = O(1)

hash-lookup(s):

if (Table[$h_1(s)$] != null) then

  (sig, t) = Table[$h_1(s)$]

  if ($h_2(s)$ == sig) then E[cost] = n/L = O(1)

    if (s == t) then return true;

Faster substring matching

Analysis:

- Size of signature.
  - $h_2 : U \rightarrow \{1..n^2\}$.
  - $\log(n^2) = 2\log(n)$

- Assume that we can read/write/compare $\log(n)$ bits in time $O(1)$.
  - Why? A machine word is > $\log(n)$.

- Cost of comparing two signatures = $O(1)$.

DNA Analysis

Binary Search Algorithm: strings *A* and *B*

substring(A, B, L)

for every substring X1 of A of length L: *n* substrings

Add X1 to the symbol table. cost to add: *n*

for every substring X2 of B of length L: *n* substrings

if X2 is in the symbol table **then** return true;

return false;

cost to search: *n*

Cost: $O(Ln) = O(n^2)$

DNA Analysis

In order to speed up `substring`:

1. Reduce false positives
   - Use second hash function as a signature.
   - Reduce cost of collisions.

2. Compute hash faster
   - It is too slow to re-compute the hash function (n – L) times.

Rolling Hash Function

Abstract data type:

- insert(s) : sets string equal to string s
- delete-first-letter()
- append-letter(c)
- hash() : returns hash of current string

Rolling Hash Function

Example:

- insert(“arith”)

  string == “arith”

- hash() → 17
- delete-first-letter()

  string == “rith”

- hash() → 47
- append-letter(‘m’)

  string == “rithm”

- hash() → 4

Rolling Hash Function

Costs:

- insert(s) : O(|S|)
- delete-first-letter() : O(1)
- append-letter(c) : O(1)
- hash() : O(1)

Example: arithmetic

- insert("arith") : 5
- delete-first-letter(), append-letter(m) : O(1)

  string == "rithm"
- delete-first-letter(), append-letter(e) : O(1)

  string == "ithme"
- delete-first-letter(), append-letter(t) : O(1)

  string == "thmet"
- delete-first-letter(), append-letter(i) : O(1)

  string == "hmeti"
- delete-first-letter(), append-letter(c) : O(1)

  string == "metic"

Conclusion: $n - L = 6$ hashes for cost $10 = O(n)$.

Longest Common Substring

substring(X1, X2, L)

1. *rollhash*.insert(X1[0 : L-1])
2. for (i = 0 to n – L – 1) do:
3. T.hash-insert(*rollhash*.hash(), i))
4. rollhash.delete-first-letter()
5. rollhash.append-letter(X1[i + L])
6. …

Longest Common Substring

substring(X1, X2, L)

1. *rollhash*.insert(X1[0 : L-1])

2. for (i = 0 to n – L – 1) do:

3. T.hash-insert(*rollhash*.hash(), i))

4. rollhash.delete-first-letter()

5. rollhash.append-letter(X1[i + L])

6. …

Loop $n – L$ times.

Insert: O(1)

Update hash: O(1).

Total cost: $O(n – L + L) = O(n)$

Longest Common Substring

substring(X1, X2, L)

1. …
2. *rollhash*.insert(X2[0 : L])
3. for (i = 0 to n – L – 1) do:
4. if (T.hash-lookup(*rollhash*.hash() , s)) then
5. return true.
6. rollhash.delete-first-letter()
7. rollhash.append-letter(X1[*i* + L])

Longest Common Substring

substring(X1, X2, L)

1. …
2. *rollhash*.insert(X2[0 : L])
3. for (i = 0 to n – L – 1) do:
4. if (T.hash-lookup(*rollhash*.hash() , s)) then
5. return true.
6. rollhash.delete-first-letter()
7. rollhash.append-letter(X1[*i* + L])

Loop *n* – *L* times.

Lookup: E[cost] = 1 + L/*n*

Update hash: O(1).

Total cost: $O((n - L)(1 + L/n) + L) = O(n)$

Rolling Hash Function

Abstract data type:

- insert(s) : sets string equal to string s
- delete-first-letter()
- append-letter(c)
- hash() : returns hash of current string

Rolling Hash

Basic idea:

- Initially (on “insert”), calculate hash of string.
- Whenever the string is updated, update the hash.
- When a hash() is requested, output the pre-computed hash.

Rolling Hash

Step 1: Represent a string as a number

- Assume all letters in a string are 8-bit chars.
- Given a sequence of letters:

  $c_{L-1}\ c_{L-2} \ldots c_1\ c_0$

- Define: 8L bit integer

  $$s = \underbrace{00101001}_{c_{L-1}}\ \underbrace{10110111}_{c_{L-2}}\ \ldots\ \underbrace{10010000}_{c_1}\ \underbrace{10010000}_{c_0}$$

Rolling Hash

Step 1: Represent a string as a number

- Assume all letters in a string are 8-bit chars.
- Given a sequence of letters:

