

Hashing and Heavy Hitters

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"

"conjunctive normal form"
CNF formula

$$\varphi(x_1, \dots, x_n) = (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_4 \vee \dots) \wedge \dots$$

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 $7/8$ -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"

$$\varphi(x_1, \dots, x_n) = (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_4 \vee \dots) \wedge \dots$$

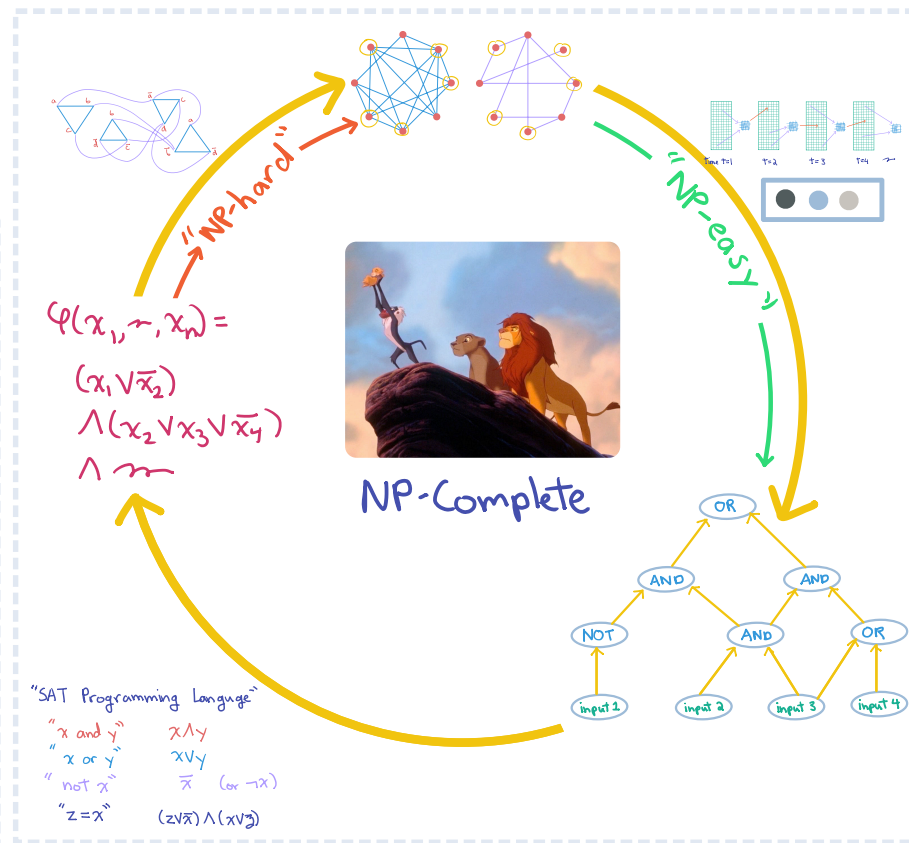
Goal: satisfy Max # clauses

Interesting even if φ is unsatisfiable

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

Algorithm satisfies $\underbrace{d \cdot \text{OPT}}$ 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 = $\sim$ C_2 $\sim$

$$\Pr[E_1, E_2] = \frac{7}{8} \cdot \frac{7}{8}$$

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

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 = $\sim$ C_2 $\sim$

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

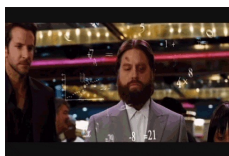
$$\underbrace{\Pr[E_1, E_2 | x_3 = T] \cdot \frac{1}{2}}_1 + \underbrace{\Pr[E_1, E_2 | x_3 = F] \cdot \frac{1}{2}}_{\left(\frac{3}{4}\right)^2}$$

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 = $\sim$ C_2 $\sim$

E_3 = $\sim$ C_3 $\sim$

$\vdots$

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

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

$$E[Z] = \frac{1}{2} E[Z | x_1 = T] + \frac{1}{2} E[Z | x_1 = F]$$

$\Rightarrow$ either $E[Z | x_1 = T]$ or $E[Z | x_1 = F]$ is $\geq E[Z]$!

$E[Z | x_1 = T]$ or $E[Z | x_1 = F]$ is $\geq E[Z]$!

how to decide which?

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

$\uparrow$
easy to calculate

so we can compute $E[Z | x_1 = T]$, $E[Z | x_2 = F]$
and decide better

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

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

$$E[Z] = \frac{1}{2} E[Z | x_1 = T] + E[Z | x_1 = F]$$

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} :

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

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

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

"method of conditional expectations"

