

Hashing and Heavy Hitters

welcome to Randomized Algorithms

fundamentalalgorithms.com/randomized

Last compiled August 21, 2022 at 6:12pm.

Randomized Algorithms, CS588, Fall 2022

Lectures: 4:30 to 5:45 PM, on Tuesdays and Thursdays, in Lawson room 1106.

Staff:	Email (@purdue.edu)	Office hours
Kent Quanrud	krq	After class, LWSN 1211
Jawad A. Raheel (TA)	jraheel	

Please see the syllabus (page 306) for more information.

Homework

- Homework 0 due August 31.

Class schedule¹

1. August 23. Randomized SAT and quicksort (chapter 1).
2. August 25. Hashing and heavy hitters (chapter 2).

Highlights

midterm, final

biweekly homework

drop lowest 20% of HW scores

late/resubmit HW policy

office hours after every class

Please read the syllabus

- will ask for questions next class

Tuesday

Probability

Events, probabilities,
conditional probabilities,
union bound

Random variables,
expected value,
linearity of expectation

Markov's inequality

Algorithms

Approx. 3-SAT,

Quicksort,

Quickselect (HW)

3-SAT

"conjunctive normal form"

CNF formula

$$\varphi(x_1, \dots, x_n) = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_4 \vee \dots) \wedge \dots$$

m clauses, n variables

3 (distinct) variables per clause

The silliest APX algorithm

1. for $i = 1, \dots, n$

a. randomly set $x_i = \begin{cases} \text{TRUE} & \text{w/ prob .5} \\ \text{FALSE} & \text{w/ prob .5} \end{cases}$

We will:

① prove silly algo is $\frac{7}{8}$ -APX

② derandomize silly algo $\Rightarrow$ deterministic $\frac{7}{8}$ -APX

Even more surprising:

"probabilistically checkable proofs"

PCP Theorem

$(\frac{7}{8} + \epsilon)$ -APX is NP-Hard $\forall \epsilon > 0$

Approximating 3SAT: "Max 3-SAT"

$$\varphi(x_1, \dots, x_n) = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_4 \vee \dots) \wedge \dots$$

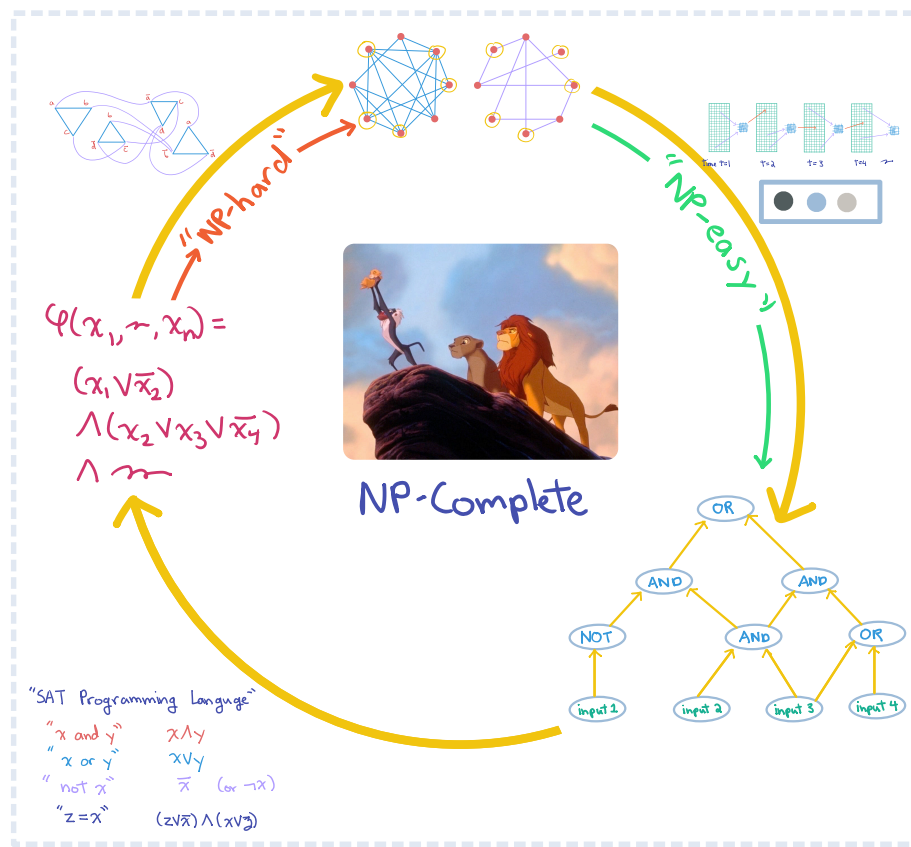
Goal: satisfy Max # clauses

Interesting even if φ is unsatisfiable

"d-Approximation algorithm" $d \in (0, 1)$

Algorithm satisfies $d \cdot \text{OPT}$ clauses

optimum (max) value



2 disjoint clauses

① prove silly algo is $\frac{7}{8}$ -APX

1. for $i=1, \dots, n$ The silliest APX algorithm
 a. randomly set $x_i = \begin{cases} \text{TRUE} & \text{w/ prob. } .5 \\ \text{FALSE} & \text{w/ prob. } .5 \end{cases}$

$$\mathcal{S}(x) = (\underbrace{x_1 \vee x_2 \vee \bar{x}_3}_{C_1}) \wedge (\underbrace{x_4 \vee \bar{x}_5 \vee x_6}_{C_2})$$

E_1 = event that C_1 satisfied

E_2 = ~ C_2 ~

$$\begin{aligned} \Pr[E_1, E_2] &= \frac{1}{8} \cdot \frac{1}{8} \\ &= \Pr[E_1] \cdot \Pr[E_2] \end{aligned}$$

2 clauses w/
1 common variable

① prove silly algo is $\frac{7}{8}$ -APX

1. for $i=1, \dots, n$ The silliest APX algorithm
 a. randomly set $x_i = \begin{cases} \text{TRUE} & \text{w/ prob. } .5 \\ \text{FALSE} & \text{w/ prob. } .5 \end{cases}$

$$\mathcal{S}(x) = (\underbrace{x_1 \vee \bar{x}_2 \vee x_3}_{C_1}) \wedge (\underbrace{x_3 \vee x_4 \vee \bar{x}_5}_{C_2})$$

E_1 = event that C_1 satisfied

E_2 = ~ C_2 ~

$$\Pr[E_1, E_2] =$$

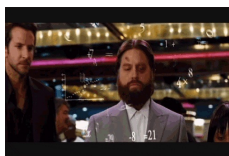
$$\underbrace{\Pr[E_1, E_2 | x_3 = T] \cdot \frac{1}{2}}_1 + \underbrace{\Pr[E_1, E_2 | x_3 = F] \cdot \frac{1}{2}}_{\left(\frac{3}{4}\right)^2}$$

m clauses w/
common variables

① prove silly algo is $\frac{7}{8}$ -APX

1. for $i=1, \dots, n$ The silliest APX algorithm
 a. randomly set $x_i = \begin{cases} \text{TRUE} & \text{w/ prob. } .5 \\ \text{FALSE} & \text{w/ prob. } .5 \end{cases}$

$$\mathcal{S}(x) = (\underbrace{x_1 \vee \bar{x}_2 \vee x_3}_{C_1}) \wedge (\underbrace{x_3 \vee x_4 \vee \bar{x}_5}_{C_2}) \wedge (\underbrace{x_2 \vee \bar{x}_4 \vee x_6}_{C_3}) \wedge \dots$$



E_1 = event that C_1 satisfied

E_2 = ~ C_2 ~

E_3 = ~ C_3 ~

$\vdots$

general question

how to analyze a randomized mechanism
w/ a combinatorial explosion of possibilities?

Big idea: we only want the average

Linearity of Expectation

$$E[X+Y] = E[X] + E[Y] \quad (\text{ridiculously useful})$$

"average sum = sum of averages"

let $Y_i = \begin{cases} 1 & \text{if } i\text{th clause satisfied} \\ 0 & \text{otherwise} \end{cases}$

($i=1, \dots, m$)

$$E[Y_i] = \frac{7}{8} \quad (1 \text{ clause case})$$

$$E[\text{\# clauses satisfied}] = E\left[\sum_{i=1}^m Y_i\right] = \sum_{i=1}^m E[Y_i] = \frac{7}{8}m$$

(actually easy!)

let $Y_i = \begin{cases} 1 & \text{if } i\text{th clause satisfied} \\ 0 & \text{otherwise} \end{cases}$

$$Z = \sum_{i=1}^m Y_i = \# \text{ clauses satisfied}$$

$$E[Z] = \frac{1}{2} E[Z | x_1 = \text{true}] + \frac{1}{2} E[Z | x_1 = \text{false}]$$

$\Rightarrow$ either $E[Z | x_1 = \text{true}]$ or $E[Z | x_1 = \text{false}]$ is $\geq E[Z]$!