$c_{L-1}\ c_{L-2} \ldots c_1\ c_0$

- Define: 8L bit integer

$$s = \underbrace{00101001}_{c_{L-1}}\ \underbrace{10110111}_{c_{L-2}} \ldots \underbrace{10010000}_{c_1}\ \underbrace{10010000}_{c_0}$$

$$s = \sum_{i=0}^{L-1} c_i \cdot 2^{8i}$$

Rolling Hash

Step 1: Represent a string as a number

- Assume all letters in a string are 8-bit chars.
- Given a sequence of letters:

$$c_{L-1}\ c_{L-2}\ \ldots\ c_1\ c_0$$

- Define: 8L bit integer

$$s = \underbrace{00101001}_{c_{L-1}}\ \underbrace{10110111}_{c_{L-2}}\ \ldots\ \underbrace{10010000}_{c_1}\ \underbrace{10010000}_{c_0}$$

$$s = \sum_{i=0}^{L-1} c_i \cdot 2^{8i} = \sum_{i=0}^{L-1} c_i \ll 8i$$

Rolling Hash

Step 2: Updating the string

Deleting character c_{L-1}:

$$
\begin{array}{rrrrr}
s = 00101001 & 10110111 & \ldots & 10010000 & 10010000 \\
-\,00101001 & 00000000 & \ldots & 00000000 & 00000000 \\
\hline
 & 10110111 & \ldots & 10010000 & 10010000
\end{array}
$$

Rolling Hash

Step 2: Updating the string

Deleting character c_{L-1}:

$$\begin{array}{rl} s = & 00101001\ \ 10110111\ \ \ldots\ 10010000\ \ 10010000 \\ - & 00101001\ \ 00000000\ \ \ldots\ 00000000\ \ 00000000 \\ \hline & \ \ 10110111\ \ \ldots\ 10010000\ \ 10010000 \end{array}$$

$$s = s - c_{L-1} \cdot 2^{8(L-1)}$$

$$= s - c_{L-1} \ll 8(L-1)$$

Multiplication: O(1)

Shift: O(1)

Subtraction: O(1)

Rolling Hash

Step 2: Updating the string

Appending character c:

$$
\begin{array}{rr}
s = & 00000000 \;\; 10110111 \;\; \ldots \; 10010000 \;\; 10010000 \\
* & 1 \;\; 00000000 \\
\hline
 & 10110111 \;\; \ldots \; 10010000 \;\; 10010000 \;\; 00000000
\end{array}
$$

Rolling Hash

Step 2: Updating the string

Appending character c:

$$
\begin{array}{rr}
s = & 00000000 \;\; 10110111 \;\; \ldots \; 10010000 \;\; 10010000 \\
* & 1 \;\; 00000000 \\
\hline
 & 10110111 \;\; \ldots \; 10010000 \;\; 10010000 \;\; 00000000 \\
+ & 10101101 \\
\hline
 & 10110111 \;\; \ldots \; 10010000 \;\; 10010000 \;\; 10101101
\end{array}
$$

Rolling Hash

Step 2: Updating the string

Appending character c:

$$
\begin{array}{rr}
s = & 00000000\ \ 10110111\ \ \ldots\ 10010000\ \ 10010000 \\
* & 1\ \ 00000000 \\
\hline
 & 10110111\ \ \ldots\ 10010000\ \ 10010000\ \ 00000000 \\
+ & 10101101 \\
\hline
 & 10110111\ \ \ldots\ 10010000\ \ 10010000\ \ 10101101
\end{array}
$$

$$
\begin{aligned}
s &= s * 2^8 + c \\
&= (s \ll 8) + c
\end{aligned}
$$

Shift, addition: O(1)

Rolling Hash

Step 3: The Hash Function

The Division Method

$$h(s) = s \bmod p$$

Rolling Hash

Step 3: The Hash Function

The Division Method

$$h(s) = s \bmod p$$

Appending a character:

$$\mathrm{h}(\mathrm{s} \ll 8 + \mathrm{c})$$

Rolling Hash

Step 3: The Hash Function

The Division Method

$$h(s) = \ s \bmod p$$

Appending a character: O(1)

$$\begin{aligned} &\mathrm{h}(\mathrm{s} \ll 8 + \mathrm{c}) \\ &= [(\mathrm{s} \ll 8) + \mathrm{c}] \bmod \mathrm{p} \\ &= [(\mathrm{s} \bmod p) \ll 8) \bmod p + \mathrm{c}] \bmod \mathrm{p} \\ &= [\mathrm{h}(\mathrm{s}) \ll 8 + \mathrm{c}] \bmod \mathrm{p} \end{aligned}$$

Rolling Hash

Step 3: The Hash Function

The Division Method

$$h(s) = s \bmod p$$

Deleting the first character:

$$h\left(s - \left(c_{L-1} \ll 8(L-1)\right)\right)$$

Rolling Hash

Step 3: The Hash Function

The Division Method

$$h(s) = s \bmod p$$

Deleting the first character: O(1)

$$h\left(s - \left(c_{L-1} \ll 8(L-1)\right)\right)$$

$$= [h(s) - (c_{L-1} \ll 8(L-1) \bmod p)] \bmod p$$

Rolling Hash Function

Costs:

- insert(s) : O(|S|)
- delete-first-letter() : O(1)
- append-letter(c) : O(1)
- hash() : O(1)

DNA Analysis

Longest Common Substring

For any length L:

`substring`(X1, X2, L)

has cost $O(n)$.

Using binary search to find maximum value of L, we find the longest common substring in time:

$$O(n \log n)$$

DNA Analysis

Longest Common Substring

For any length L:

`substring`(X1, X2, L)

has cost $O(n)$.

Using binary search to find maximum value of L, we find the longest common substring in time:

$O(n \log n)$

The story continues… suffix-trees… $O(n)$….

DNA Analysis Summary

Using Hash Tables

- To get efficient algorithms, you have to be careful!
- Signatures…
  - Hash functions are useful as a “summary” of a longer / bigger document.
- Rolling hashes…
  - Fast way to calculate hashes in an incremental fashion.

Today: Hash Tables (continued)

- Table (re)sizing
  - Proper hash table size
  - Amortized analysis
- Sets
  - Hash table sets
  - Bloom Filters
- Application: DNA analysis
  - Longest Common Substring
  - Rolling hashes