CSE 312

Foundations of Computing II

Lecture 10: Bloom Filters; LOTUS

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

Anna R. Karlin

Slide Credit: Based on slides by Shreya Jayraman, Luxi Wang, Alex Tsun & myself ☺

Last Class:

- Linearity of Expectation

Today:

- An application: Bloom Filters!
- LOTUS

Kandinsky

Basic Problem

Problem: Store a subset S of a <u>large</u> set U.

Example. $U =$ set of 128 bit strings

$|U| \approx 2^{128}$

$S =$ subset of strings of interest

$|S| \approx 1000$

Two goals:

1. **Very fast** (ideally constant time) answers to queries "Is $x \in S$?"
2. **Minimal storage** requirements.

$x \in U$

Naïve Solution – Constant Time

Idea: Represent S as an array A with 2^{128} entries.

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

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

0	1	2	…	*K*	…		
1	0	1	0	1	…	0	0

Naïve Solution – Constant Time

Idea: Represent S as an array A with 2^{128} entries.

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

$S = \{0,2, \dots, \mathrm{K}\}$

0	1	2	…	*K*	…		
1	0	1	0	1	…	0	0

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

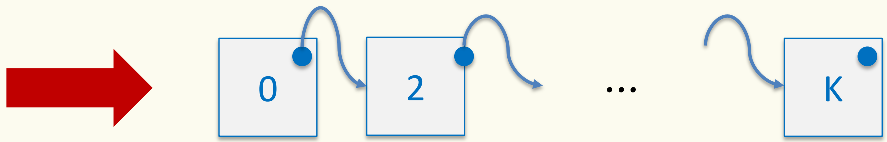
→ **constant time!** 👍 😀

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

Naïve Solution – Small Storage

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

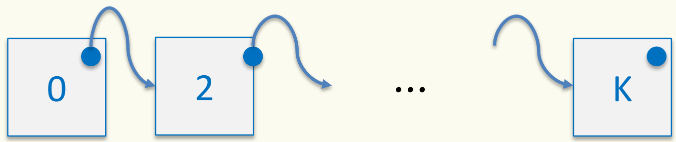
$S = \{0, 2, \ldots, K\}$



Naïve Solution – Small Storage

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

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



Storage: Grows with $|S|$ only

Membership test: Check $x \in S$ requires time linear in $|S|$

(Can be made logarithmic by using a tree)

Hash Table

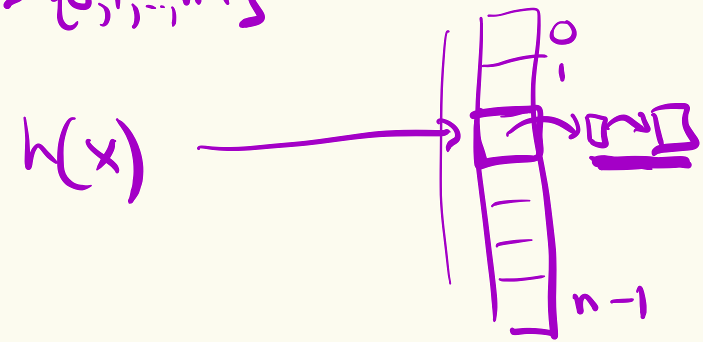
$|S| = n$

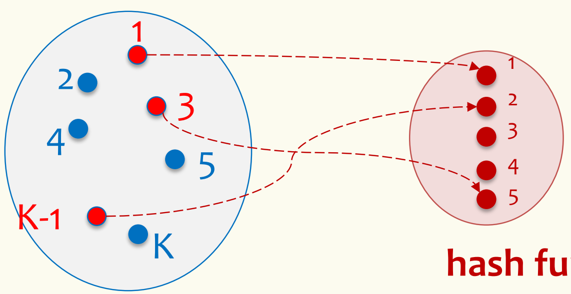
Idea: Map elements in S into an array A of size n using a hash function

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

$h : U \rightarrow \{0, 1, \ldots, n-1\}$

Storage: n elements (size of array)





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

Hash Table

Idea: Map elements in S into an array A using a hash function

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

Storage: n elements

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

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

Hashing: collisions

- **Collisions** occur when two elements of set map to the same location in the hash table.
- Common solution: chaining – at each location (bucket) in the table, keep linked list of all elements that hash there.

- **Want:** hash function that distributes the elements of S well across hash table locations. Ideally uniform distribution!

Hashing: summary

Hash Tables

- They store the data itself
- With a good hash function, the data is well distributed in the table and lookup times are small.
- However, they need at least as much space as all the data being stored
- E.g. storing strings, or IP addresses or long DNA sequences.