$E[Z | x_1 = \text{true}]$ or $E[Z | x_1 = \text{false}]$ is $\geq E[Z]$!

how to decide which?

$$E[Z | x_1 = \text{true}] = \sum_{i=1}^m E[Y_i | x_1 = \text{true}]$$

$\uparrow$
easy to calculate

so we can compute $E[Z | x_1 = \text{true}]$, $E[Z | x_2 = \text{false}]$
and decide better

$$Y_i = \begin{cases} 1 & \text{if } i\text{th clause satisfied} \\ 0 & \text{otherwise} \end{cases}$$

$$Z = \sum_{i=1}^m Y_i = \# \text{ clauses satisfied}$$

$$E[Z] = \frac{1}{2} E[Z | x_1 = \text{true}] + E[Z | x_1 = \text{false}]$$

induction step

$a_1, \dots, a_k \in \{\text{true}, \text{false}\}$ s.t.

$$E[Z | x_1 = a_1, \dots, x_k = a_k] \geq E[Z]$$

for x_{k+1} :

$$E[Z | x_1 = a_1, \dots, x_k = a_k] =$$

$$\frac{1}{2} E[Z | x_1 = a_1, \dots, x_k = a_k, x_{k+1} = \text{true}]$$

$$+ \frac{1}{2} E[Z | x_1 = a_1, \dots, x_k = a_k, x_{k+1} = \text{false}]$$

"method of conditional expectations"

$$Y_i = \begin{cases} 1 & \text{if } i\text{th clause satisfied} \\ 0 & \text{otherwise} \end{cases}$$

$$Z = \sum_{i=1}^m Y_i = \# \text{ clauses satisfied}$$

$$E[Z] = \frac{1}{2} E[Z | x_1 = \text{true}] + E[Z | x_1 = \text{false}]$$

$$E[Z | x_1 = \text{true}] = \sum_{i=1}^m E[Y_i | x_1 = \text{true}]$$

$\uparrow$
easy to calculate

one of these
must be good

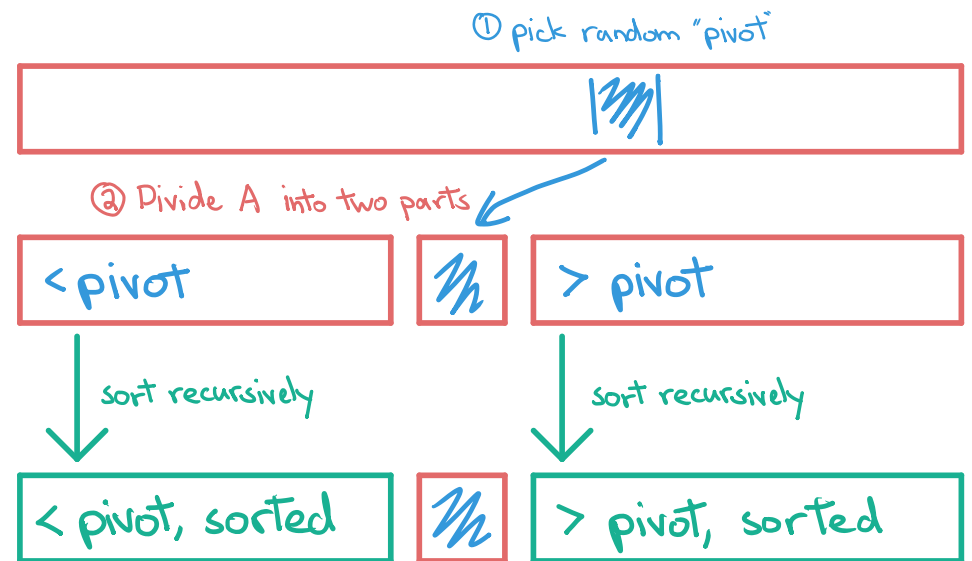
Randomized Sorting

Goal: sort $A[1, n, n]$

Silly algorithm:

1. choose $i \in [n]$ uniformly at random
 2. Recursively sort all #'s $> A[i]$
 3. Recursively sort all #'s $< A[i]$
- ↪ called "Quick-Sort"

Random divide and conquer



we will prove that QuickSort:

① takes $O(n \log n)$ time in expectation

② takes $O(n \log n)$ time w/ high probability

Expected running time of QuickSort

For each $i, j \in [n]$ w/ $i < j$,

$$X_{ij} = \begin{cases} 1 & \text{if } i\text{th smallest element is} \\ & \text{compared to } j\text{th smallest} \\ 0 & \text{otherwise} \end{cases}$$

$\sum_{i < j} X_{ij}$ = total # comparisons made
by the algorithm
 $\approx$ total running time

$E[\sum_{i < j} X_{ij}]$ = average running time

$$E[\sum_{i < j} X_{ij}] = \sum_{i < j} E[X_{ij}]$$

Linearity of Expectation
 $E[X+Y] = E[X] + E[Y]$ (extremely useful)
"average sum = sum of averages"

now analyze $E[X_{ij}]$ in isolation

$$E[X_{ij}] = \Pr[X_{ij} = 1]$$

$$= \Pr \left[\begin{array}{l} i\text{th element is} \\ \text{compared to } j\text{th} \end{array} \right]$$



i th elem is compared w/ j th
 $\Leftrightarrow$

i th or j th elem selected
before any element in between
happens w/ probability $\frac{2}{j-i+1}$

$$E[X_{ij}] = \frac{2}{j-i+1}$$

$$E[\sum_{i < j} X_{ij}] = \sum_{i < j} E[X_{ij}] \quad \text{by linearity of expectation}$$

$$= \sum_{i=1}^n \sum_{j=i+1}^n \frac{2}{j-i+1} \equiv \sum_{j=1}^n \sum_{i=1}^j \frac{2}{j-i+1}$$

$$\leq 2n \left(\frac{1}{1} + \frac{1}{2} + \dots + \frac{1}{n} \right)$$

$$\leq \underline{O(n \log n)} \quad \text{nth harmonic number}$$

Today:

② Randomized algorithms ("average over random bits")

Probability

Hash Functions,
Markov's Inequality

Algorithms

Frequency Estimation,
Heavy Hitters (in streams)

Explore what the world is searching

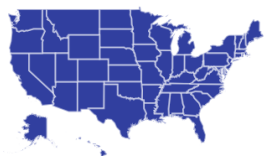
Enter a search term or a topic



Or start with an example

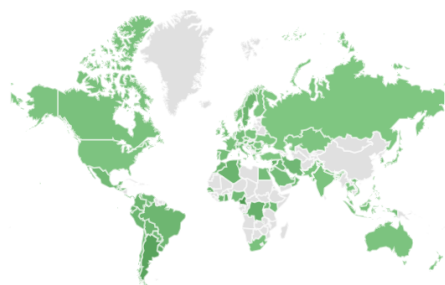
HIDE

● Taylor Swift ● Kim Kardashian



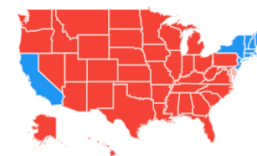
Interest by subregion, Past 7 days, United States

● World Cup



Interest by region, Past 7 days, Worldwide

● Football ● American football



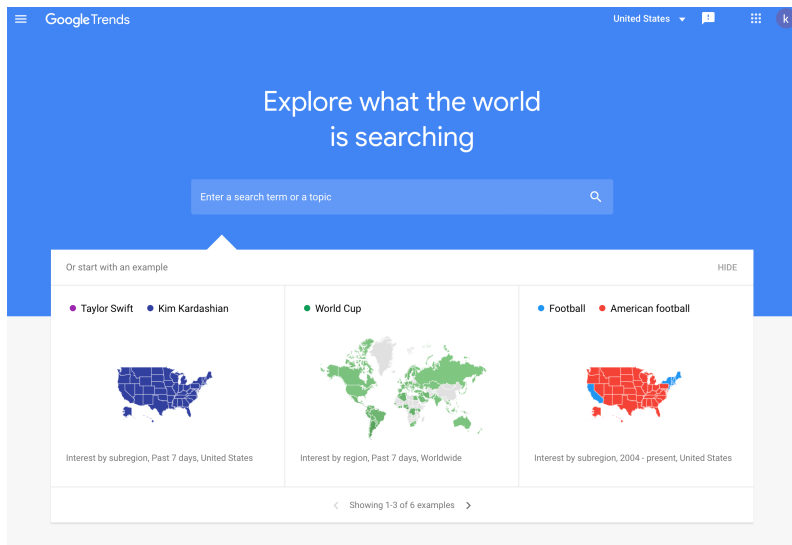
Interest by subregion, 2004 - present, United States

< Showing 1-3 of 6 examples >

Recently trending

Keep up with the latest trending searches

Don't Worry Darling 200K+ searches	Barcelona vs Man City 200K+ searches
Liger movie review 100K+ searches	Chet Holmgren 200K+ searches
Tyron Smith 50K+ searches	Nick Cannon 200K+ searches
Barcelona 200K+ searches	Vanessa Bryant 200K+ searches
Sylvester Stallone 200K+ searches	Patrick Beverley 100K+ searches
MORE TRENDING SEARCHES →	



Recently trending	
Keep up with the latest trending searches	
Don't Worry Darling 200K+ searches	Barcelona vs Man City 200K+ searches
Liger movie review 100K+ searches	Chet Holmgren 200K+ searches
Tyron Smith 50K+ searches	Nick Cannon 200K+ searches
Barcelona 200K+ searches	Vanessa Bryant 200K+ searches
Sylvester Stallone 200K+ searches	Patrick Beverley 100K+ searches
MORE TRENDING SEARCHES →	

tracking most popular queries is important for
caching

But how?

Way too many different queries to
maintain counters for each

Heavy Hitters In Streams



- Elements one by one online
(can repeat)

a, b, c, b, a, d, a, e, a, b, a, $\$$

- 5% Heavy Hitters: elements that appear 5% of the time
(more generally 5% $\rightarrow$ parameter $\epsilon > 0$)

Heavy Hitters In Streams

- Elements one by one online
(can repeat)



a, b, c, b, a, d, a, e, a, b, a, f

- 5% Heavy Hitters: elements that appear $\geq 5\%$ of the time

(more generally 5% $\rightarrow$ parameter $\epsilon > 0$)

Goal: identify all heavy hitters
w/ sublinear space

Saving all counts in a dictionary
takes too much space

e.g. $O(\sqrt{n})$, $O(\log n)$ where all elements are
IDs between 1 and n

- 5% Heavy Hitters: elements that appear $\geq 5\%$ of the time

(more generally $5\% \rightarrow$ parameter $\epsilon > 0$)

Goal: identify all heavy hitters
w/ sublinear space

How is this even possible?