Bloom Filters: motivation

- Large universe of possible data items.
- Data items are large (say 128 bits or more)
- Hash table is stored on disk or across network, so any lookup is expensive.
- Many (if not nearly all) of the lookups return “Not found”.

Altogether, this is bad. You’re wasting **a lot of time and space** doing lookups for items that aren’t even present.

Bloom Filters: Motivation

- Large universe of possible data items.
- Hash table is stored on disk or in network, so any lookup is expensive.
- Many (if not most) of the lookups return "Not found".

Altogether, this is bad. You're wasting **a lot of time and space** doing lookups for items that aren't even present.

Example:

- **Network routers**: want to track source IP addresses of certain packets, .e.g., blocked IP addresses.

Bloom Filters

to the rescue

Bloom Filters: motivation (3)

- Probabilistic data structure.
- Close cousins of hash tables.
- Ridiculously space efficient
- To get that, make occasional errors, specifically false positives.

Bloom Filters

- Stores information about a set of elements.
- Supports two operations:
  1. **add(x)** - adds x to bloom filter
  2. **contains(x)** - returns true if x in bloom filter, otherwise returns false
     - If returns false, **definitely** not in bloom filter.
     - If returns true, **possibly** in the structure (some false positives).

Bloom Filters

- Why accept false positives?
  - **Speed** – both operations very very fast.
  - **Space** – requires a miniscule amount of space relative to storing all the actual items that have been added.

○ Often just 8 bits per inserted item!

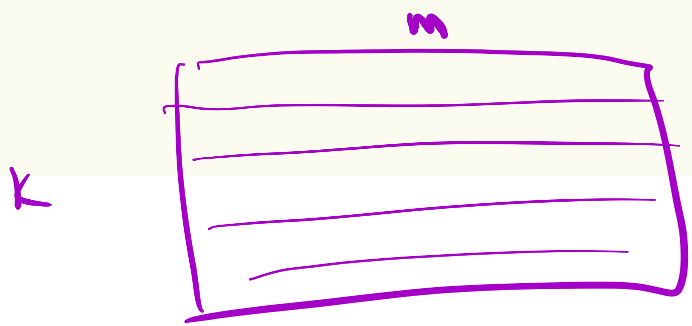
Bloom Filters: Initialization

Number of hash functions

Size of array associated to each hash function.

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

for each hash function, initialize an empty bit vector of size m



Bloom Filters: Example

bloom filter t with m = 5 that uses k = 3 hash functions

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

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

Bloom Filters: Add

$$
\begin{array}{l}
\textbf{function } \text{ADD}(x) \\
\quad \textbf{for } i = 1, \ldots, k\text{: } \textbf{do} \\
\quad\quad t_i[\underbrace{h_i(x)]}_{} = 1
\end{array}
$$

for each hash function h_i

Index into ith bit-vector, at index produced by hash function and set to 1

$h_i(x)$ → result of hash function h_i on x

Bloom Filters: Example

bloom filter t with m = 5 that uses k = 3 hash functions

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

$i = 1$

$h_1(x) = 2$

x

add("thisisavirus.com")

h_1("thisisavirus.com") → 2

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

Bloom Filters: Example

bloom filter t of length m = 5 that uses k = 3 hash functions

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

$i=2$

$h_2(x)=1$

add("thisisavirus.com")

h_1("thisisavirus.com") → 2

h_2("thisisavirus.com") → 1

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

Bloom Filters: Example

bloom filter t of length m = 5 that uses k = 3 hash functions

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

$i=3$

$h_3(x)=4$

add("thisisavirus.com")

h_1("thisisavirus.com") → 2

h_2("thisisavirus.com") → 1

h_3("thisisavirus.com") → 4

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

Bloom Filters: Example

bloom filter t of length m = 5 that uses k = 3 hash functions

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

add("thisisavirus.com")

h_1("thisisavirus.com") → 2
h_2("thisisavirus.com") → 1
h_3("thisisavirus.com") → 4

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

Bloom Filters: Example

bloom filter t with m = 5 that uses k = 3 hash functions

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

contains("thisisavirus.com")

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

Bloom Filters: Example

bloom filter t of length m = 5 that uses k = 3 hash functions

contains("thisisavirus.com")

h_1("thisisavirus.com") → 2

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

True

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

Bloom Filters: Example

bloom filter t of length m = 5 that uses k = 3 hash functions

contains("thisisavirus.com")

h_1("thisisavirus.com") → 2

h_2("thisisavirus.com") → 1

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

True   True

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

Bloom Filters: Example

bloom filter t of length m = 5 that uses k = 3 hash functions

contains(“thisisavirus.com”)

h_1(“thisisavirus.com”) → 2
h_2(“thisisavirus.com”) → 1
h_3(“thisisavirus.com”) → 4

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

True True True

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

Bloom Filters: Example

bloom filter t of length m = 5 that uses k = 3 hash functions

contains("thisisavirus.com")

h$_1$("thisisavirus.com") → 2
h$_2$("thisisavirus.com") → 1
h$_3$("thisisavirus.com") → 4

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

True True True

Since all conditions satisfied, returns True (correctly)

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

Bloom Filters: Contains

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

Returns True if the bit vector t_i for each hash function has bit 1 at index determined by $h_i(x)$, otherwise returns False

Bloom Filters: False Positives

bloom filter t of length m = 5 that uses k = 3 hash functions

y

add("totallynotsuspicious.com")

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

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

Bloom Filters: False Positives

bloom filter t of length m = 5 that uses k = 3 hash functions

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

add("totallynotsuspicious.com")

h_1("totallynotsuspicious.com") → 1

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

Bloom Filters: False Positives

bloom filter t of length m = 5 that uses k = 3 hash functions

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

add("totallynotsuspicious.com")

h_1("totallynotsuspicious.com") → 1

h_2("totallynotsuspicious.com") → 0

$h_1(y) = 1$

$h_2(y) = 0$

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

Bloom Filters: False Positives

bloom filter t of length m = 5 that uses k = 3 hash functions

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

add("totallynotsuspicious.com")

h_1("totallynotsuspicious.com") → 1

h_2("totallynotsuspicious.com") → 0

h_3("totallynotsuspicious.com") → 4

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

$h_1 : U \rightarrow \{0,1,2,3,4\}$

Bloom Filters: False Positives

bloom filter t of length m = 5 that uses k = 3 hash functions

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

add("totallynotsuspicious.com")

h_1("totallynotsuspicious.com") → 1

h_2("totallynotsuspicious.com") → 0

h_3("totallynotsuspicious.com") → 4

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

Collision, is already set to 1

Bloom Filters: False Positives

bloom filter t of length m = 5 that uses k = 3 hash functions

$$\textbf{function } \textsc{Add}(x)$$
$$\quad \textbf{for } i = 1, \ldots, k\text{: } \textbf{do}$$
$$\qquad t_i[h_i(x)] = 1$$

add("totallynotsuspicious.com")

h_1("totallynotsuspicious.com") → 1

h_2("totallynotsuspicious.com") → 0

h_3("totallynotsuspicious.com") → 4

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

Bloom Filters: Example

bloom filter t of length m = 5 that uses k = 3 hash functions

contains("verynormalsite.com")

2

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

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

Bloom Filters: Example

bloom filter t of length m = 5 that uses k = 3 hash functions

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

True

contains("verynormalsite.com")

h_1("verynormalsite.com") → 2

Index →	0	1	2	3	4
t_1	0	1	$\boxed{1}$	0	0
t_2	1	1	0	0	0
t_3	0	0	0	0	1

Bloom Filters: Example

bloom filter t of length m = 5 that uses k = 3 hash functions

contains("verynormalsite.com")

h_1("verynormalsite.com") → 2

h_2("verynormalsite.com") → 0

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

True True

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

Bloom Filters: Example

bloom filter t of length m = 5 that uses k = 3 hash functions

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

True True True

contains("verynormalsite.com")

h_1("verynormalsite.com") → 2
h_2("verynormalsite.com") → 0
h_3("verynormalsite.com") → 4

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

Bloom Filters: Example

bloom filter t of length m = 5 that uses k = 3 hash functions

contains("verynormalsite.com")

h_1("verynormalsite.com") → 2
h_2("verynormalsite.com") → 0
h_3("verynormalsite.com") → 4

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

True True True

Since all conditions satisfied, returns True (incorrectly)

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

Bloom Filters: Summary

- An empty bloom filter is an empty k x m bit array with all values initialized to zeros
  - k = number of hash functions
  - m = size of each array in the bloom filter
- add(x) runs in O(k) time
- contains(x) runs in O(k) time
- requires O(km) space (in bits!)
- Probability of false positives from collisions can be reduced by increasing the size of the bloom filter

Bloom Filters: Application

- Google Chrome has a database of malicious URLs, but it takes a long time to query.
- Want an in-browser structure, so needs to be efficient and be space-efficient
- Want it so that can check if a URL is in structure:
  - If return False, then definitely not in the structure (don't need to do expensive database lookup, website is safe)
  - If return True, the URL may or may not be in the structure. Have to perform expensive lookup in this rare case.