(a) Only 20 (5%)-heavy hitters

(b) Randomization: allow small prob. of failure

(c) Approximation: allow some margin of error

Today:

② Randomized algorithms ("average over random bits")

Probability

Hash Functions,
Markov's Inequality

Algorithms

Frequency Estimation,
Heavy Hitters (in streams)

Markov's Inequality (special case)

Let $X \geq 0$ be nonnegative RV

$$\Pr[X \geq 2E[X]] \leq \frac{1}{2}$$

e.g.

$\leq 50\%$ of US popul. earns $2 \times$ average

$\leq 50\%$ of NBA players score
 $2 \times$ league average points

$\leq 50\%$ students get
 $2 \times$ average midterm score

Markov's Inequality

Let $X \geq 0$ be nonnegative RV

$$\Pr[X \geq 2E[X]] \leq \frac{1}{2}$$

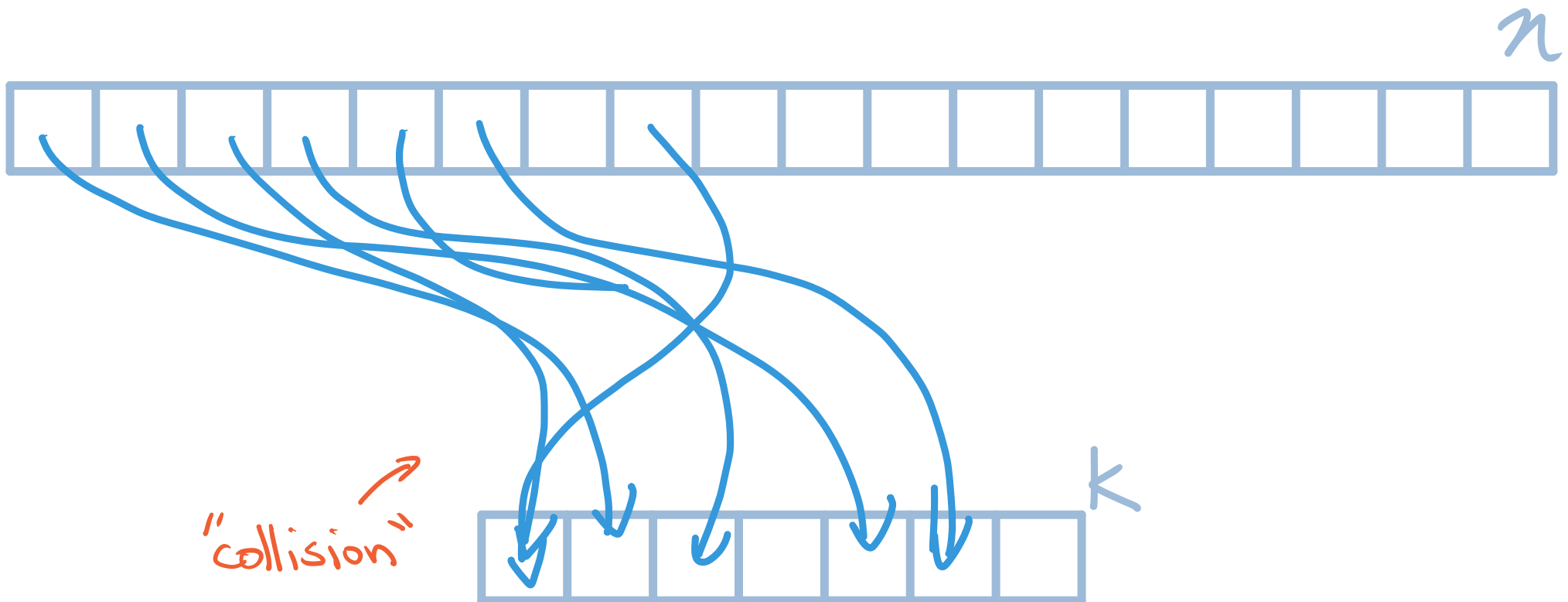
more generally, for $\alpha \geq 1$,

$$\Pr[X \geq \alpha E[X]] \leq \frac{1}{\alpha}$$

② Hash Functions

Randomly constructed function

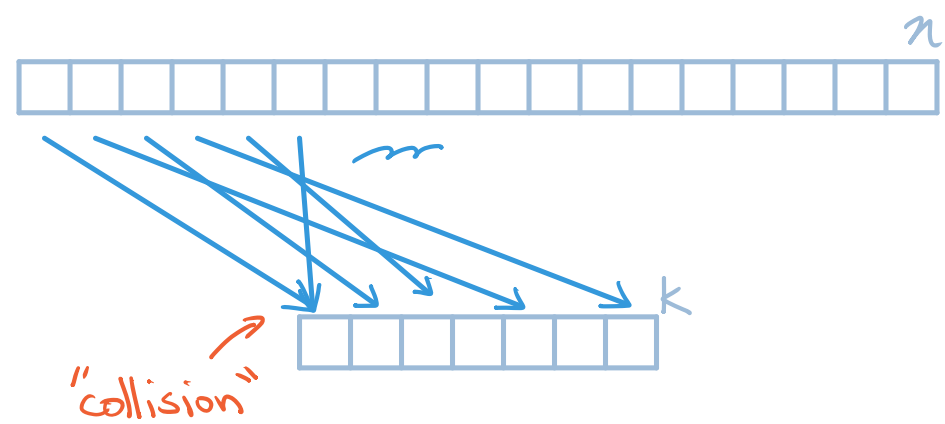
big $h: [n] \rightarrow [k]$ rel. small



② Hash Functions

Randomly constructed function

$h: [n] \rightarrow [k]$
big $\rightarrow$ rel. small



"Ideal Hash Function"

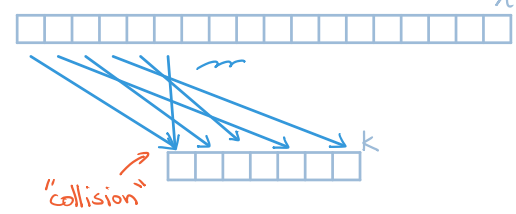
$nk \quad k^n$

$h: [n] \rightarrow [k]$

w/ each $h(i)$ indep. + uniformly at random
in $[k]$

requires $\log_2 k^n = n \log k$ space

Universal Hash Functions



$h: [n] \rightarrow [k]$ s.t. for all distinct $i, j \in [n]$

$$\Pr[h(i) = h(j)] \leq \frac{1}{k}$$

(much weaker than ideal hash)

e.g. $h(x) = ((ax + b) \bmod p) \bmod k$

where

- p prime, $> k$
- a in $\{1, \dots, p\}$ unif. random
- b in $\{0, \dots, p-1\}$ unif. random

(proofs left as exercise)

2 new probabilistic tools

① "Markov's Inequality"

$$\Pr[X \geq 2E[X]] \leq \frac{1}{2}$$

② universal hash $h: [n] \rightarrow [k]$

$$\Pr[h(i) = h(j)] \leq \frac{1}{k}$$

Today:

② Randomized algorithms ("average" over random bits)

Probability

Hash Functions,

Universal Hash Functions

$h: [n] \rightarrow k$ s.t. for all pairs $i, j \in [n]$,
 $\Pr[h(i) = h(j)] \leq \frac{1}{k}$ (much weaker than ideal hash)

e.g. $h(x) = (ax + b) \bmod p \bmod k$

where • p prime, $> k$

• a in $\{1, \dots, p\}$ unif. random

• b in $\{0, \dots, p-1\}$ unif. random



Algorithms

Frequency Estimation, Heavy Hitters (in streams)

Markov's Inequality

Markov's Inequality (special case)

Let $X \geq 0$ be nonnegative RV

$$\Pr[X \geq 2E[X]] \leq \frac{1}{2}$$

Heavy Hitters In Streams



- Elements one by one online
(can repeat)

a, b, c, b, a, d, a, e, a, b, a, §

- 5% Heavy Hitters: elements that appear $\geq 5\%$ of the time
(more generally 5% $\rightarrow$ parameter $\epsilon > 0$)

Heavy Hitters In Streams

- Elements one by one online
(can repeat)



a, b, c, b, a, d, a, e, a, b, a, f

- 5% Heavy Hitters: elements that appear $\geq 5\%$ of the time

(more generally 5% $\rightarrow$ parameter $\epsilon > 0$)

Goal: identify all heavy hitters
w/ sublinear space

Saving all counts in a dictionary
takes too much space

e.g. $O(\sqrt{n})$, $O(\log n)$ where all elements are
IDs between 1 and n

From now on:

ϵ -heavy hitters for fixed $\epsilon > 0$

Related Problem

Heavy Hitters In Streams

- Elements one by one online
(can repeat)

a, b, c, b, a, d, a, e, a, b, a, f

- 5% Heavy Hitters: elements that appear $\geq 5\%$ of the time
(more generally 5% $\rightarrow$ parameter $\epsilon > 0$)



estimate relative frequency of each element

(up to ϵ additive error)

(absolute freq x) = # times x appears in stream

$$\left(\begin{array}{c} \text{relative} \\ \text{freq. of } x \end{array} \right) = \frac{(\text{abs. freq. } x)}{\text{total length of stream}}$$