Comparison with Hash Tables - Space

n

- Google storing 5 million URLs, each URL 40 bytes.
- Bloom filter with ~~k = 30 and m = [illegible],500,000~~   m = 10,000,000   k = 8

Hash Table

5,000,000 x 40

= 200,000,000 B

Bloom Filter

$$\frac{8 \times 10{,}000{,}000}{8}$$ bits

10,000,000 B

5% of space

False pos rate   0.06%

0.0006

Comparison with Hash Tables - Time

in real hash table.

- Say avg user visits 102,000 URLs in a year, of which 2,000 are malicious.
- 0.5 seconds to do lookup in the database, 1ms for lookup in Bloom filter.
- Suppose the false positive rate is 3%

Time

Hash Table

102,000 x 0.5 sec

= 51,000 sec

Bloom Filter

$102,000 \times \frac{1}{1000} = 102$ secs

neg

pos

lookup in actual hashtable

2000 x 0.5 + 100,000 x 0.03 x 0.5

about 5%

2,602 sec

Bloom Filters: Many Applications

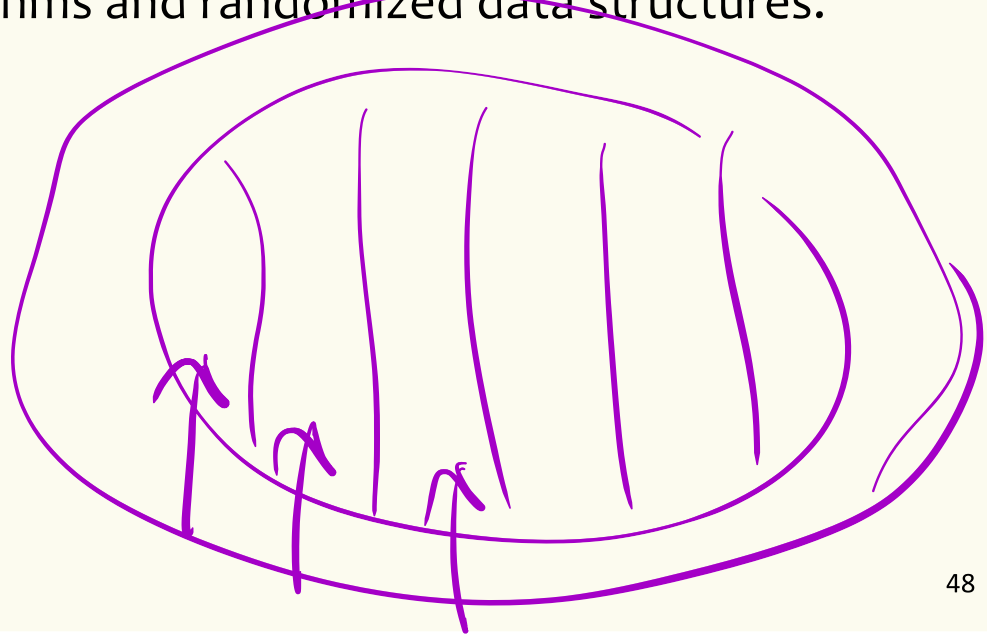
- Any scenario where space and efficiency are important.
- Used a lot in networking
- In distributed systems when want to check consistency of data across different locations, might send a Bloom filter rather than the full set of data being stored.
- Google BigTable uses Bloom filters to reduce disk lookups
- Internet routers often use Bloom filters to track blocked IP addresses.
- And on and on…

h(100) = 31

Bloom Filters typical of….

of randomized algorithms and randomized data structures.

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



Back to R.V.s… .

LOTUS

Law Of The Unconscious Statistician

Expectation of Random Variable

Definition. Given a discrete RV $X : \Omega \to \mathbb{R}$, the expectation or expected value of X is

$$\mathrm{E}[X] = \sum_{\omega \in \Omega} X(\omega) \cdot \Pr(\omega)$$

or equivalently

$$\mathrm{E}[X] = \sum_{x \in \Omega_X} x \cdot \Pr(X = x)$$

Intuition: "Weighted average" of the possible outcomes (weighted by probability)

Linearity of Expectation

Theorem. For **any** two random variables X and Y

$$\mathbb{E}(X+Y) = \mathbb{E}(X) + \mathbb{E}(Y).$$

Theorem. For any random variables $X_1, \dots, X_n$, and real numbers $a_1, \dots, a_n, c \in \mathbb{R}$,

$$\mathbb{E}(a_1 X_1 + \cdots + a_n X_n + c) = a_1 \mathbb{E}(X_1) + \cdots + a_n \mathbb{E}(X_n) + c.$$