ϵ -heavy hitter $\Leftrightarrow$ relative freq $\geq \epsilon$

So to recap:

want to estimate all relative frequencies
up to ϵ -additive error

not enough space for all counters

O is a good estimate for all but $\frac{1}{\epsilon}$ elements

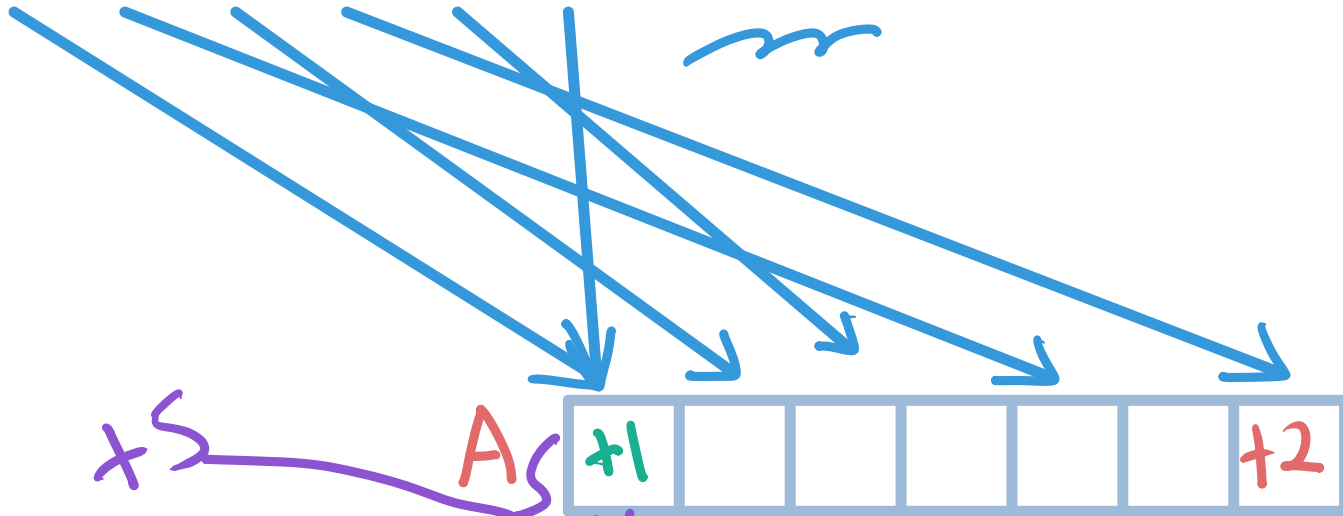
still: don't know which $\frac{1}{\epsilon}$ elements to count

Nashing has entered the chat

Idea: allocate $w = 10/\epsilon$ counters

hash element i to counter $h(i)$

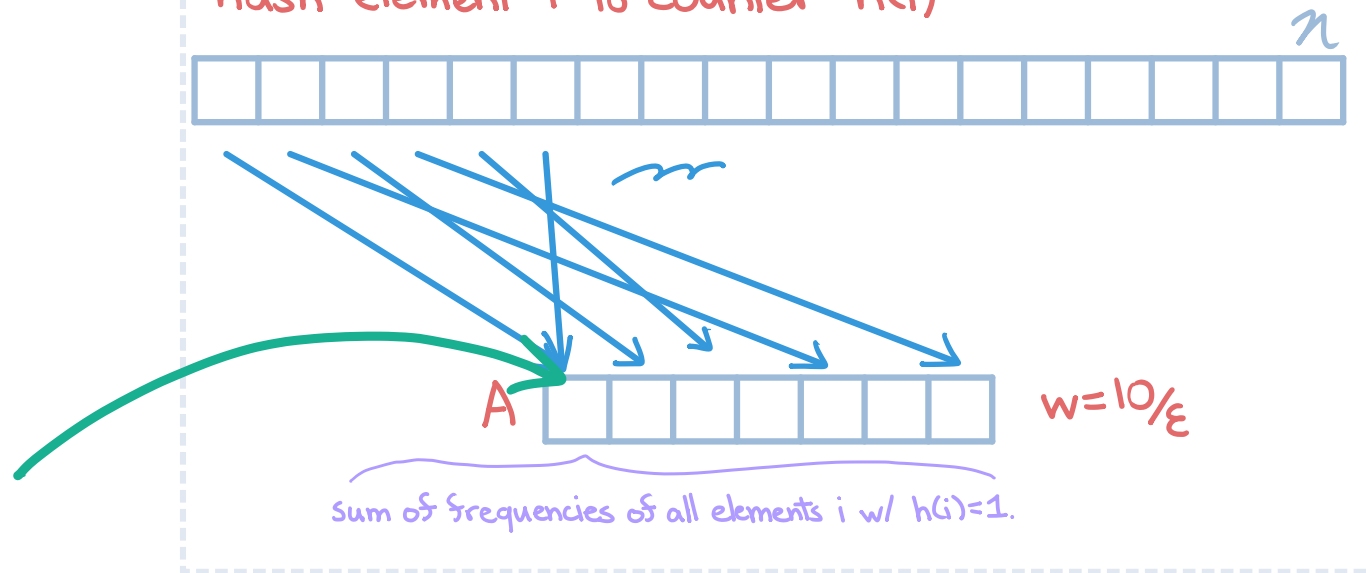
n



$w = 10/\epsilon$

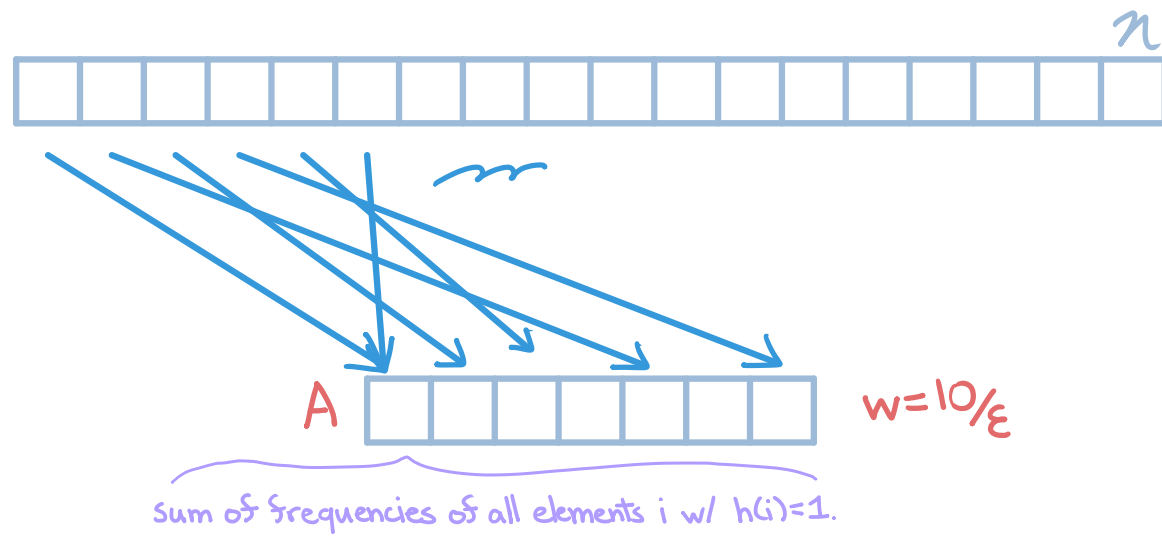
sum of frequencies of all elements i w/ $h(i)=1$.

Idea: allocate $w = 10/\epsilon$ counters
hash element i to counter $h(i)$



To estimate frequency f_i of i :

return $A[h(i)]$



Notation

$A[1, \dots, w]$ = array of counters

f_i = (true) frequency of i

$h: [n] \rightarrow [w]$: universal hash function

m = length of stream

e.g. $A[j] = \sum_{i: h(i)=j} f_i$

Universal Hash Functions
 $h: [n] \rightarrow [k]$ s.t. for distinct $i, j \in [n]$
 $\Pr[h(i)=h(j)] \leq \frac{1}{k}$

Lemma For all i ,
w/ probability $\geq \frac{1}{2}$,

$$f_i \leq \underbrace{A[h(i)]}_{\leq f_i + \frac{\epsilon}{5}m}$$

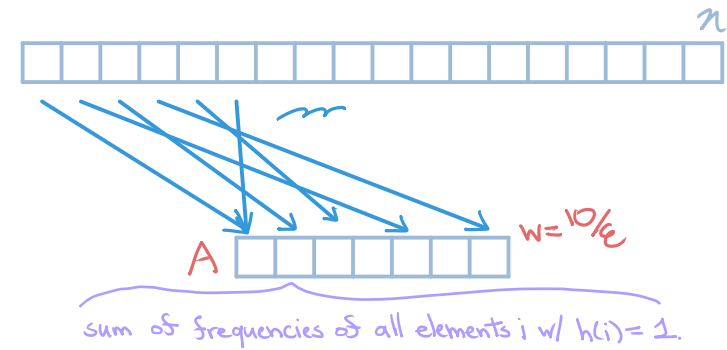
$$A[h(i)] - f_i \leq \frac{\epsilon}{5}m$$

$$\underline{E[A[h(i)] - f_i]}$$

$$= E\left[\sum_{j: h(j)=h(i)} f_j\right] = \sum_j E[f_j \mathbb{1}_{h(j)=h(i)}]$$

$$= \sum_j f_j \Pr[h(j)=h(i)]$$

Universal Hash Functions
 $h: [n] \rightarrow [k]$ s.t. for distinct $i, j \in [n]$
 $\Pr[h(i)=h(j)] \leq \frac{1}{k}$



$A[1..w]$ = array of counters

f_i = (true) frequency of i

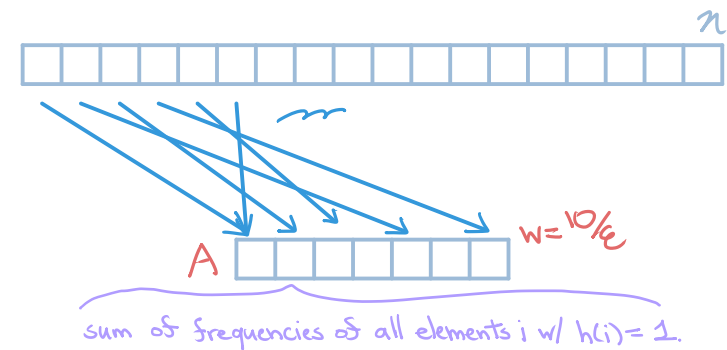
$h: [n] \rightarrow [w]$: universal hash fn.

m = length of stream

$$= \frac{\epsilon}{10} \sum_j f_j \leq \frac{\epsilon m}{10}$$

$$\Pr[A[h(i)] f_i \geq \frac{\epsilon m}{5}] \leq \frac{1}{2}$$

Universal Hash Functions
 $h: [n] \rightarrow [k]$ s.t. for distinct $i, j \in [n]$
 $\Pr[h(i) = h(j)] \leq \frac{1}{k}$



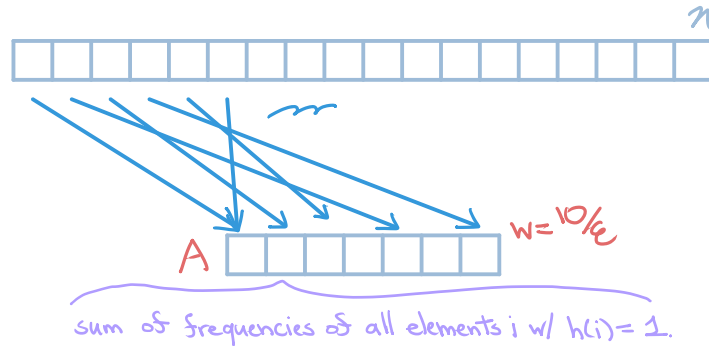
$A[1..w]$ = array of counters

f_i = (true) frequency of i

$h: [n] \rightarrow [w]$: universal hash fn.

m = length of stream

Recap:



$A[1..w]$ = array of counters

f_i = (true) frequency of i

$h: [n] \rightarrow [w]$: universal hash fn.

m = length of stream

Lemma For all i , w/ probability $\geq \frac{1}{2}$,
 $f_i \leq A[h(i)] \leq f_i + \frac{\epsilon}{5} m$

we have: for each element, good estimate
of frequency w/ constant probability

we want: estimates of all frequencies w/
high probability

we want to "amplify" our bounds



Repeatedly flip coins:

$$P[\text{first toss is tails}] = \frac{1}{2}$$

$$P[\text{first 2 tosses are tails}] = \frac{1}{4}$$

6

$$P[\text{first 100 tosses are tails}] = \frac{1}{2^{100}}$$

make d copies of hashed counters

A_1, h_1

--	--	--	--	--	--	--	--

A_2, h_2

--	--	--	--	--	--	--	--

A_3, h_3

--	--	--	--	--	--	--	--

A_4, h_4

--	--	--	--	--	--	--	--

$\vdots$

A_d, h_d

--	--	--	--	--	--	--	--

total space
 $O(wd)$

estimate S_i w/ $\min_{j=1, \dots, d} A_j[h_j(i)]$

bad estimate iff ALL hashed counters are bad

set $d = 10 \log(n)$

each element is "bad" w/ prob.

$$\leq \frac{1}{2^d} = \frac{1}{n^{10}}$$

goal:

ensure all elements are correct

Union bound

$P[\text{some } e \text{ bad}]$

$$\leq \sum_e \Pr[e \text{ bad}] = n \cdot \frac{1}{n^{10}} = \frac{1}{n^9}$$

Heavy Hitters In Streams



- Elements one by one online
(can repeat)

$a, b, c, b, a, d, a, e, a, b, a, \dots$

- 5% Heavy Hitters: elements that appear $\geq 5\%$ of the time
(more generally $5\% \rightarrow$ parameter $\epsilon > 0$)

A_1, h_1

--	--	--	--	--	--	--	--

A_2, h_2

--	--	--	--	--	--	--	--

A_3, h_3

--	--	--	--	--	--	--	--

A_4, h_4

--	--	--	--	--	--	--	--

$\vdots$

A_d, h_d

--	--	--	--	--	--	--	--

estimate f_i

w/ $\min_{j=1, \dots, d} A_j[h_j(i)]$

bad
estimate

iff

ALL
hashed counters
are bad
 $\nearrow$ prob $\frac{1}{2^d}$

$\Rightarrow$ all elements estimated accurately w/ prob. $\geq 1 - \frac{1}{n^9}$

Space:

$$O(nd) = O\left(\frac{\log(n)}{\epsilon}\right)$$

Heavy Hitters In Streams



- Elements one by one online (can repeat)

a, b, c, b, a, d, a, e, a, b, a, f

- 5% Heavy Hitters: elements that appear $\geq 5\%$ of the time

(more generally $5\% \rightarrow$ parameter $\epsilon > 0$)

A_1, h_1

--	--	--	--	--	--	--	--

A_2, h_2

--	--	--	--	--	--	--	--

A_3, h_3

--	--	--	--	--	--	--	--

A_4, h_4

--	--	--	--	--	--	--	--

$\vdots$

A_d, h_d

--	--	--	--	--	--	--	--

estimate S_i

w/ $\min_{j=1, \dots, d} A_j[h_j(i)]$

bad estimate iff ALL hashed counters are bad
 $\uparrow$
 prob $\frac{1}{2^d}$

set $d = 10 \log(n)$

each element is "bad" w/ prob.

$$\leq 2^{-d} = \frac{1}{n^{10}}$$

$$\text{Prob[any element bad]} \leq \sum_{i=1}^n \text{Pr[element } i \text{ bad]} \\ \leq n \cdot \frac{1}{n^{10}} = \frac{1}{n^9}$$

$\Rightarrow$ all elements estimated accurately w/ prob. $\geq 1 - \frac{1}{n^9}$

$\Rightarrow$ all elements estimated accurately w/ prob. $\geq 1 - \frac{1}{n^9}$

Space: $O(wd) = O(\log(n)/\epsilon)$

All put together

Estimate all rel. frequencies up to $\pm \epsilon$ error w/

- $O(\log n / \epsilon)$ space

- $\geq 1 - \frac{1}{n^{10}}$ prob. of success

Heavy Hitters In Streams



- Elements one by one online (can repeat)

a, b, c, b, a, d, a, e, a, b, a, f

- 5% Heavy Hitters: elements that appear $\geq 5\%$ of the time

(more generally 5% $\rightarrow$ parameter $\epsilon > 0$)

A_1, h_1

--	--	--	--	--	--	--	--

A_2, h_2

--	--	--	--	--	--	--	--

A_3, h_3

--	--	--	--	--	--	--	--

A_4, h_4

--	--	--	--	--	--	--	--

$\vdots$

A_d, h_d

--	--	--	--	--	--	--	--

estimate f_i

w/ $\min_{j=1, \dots, d} A_j[h_j(i)]$

$\nearrow$

bad estimate

iff ALL hashed counters are bad
 $\nearrow$ prob $\frac{1}{2^d}$

set $d = 10 \log(n)$

each element is "bad" w/ prob.

$$\leq 2^{-d} = \frac{1}{n^{10}}$$

$$\text{Prob[any element bad]} \leq \sum_{i=1}^n \text{Pr[element } i \text{ bad]} \leq n \cdot \frac{1}{n^{10}} = \frac{1}{n^9}$$

$\Rightarrow$ all elements estimated accurately w/ prob. $\geq 1 - \frac{1}{n^9}$

Today:

② Randomized algorithms ("average over random bits")

Probability

Hash Functions,
Markov's Inequality

Algorithms

Frequency Estimation,
Heavy Hitters (in streams)

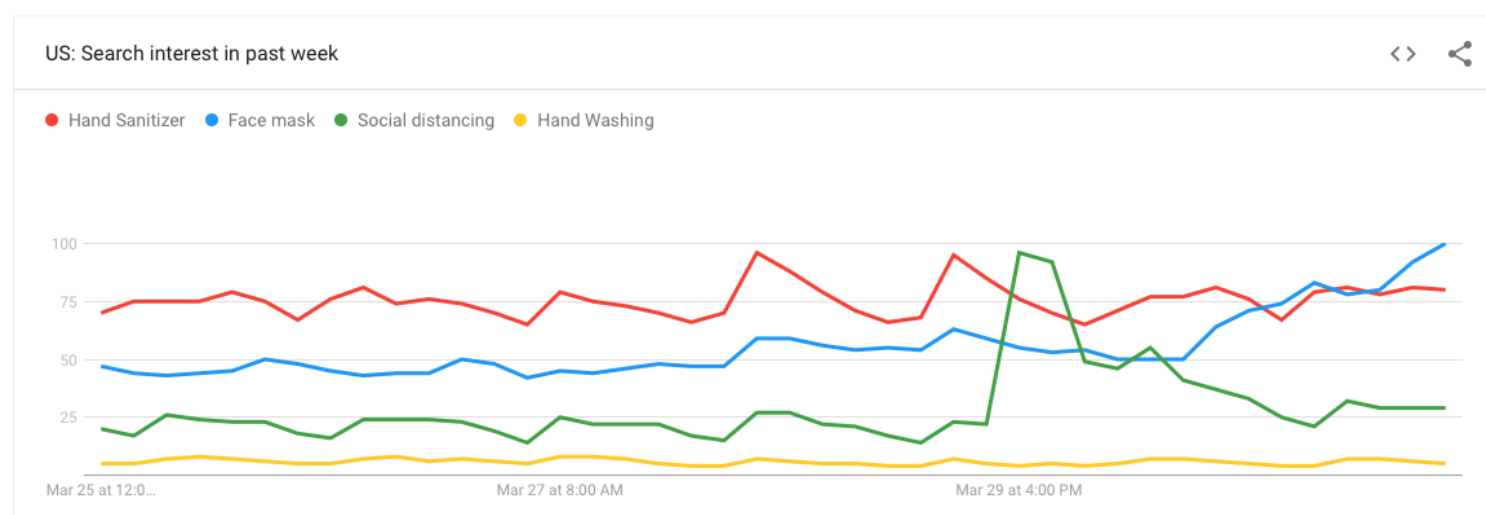
next time: hash tables

Chapter 2

Hashing and heavy hitters

2.1 Introduction

Google has an interesting web page called “[Google trends](#)”, which tracks surging search queries around the world in real time. In Spring 2021, there was even a subpage of search trends specifically related to the Covid-19.