Computing complicated expectations

Often boils down to the following three steps

- Decompose: Finding the right way to decompose the random variable into sum of simple random variables

$$X = X_1 + \cdots + X_n$$

- LOE: Observe linearity of expectation.

$$\mathbb{E}(X) = \mathbb{E}(X_1) + \cdots + \mathbb{E}(X_n).$$

- Conquer: Compute the expectation of each X_i

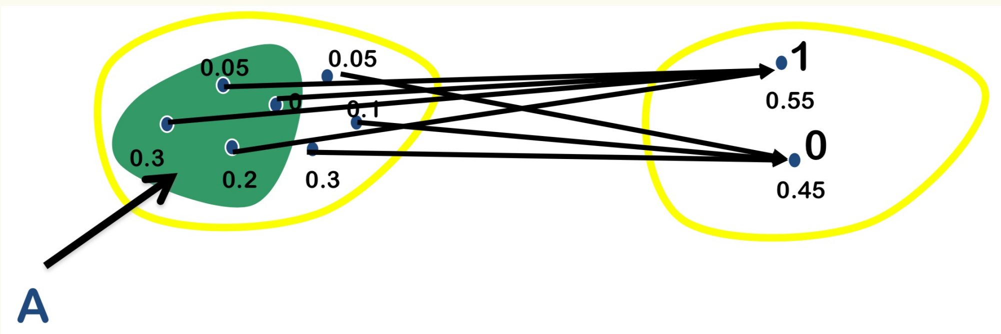
Often, X_i are indicator (0/1) random variables.

Indicator random variable

For any event A, can define the indicator random variable X

$$X = \begin{cases} 1 & \text{if event A occurs} \\ 0 & \text{if event A does not occur} \end{cases}$$

$$\mathbb{P}(X=1) = \mathbb{P}(\mathrm{A})$$
$$\mathbb{P}(X=0) = 1 - \mathbb{P}(\mathrm{A})$$



$$\boxed{\mathbb{E}(X) = \mathbb{P}(\mathrm{A})}$$

Example: Returning Homeworks

- Class with n students, randomly hand back homeworks. All permutations equally likely.
- Let X be the number of students who get their own HW
- Let $Y = (X^2+4)\ mod\ 8$.
- what is $\mathbb{E}(Y)$?

$\Pr(\omega)$	ω	$X(\omega)$
1/6	1, 2, 3	3
1/6	1, 3, 2	1
1/6	2, 1, 3	1
1/6	2, 3, 1	0
1/6	3, 1, 2	0
1/6	3, 2, 1	1

$Y(\omega)$

$(3^2+4) \bmod 8 = 5$

$(1^2+4) \bmod 8 = 5$

5

4

4

5

$\boxed{g(x) = (x^2+4) \bmod 8}$

$\Pr(Y=4) = \frac{1}{3}$

$\Pr(Y=5) = \frac{2}{3}$

$E(Y) = 5 \cdot \Pr(Y=5) + 4 \Pr(Y=4) = \boxed{\sum_{y \in \Omega_Y = \{4,5\}} y \Pr(Y=y)}$

$E(Y) = \sum_{x \in \Omega_X = \{0,1,3\}} g(x) \Pr(X=x) = (3^2+4) \bmod 8 \Pr(X=3) + (1^2+4) \bmod 8 \Pr(X=1) + (0^2+4) \bmod 8 \Pr(X=0)$

Linearity is special!

In general $\mathbb{E}(g(X)) \neq g(\mathbb{E}(X))$

E.g., $X = \begin{cases} \boxed{1} \text{ with prob } 1/2 \\ \boxed{-1} \text{ with prob } 1/2 \end{cases}$

- $\mathbb{E}(X^2) \neq \mathbb{E}(X)^2$

1 0

How DO we compute $\mathbb{E}(g(X))$?

$Y = g(X)$

$g(x) = x^2$

$E[g(X)] = E(X^2)$

$\boxed{X^2}$

1

1

$E(X) = 0$

$1 \cdot \frac{1}{2} + (-1) \cdot \frac{1}{2}$

$E(X^2) = 1$

Expectation of $g(X)$ (LOTUS)

Definition. Given a discrete RV $X: \Omega \to \mathbb{R}$, the expectation or expected value of $Y = g(X)$ is

$$\mathrm{E}[Y] = \sum_{\omega \in \Omega} g(X(\omega)) \cdot \Pr(\omega)$$

or equivalently

$$\mathrm{E}[Y] = \sum_{x \in \Omega_X} g(x) \cdot \Pr(X = x)$$

LOTUS

or equivalently

$$\mathrm{E}[Y] = \sum_{y \in \Omega_Y} y \cdot \Pr(Y = y)$$