Google tracks the trending search queries not just for the sake of curiosity. Its goal is not only to serve queries, but to serve queries *fast*. The best way to serve something quickly is to have it ready before it is even asked for. By keeping track of the ‘heavy hitter’ search terms - a few search terms that make up a disproportionate amount of the search traffic - Google can cache the answers to most search requests before they are even made.

Google currently serves billions¹ of queries a day. Given the sheer magnitude of Google’s search traffic, and the diversity of search queries, it is not obvious how to

¹Maybe 7 billion? See <https://www.internetlivestats.com/google-search-statistics/>

identify the most popular search queries. Certainly one cannot simply have a tally for each search term, since there are too many search terms out there to be stored. More generally, it is prohibitively expensive to maintain any data structure proportional to the input size. Somehow we need an approach that takes *sublinear* space.

Streaming models. We study the heavy hitter problem in the *streaming model* of computation. In the streaming model, the input is a sequence of items presented to the algorithm *one at a time*. We assume the algorithm has memory much less than the size of the input. In particular, it cannot simply write down everything and solve the problem offline. For example, suppose the stream had m items in the stream, and the algorithm was allowed only $O(\sqrt{m})$ space. When the space is so much smaller than the input size, each time an item from the stream is given to the algorithm, the algorithm needs to be fairly selective about what parts of the item (if any) it wants to remember.



The heavy hitters problem. We formalize the heavy hitters problem as follows. We have m elements in a stream $e_1, \dots, e_m$, where each element is from some large universe $[n] = \{1, \dots, n\}$. Elements can repeat. The *absolute frequency* of an element e , denoted f_e , is the number of times the element appears in the stream. The *relative frequency*, denoted p_e , is the fraction of the stream that the element appears in. In a stream of m elements, the relative frequency of an element is the total frequency divided by m .

Ideally we could keep track of the exact frequencies of all the elements. This is impossible with less than $\min\{n, m\}$ bits². Here, and unlike standard algorithmic settings, $O(m)$ or $O(n)$ bits is not good enough. *We have entered an algorithmic regime where simple and exact computations cannot be taken for granted.* Our first step, then, is to identify some new problems that are both *tractable* and *useful*. We may have to relinquish exactness and consider *approximations*, where we allow for some error that we can analyze and control. And, last but not least, we will give up on deterministic computation, and design *randomized* algorithms that have some small – analyzed and controlled – *probability of failure*.

²One can formalize this impossibility as follows. There are $\binom{m+n-1}{n-1}$ ways to make a frequency

ϵ -heavy hitters. Given a fixed parameter $\epsilon \in (0, 1)$, an ϵ -heavy hitter is an element with relative frequency $\geq \epsilon$. The heavy hitters problem is to identify all of the ϵ -heavy hitters for an input parameter $\epsilon > 0$. Note that there can only be $(1/\epsilon)$ -many ϵ -heavy hitters, which preserves some hope that we can identify all of them with space proportional to $1/\epsilon$, rather than m .

We also consider a closely related problem of *(approximate) frequency estimation*. Given a fixed error parameter $\epsilon > 0$, the goal is to estimate every element's relative frequency up to an additive error of ϵ . Equivalently, we want to estimate the absolute frequency of each element up to an ϵm -additive factor. At first it might seem impossible to estimate n -many counts with $o(n)$ space. We have already argued that m exact counters is impossible. Allowing for ϵ -relative error, however, means that 0 is a satisfactory estimate for all but at most $\lfloor 1/\epsilon \rfloor$ elements.

If we can estimate the absolute frequency of each element up to additive error ϵ , then one can find all (3ϵ) -heavy hitters by considering all of the elements with estimated value at least 2ϵ . Such a list might also include $1/\epsilon$ extra elements who have relative frequency $< 3\epsilon$, but still have frequency $> 1/\epsilon$. Frequency estimation gives more information than just who are the heavy hitters. By knowing their frequencies up to some small error, one can also rank them (approximately) from most to least frequent, such as in Google trends.

Conversely, if we knew *a priori* which of the elements are the ϵ -heavy hitters, then ϵ -frequency estimation is trivial. Namely, we would maintain a counter for each of the $\lfloor 1/\epsilon \rfloor$ -many ϵ -heavy hitters. All other elements are ignored and assigned frequency 0. Of course this approach is not possible since we do not know the heavy hitters. Surprisingly we will pursue a strategy that is actually quite similar. We will allocate $w = O(1/\epsilon)$ counters, hoping to use one counter for each heavy hitter (plus a few

vector of n items where the total sum is m . (*Why?*) Suppose we claim that we can describe any combination of counts with k bits. Each k -bit string can describe by only one of these $\binom{m+n-1}{n-1}$ outcomes, so we must have $2^k \geq \binom{m+n-1}{n-1}$, hence $k \geq \log \left(\binom{m+n-1}{n-1} \right)$. Here $\log$ denotes $\log_2$. We also have

$$\binom{n+m-1}{n-1} = \left(\frac{n+m-1}{n-1} \right)^{n-1} \text{ and also } \binom{n+m-1}{n-1} = \binom{n+m-1}{m} \geq \left(\frac{n+m-1}{m} \right)^m$$

Thus

$$k \geq (n-1) \log \left(1 + \frac{m}{n-1} \right) \text{ and } k \geq m \log \left(1 + \frac{n-1}{m} \right).$$

Slightly better low bounds can be obtained via *Stirling's approximation*, which is also related to entropy.

extra for safe measure). Although we do not know the heavy hitters, we will use *randomized hash functions* to obtain a similar effect.

2.2 Hashing

Likely the reader has used hash tables before, and may be aware that they use *hash functions* to randomly map keys to slots in an array. We will discuss hash tables in detail in the following chapter. Let us now define hash functions mathematically.

Loosely speaking, a hash function is a randomly constructed function $h : [n] \rightarrow [k]$ where the values $h(i)$ are (in a qualified sense) randomly distributed through $[k]$. A *collision* is a pair of distinct indices $i_1 \neq i_2 \in [n]$ such that $h(i_1) = h(i_2)$. In most applications, n is much (much, much) larger than k . In this case, there are necessarily many “collisions” (see exercise 3.1). A goal of hash functions is to distribute these collisions “fairly”.

Ideal hash functions. One way to construct a hash function, for example, is to sample, for each $i \in [n]$, a value $h(i) \in [k]$ independently and uniformly at random. This produces an “ideal hash function”, defined as follows.

Definition 2.1. An ideal hash function $h : [n] \rightarrow [k]$ is a uniformly random function $h : [n] \rightarrow [k]$. That is, each $h(i)$ is drawn from $[k]$ independently and uniformly at random.

An ideal hash function h is particularly easy to reason about. For example, for every input i and possible output $j \in [k]$, we have

$$\mathbf{P}[h(i) = j] = \frac{1}{k}.$$

More generally, for any ℓ distinct inputs $i_1, \dots, i_\ell \in [n]$ and ℓ possible outputs $j_1, \dots, j_\ell \in [k]$, we have

$$\begin{aligned} & \mathbf{P}[h(i_1) = j_1, h(i_2) = j_2, \dots, h(i_\ell) = j_\ell] \\ &= \mathbf{P}[h(i_1) = j_1] \mathbf{P}[h(i_2) = j_2] \cdots \mathbf{P}[h(i_\ell) = j_\ell] \\ &= \frac{1}{k^\ell}. \end{aligned}$$

Ideal hash functions are a good model to keep in mind when designing randomized algorithms. Assuming the hash values are completely independent simplifies calculations. In reality, however, ideal hash functions are very expensive to make and store. Indeed, one has to have $n \log k$ bits to be able to describe all of the possible functions

from $[n]$ to $[k]$ (as there are k^n such functions, and we must pay the logarithm of this quantity). This is particularly ill-suited to our streaming setting where n is astronomical and the goal is to use space sublinear to the input size.

Universal hash functions. Fortunately, in most applications, only a *limited amount of randomization* is actually required. For the current discussion, we only require “universal” hash functions that have “ideal pairwise collision probabilities”, in the following sense.

Definition 2.2. A random hash function $h : [n] \rightarrow [k]$ is universal if for any distinct indices $i_1 \neq i_2 \in [n]$, we have

$$\mathbf{P}[h(i_1) = h(i_2)] = \frac{1}{k}.$$

In contrast to ideal hash functions, universal hash functions can be constructed compactly by the following simple algorithm.

new-universal-hash-function(n, k)

1. Pick a prime number p greater than n .
2. Draw an integer $a \in \{1, \dots, p-1\}$ uniformly at random.
3. Draw an integer $b \in \{0, \dots, p-1\}$ uniformly at random.
4. Return the function $h(x) = ((ax + b) \bmod p) \bmod k$.

Theorem 2.3. Consider the randomly constructed function $h : [n] \rightarrow [k]$

$$h(x) = (ax + b \bmod p) \bmod k,$$

where p is a prime number larger than n , $a \in \{1, \dots, p-1\}$ is drawn uniformly at random, and $b \in \{0, \dots, p\}$ is drawn uniformly at random. Then h is a 2-universal hash function.

The proof of theorem 2.3 is given as exercise 2.2. Here we will assume theorem 2.3 and focus on its application to the heavy hitters problem.

2.3 Using hashing to approximate frequencies

Let us now return to the frequency estimation problem. We have elements from the set $[n]$ coming in a stream of elements. We assume we know n *a priori* but not the

length of the stream, which one can think of as being of infinite length. In the analysis, we imagine pausing the stream at a fixed point in time after m elements have arrived, and analyze the algorithm at that point in time.

The goal is to estimate the absolute frequency of each element up to an (ϵm) -additive factor. The crux of the problem is that total space usage should be (more or less) independent of the length of the stream, m , or the number of elements, n . We mentioned briefly above that if we knew the heavy hitters, then we could just maintain a counter for each one. Since there are at most $1/\epsilon$ heavy hitters, this approach would satisfy our space constraints. Of course we do not know the heavy hitters. In the following, we will use hash functions to, in effect, *guess* the heavy hitters.

We first create an array of counters $A[1..w]$ with $w = \lceil 2/\epsilon \rceil$ entries. Note that $2/\epsilon$ is extremely small compared to the total length of the stream, or the distinct number of keys. We also sample a universal hash function $h : \{1, \dots, n\} \rightarrow \{1, \dots, w\}$. For each element e presented by the stream, we increase $A[h(e)]$ by 1. In turn, for each element e , we treat $A[h(e)]$ as an estimate for f_e .

hashed-counters($\epsilon > 0$)

1. allocate an array of size $A[1..w]$ for $w = \lceil 2/\epsilon \rceil$
2. sample a universal hash function $h : \{1, \dots, n\} \rightarrow \{1, \dots, w\}$
3. for each item e in the stream
 - A. $A[h(e)] \leftarrow A[h(e)] + 1$

$A[h(e)]$ never underestimates f_e , and the hope is that it does not overestimate f_e by too much. The risk of error comes from other elements' frequencies possibly adding more than ϵm to $A[h(e)]$. Here the intuition is that the “noise” coming from other frequencies is spread out by the hash function over $\epsilon/2$ entries, so we would only expect $\epsilon m/2$ error for each element e . To translate “expected error” to “probability of error”, we use Markov's inequality, as follows.

Lemma 2.4. *For each element e , with probability $\geq 1/2$, we have*

$$f_e \leq A[h(e)] \leq f_e + \epsilon m.$$

Proof. We have $A[h(e)] \geq f_e$ always because $A[h(e)]$ is a sum of frequencies of elements with hash code $h(e)$, which of course includes e . The expected additive error is bounded above by

$$\mathbf{E}[A[h(e)]] - f_e \stackrel{(a)}{=} \sum_{d \neq e} f_d \mathbf{P}[h(d) = h(e)] \stackrel{(b)}{\leq} m/w \leq \frac{\epsilon}{2} m. \quad (2.1)$$

Here (a) is by linearity of expectation. (b) is because h is universal. Now we have

$$\mathbf{P}[A[h(e)] \geq f_e + \epsilon m] \stackrel{(c)}{\leq} \mathbf{P}[A[h(e)] - f_e \geq 2 \mathbf{E}[A[h(e)] - f_e] \stackrel{(d)}{\leq} \frac{1}{2}$$

Here (c) plugs in the inequality obtained in (2.1). (d) applies Markov's inequality, where we note that $A[h(e)] - f_e \geq 0$. $\square$

Given that it is impossible to track frequencies exactly in sublinear space, it is suprising that we can now count every element's frequency with extremely small space, *sometimes and with small additive error*. The algorithm, including the construction of the universal hash function, is extremely simple.

2.4 Amplification

Section 2.3 shows how to estimate each element with fairly small error with constant probability of error. Our goal now is to reduce the error probability enough to even take the union bound over all of the elements, and thus estimate all frequencies up to ϵm -additive error.

The idea is to use repetition, and one analogy is coin tossing. The goal is to flip enough coin tosses to get at least one heads with very high probability. With one coin toss, the probability that it is tails is $1/2 = .5$. With two coin tosses, the probability that both come up tails is still $1/4 = .25$. But with 100 coin tosses, the probability that all 100 coin tosses come up tails is $1/2^{100} \approx .00000000000000000000000000007886\dots$

The point is that independent trials magnify the probability of at least one success exponentially. For a specified probability of error $\delta \in (0, 1)$, the algorithm `count-min-sketch` below makes $\lceil \log 1/\delta \rceil$ independent instances of `hashed-counters( $\epsilon$ )`. For each element e , it uses the minimum estimate over all of the instances of `hashed-counters`. The overall data structure fails for an element e only if *every* instance of `hashed-counters` fails, which by the analogy with coins, is exceedingly unlikely.

count-min-sketch($\epsilon > 0, \delta > 0$)

1. build $d = \lceil \log 1/\delta \rceil$ instances $(A_1, h_1), \dots, (A_d, h_d)$ of `hashed-counters`(ϵ) over the stream
2. to query an element e :

A. return $\min_{i=1,\dots,d} A_i[h_i(e_i)]$

We now have the following smaller error probability for each element e .

Lemma 2.5. *For each element e , with probability $\geq 1 - \delta$, we have*

$$\min_{i=1,\dots,d} A_i[h_i(e)] \leq f_e + \epsilon m.$$

Proof. We have

$$\mathbf{P}\left[\min_{i=1,\dots,d} A_i[h_i(e)] > f_e + \epsilon m\right] \stackrel{(a)}{=} \prod_{i=1}^d \mathbf{P}[A_i[h_i(e)] > f_e + \epsilon m] \stackrel{(b)}{\leq} \frac{1}{2^d} \leq \delta.$$

Here (a) is by independence of each $A_i[h_i(e)]$. (b) is by lemma 2.4. $\square$

For δ set to a polynomial of $1/n$, the probability of error becomes low enough to take a union bound over all elements in $[n]$, as follows.

Theorem 2.6. *Given a stream of elements from the range $[n]$, **count-min-sketch** $(\epsilon, 1/n^2)$ has the following guarantee at any fixed point in the stream.*

*Suppose m elements have been presented in the stream. With probability at least $1 - 1/n$, **count-min-sketch** $(\epsilon, 1/n^2)$ overestimates the total frequency of each element with additive error at most ϵm and total space $O(\log(n)/\epsilon)$.*

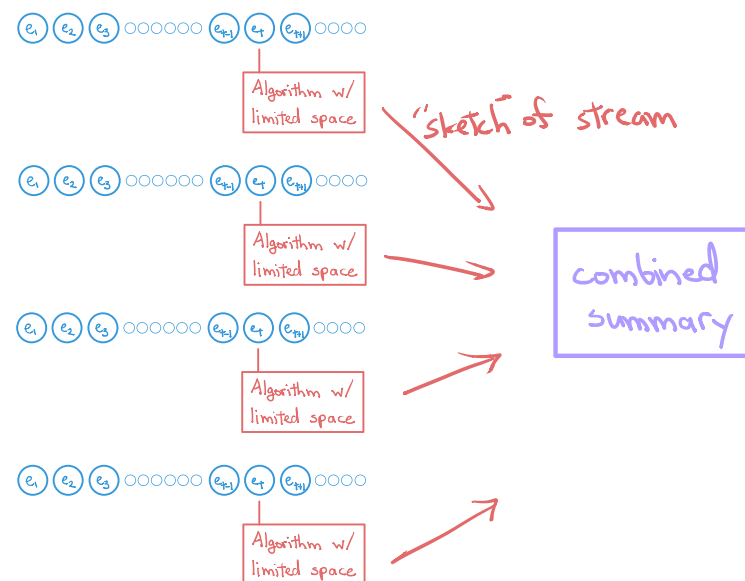
Proof. By lemma 2.5, we have probability of error $\leq 1/n^2$ for each element e . Taking the union bound over all n elements in the stream, we have probability of error $\leq 1/n$. $\square$

Remark 2.7. More precisely, the space usage of **count-min-sketch** $(\epsilon, 1/n^2)$ is that of $O(\log(n)/\epsilon)$ counters. Here we assume each counter takes $O(1)$ space for simplicity.

2.5 Extensions

2.5.1 Crossing streams

One can extend the streaming model to multiple streams in the following *distributed* model of computation. Here we have several streams simultaneously, each served by an algorithm using sublinear space. The goal is to solve the heavy hitters problem over the *combined streams*.



count-min-sketch has the convenient property of being a *sketch*. To handle multiple streams, we have an instance of **count-min-sketch** for each stream arranged so that they are *all using the same hash functions*. To combine their results, we simply sum up the arrays A_i of hashed sums entry-wise. The result is an instance of **count-min-sketch** over the combined streams.

2.5.2 Turnstile streams

Consider the more general model where each item in the stream consists of an element e and a value Δ , signifying that we should increase the frequency count for e , f_e by Δ . Δ is allowed to be negative, with the restriction that the frequency f_e of each element (which is now the sum of Δ 's for that element) remains nonnegative. This model is sometimes called a “turnstile stream”, in the sense that a turnstile counting the number of people in an amusement park is always nonnegative because each decrease corresponds to a person who entered the park earlier.

count-min-sketch adapts immediately to turnstile stream, by simply adding Δ to $A_i[h_i(e)]$ for each instance (A_i, h_i) of **hashed-counters**. The additive error is now ϵ times the sum of all Δ 's in the stream.

2.6 Takeaways

- There are many basic and useful problems – heavy hitters with sublinear space being on of our first examples – that are too difficult or even impossible to compute exactly and deterministically. Instead we consider *randomized approximation algorithms* that are potentially more scalable. This requires

quantitative analysis to address the approximation factor in addition to algorithm design.

- **count-min-sketch** uses *hashing* to try to distribute the heavy hitters across an array. It does not know which are the heavy hitters, but relies on *randomization* to separate the heavy hitters (most of the time) in an *oblivious* fashion.
- *Ideal hash functions*, while easy to reason about, are prohibitively expensive. Luckily, weaker hash functions with *limited randomness* often suffice, and are easily constructed. **count-min-sketch** requires only *universal hash functions*. Universal hash functions can be implemented very easily.
- *Linearity of expectation*, combined with universal hash functions, implies that the noise seen by a particular element is evenly spread out *on average*. *Markov's inequality* allowed us to argue that the noise encountered by an element is close to the average, most of the time.
- **count-min-sketch** *amplifies* the probability of success by taking the minimum over many independent trials. A particular element is miscounted if and only if *all* independently trials miscount the element, which happens with vanishingly small probability.
- The error probability drops so rapidly that we can apply the *union bound* over all of the elements after just $O(\log n)$ trials.
- **count-min-sketch** does not give *unbiased estimates* of the counters. Instead, **count-min-sketch** tries to be *within a prescribed error with high probability*. It is *consistent*. It is more important to be *consistent* than *unbiased*, since we can (psychologically) adjust for the bias. Many real-world apparatus are designed on this principle.

2.7 Additional notes and references

Spring 2022 lecture materials. Click on the links below for the following files:

- [Handwritten notes prepared before the lecture.](#)
- [Handwritten notes annotated during the presentation.](#)
- [Recorded video lecture.](#)

2.8 Exercises

Exercise 2.1. Let $h : [n] \rightarrow [k]$ be *any* fixed function.

1. Prove that the number of collisions is

$$\geq \frac{n(n-k)}{2k}$$

2. Show that the above inequality is tight when k divides n .

Exercise 2.2. Show that the construction given in section 2.2 is indeed a universal hash function, using the steps listed below.

To recall the construction, we randomly construct a function $h : [n] \rightarrow [k]$ as follows. First, let p be any prime number $> n$. Draw $a \in \{1, \dots, p-1\}$ uniformly at random, and draw $b \in \{0, \dots, p-1\}$ uniformly at random. We define a function $h(x)$ by

$$h(x) = ((ax + b) \mod p) \mod k.$$

1. Let $x_1, x_2 \in [n]$ with $x_1 \neq x_2$, and let $c_1, c_2 \in \{0, \dots, p-1\}$ with $c_1 \neq c_2$. Show that the system of equations

$$\begin{aligned} ax_1 + b &= c_1 \mod p \\ ax_2 + b &= c_2 \mod p \end{aligned}$$

uniquely determines a and b .³

- Step 1 implies that the map $(a, b) \mapsto (ax_1 + b \mod p, ax_2 + b \mod p)$ is a bijection.
2. Let $x_1, x_2 \in [n]$ with $x_1 \neq x_2$, and let $c_1, c_2 \in \{0, \dots, p-1\}$ with $c_1 \neq c_2$. Show that

$$\mathbf{P}[ax_1 + b = c_1, ax_2 + b = c_2] = \frac{1}{p(p-1)}.$$

(Here the randomness is over the uniformly random choices of a and b .)

³Here it is helpful to know that division is well-defined in the set of integers $\mod p$ when p is prime. In particular, “ a/b ” is defined as the unique integer c such that $bc = a$.

3. Fix $x_1, x_2 \in [n]$ with $x_1 \neq x_2$, and $c_1 \in \{0, \dots, p-1\}$. Show that

$$\sum_{\substack{c_2 \in \{1, \dots, p\} \\ c_2 \neq c_1 \\ c_1 = c_2 \pmod n}} \mathbf{P}[ax + b = c_1, ax + b = c_2] \leq \frac{1}{n}.$$

Hint: You may want to show that the number of values c_2 such that $c_1 = c_2 \pmod m$ is $\leq \frac{p-1}{n}$.

4. Finally, show that $\mathbf{P}[h(x_1) = h(x_2)] \leq \frac{1}{k}$.

Exercise 2.3. The **count-min-sketch** data structure allows us to estimate the relative frequency of each element up to an ϵ -additive factor with probability of error $\leq 1/\text{poly}(n)$ with $O(\log(n)/\epsilon)$ space.⁴ The original motivation, however, was to also obtain a list of ϵ -heavy hitters. Design and analyze an algorithm that maintains a list of elements, with at any *particular* point in time,⁵ with probability of error $\leq 1/n^2$:

1. Contains all of the ϵ -heavy hitters.
2. Only includes $(\epsilon/2)$ -heavy hitters.

Your space usage should be comparable to the space used by the **count-min-sketch** data structure.⁶

Additional remark. The question asks for *one data structure* that satisfies *both the criteria* simultaneously. That is, you should maintain a list S that (a) contains all ϵ -heavy hitters, *and* (b) only includes $(\epsilon/2)$ -heavy hitters. The tricky part is that **count-min-sketch** only approximates the frequencies. You may want to account for the fact that an instance of **count-min-sketch** (ϵ, δ) may overestimate the relative frequency of an element by as much as ϵ , which can make a very infrequent element look like an ϵ -heavy hitter.

Exercise 2.4. In this exercise, we develop a refined analysis that can reduce the additive error substantially in many real settings.

⁴Here the elements are integers from $[n] = \{1, \dots, n\}$, where n is known, and $\epsilon \in (0, 1)$ is an input parameter.

⁵To clarify, what we mean by “particular point in time” is as follows. You have a data structure that is processing data over time. Suppose we suddenly paused the stream and asked you to report your list of heavy hitters. Your algorithm should succeed then and there with probability of error $\leq 1/n^2$. For this criteria, you do not need to know the length of the stream.

⁶You may want to use the **count-min** (ϵ, δ) data structure as a black box, but you should be clear about your choice of parameters ϵ and δ .

Let S denote the sum of frequency counts of all elements that are *not* ϵ -heavy hitters:

$$S = \sum_{e: p_e < \epsilon} f_e.$$

Note that $S \leq n$, and S might be much less than n when the stream is dominated by heavy hitters.

Show that, by adjusting `count-min-sketch`($\epsilon, 1/n^2$) slightly by increasing w to $\lceil 4/\epsilon \rceil$, the additive error for every element is at most ϵS with probability of error $\leq 1/n$.

Exercise 2.5. Consider the streaming model where we have elements $e_1, e_2, \dots$ presented one at a time by a stream. A natural task is to sample a fixed number of elements uniformly at random from the stream. Usually, sampling (say) 1 item from a set of m elements is easy: randomly generate a number k between 1 and m , and return the k th element from your set. Sampling in streaming is trickier because we cannot hold the entire stream in memory, and don't know the length of the stream.

1. Consider the following randomized streaming algorithm that selects one element s from the stream:

sample-one

/ m counts the number of elements in the stream so far, and s is
the "sample" of 1 element from the stream. */*

1. $m \leftarrow 0, s \leftarrow \text{nil}.$
2. For each element e presented by the stream:
 - A. $m \leftarrow m + 1.$
 - B. With probability $1/m$:
 1. $s \leftarrow e.$

For $i \in \mathbb{N}$, let e_i denote the i th element in the stream. For $m \in \mathbb{N}$ let s_m denote the value of s after the m th iteration. Show that for all i and m ,

$$\mathbf{P}[s_m = e_i] = \begin{cases} 0 & \text{if } m < i \\ 1/m & \text{if } m \geq i. \end{cases}$$

That is, for each m , s_m is a uniformly random element out of $\{e_1, \dots, e_m\}$.⁷

2. Now let $k \in \mathbb{N}$ be a fixed parameter. (e.g., $k = 3$.) Suppose you want to sample a set of k elements from the stream without replacement. Design and analyze an algorithm generalizing **sample-one** that maintains a sample S of k elements drawn uniformly at random from the stream. That is, for $m \geq k$, your algorithm should have a set S of k elements, where any particular set of k elements is equally likely (i.e., with probability $1/\binom{m}{k}$). For $k = 1$, your algorithm should coincide with **sample-one** above.⁸

⁷Fix i . For $m < i$ the probability 0 since e_i hasn't even appeared in the stream. Now, what about $m = i$? What about $m = i + 1$?

⁸One way to frame your analysis is as follows. For $m \geq k$, let S_m denote the (randomized) sample S after m iterations. Prove the following statement by induction on $m - k$:

For all $m \geq k$, and all sets $X \subseteq \{e_1, \dots, e_m\}$ of k elements,

$$\mathbf{P}[S_m = X] = \frac{1}{\binom{m}{k}}.$$

In our argument, you may have two cases depending on whether or not $e_m \in X$.