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

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

$$E[Z] = \frac{1}{2} E[Z | x_1 = T] + E[Z | x_1 = F]$$

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

$\uparrow$
easy to calculate

one of these
must be good

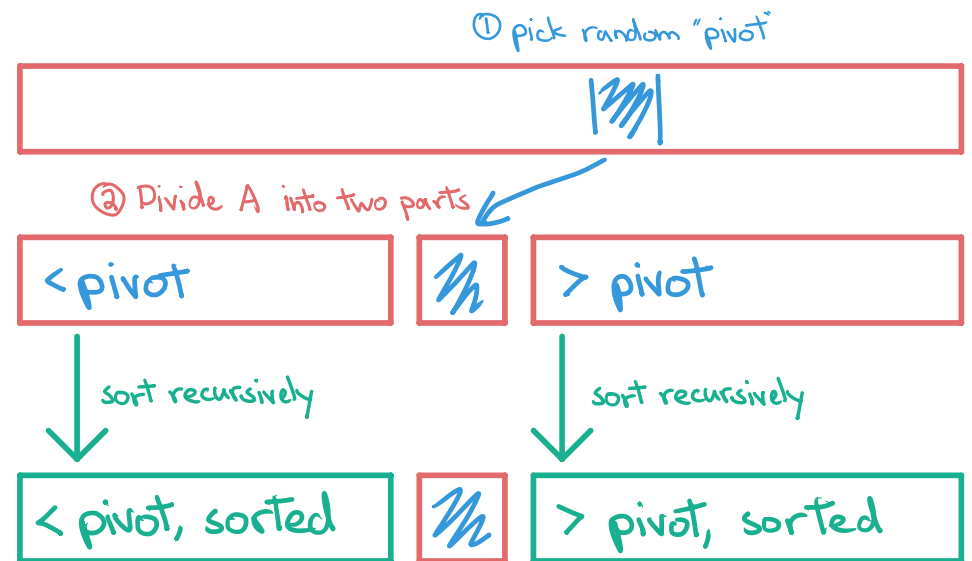
Randomized Sorting

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 $\frac{2}{j-i+1}$

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

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

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

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

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

"Average" analysis of algorithms

↑
(without sacrificing worst-case robustness)

① Amortized analysis ("average" in hindsight)

② Randomized algorithms ("average" over random bits)

Both are very practical, different perspectives

Today:

② Randomized algorithms ("average" over random bits)

Probability

Hash functions,
Markov's Inequality

Algorithms

Hash tables,
Frequency Estimation,
Heavy Hitters (in streams)

5.5. Dictionaries

Another useful data type built into Python is the *dictionary* (see [Mapping Types — dict](#)). Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can’t use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like `append()` and `extend()`.

It is best to think of a dictionary as a set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}`. Placing a comma-separated list of *key:value* pairs within the braces adds initial *key:value* pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a *key:value* pair with `del`. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

Performing `list(d)` on a dictionary returns a list of all the keys used in the dictionary, in insertion order (if you want it sorted, just use `sorted(d)` instead). To check whether a single key is in the dictionary, use the `in` keyword.

Here is a small example using a dictionary:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```


compact1, compact2, compact3
java.util

Interface Map<K,V>

Type Parameters:

K - the type of keys maintained by this map

V - the type of mapped values

All Known Subinterfaces:

Bindings, ConcurrentMap<K,V>, ConcurrentNavigableMap<K,V>, LogicalMessageContext, MessageContext, NavigableMap<K,V>, SOAPMessageContext, SortedMap<K,V>

All Known Implementing Classes:

AbstractMap, Attributes, AuthProvider, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, PrinterStateReasons, Properties, Provider, RenderingHints, SimpleBindings, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap

```
public interface Map<K,V>
```

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

This interface takes the place of the Dictionary class, which was a totally abstract class rather than an interface.

The Map interface provides three *collection views*, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The *order* of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the `TreeMap` class, make specific guarantees as to their order; others, like the `HashMap` class, do not.

Note: great care must be exercised if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects `equals` comparisons while the object is a key in the map. A special case of this prohibition is that it is not permissible for a map to contain itself as a key. While it is permissible for a map to contain itself as a value, extreme caution is advised: the `equals` and `hashCode` methods are no longer well defined on such a map.

All general-purpose map implementation classes should provide two "standard" constructors: a void (no arguments) constructor which creates an empty map, and a constructor with a single argument of type `Map`, which creates a new map with the same key-value mappings as its argument. In effect, the latter constructor allows the user to copy any map, producing an equivalent map of the desired class. There is no way to enforce this recommendation (as interfaces cannot contain constructors) but all of the general-purpose map implementations in the JDK comply.

The "destructive" methods contained in this interface, that is, the methods that modify the map on which they operate, are specified to throw `UnsupportedOperationException` if this map does not support the operation. If this is the case, these methods may, but are not required to, throw an `UnsupportedOperationException` if the invocation would have no effect on the map. For example, invoking the `putAll(Map)` method on an unmodifiable map may, but is not required to, throw the exception if the map whose mappings are to be "superimposed" is empty.

Some map implementations have restrictions on the keys and values they may contain. For example, some implementations prohibit null keys and values, and some have restrictions on the types of their keys. Attempting to insert an ineligible key or value throws an unchecked exception, typically `NullPointerException` or `ClassCastException`. Attempting to query the presence of an ineligible key or value may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible key or value whose completion would not result in the insertion of an ineligible element into the map may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.

Many methods in Collections Framework interfaces are defined in terms of the `equals` method. For example, the specification for the `containsKey(Object key)` method says: "returns true if and only if this map contains a mapping for a key `k` such that (`key==null ? k==null : key.equals(k)`)." This specification should *not* be construed to imply that invoking `Map.containsKey` with a non-null argument `key` will cause `key.equals(k)` to be invoked for any key `k`. Implementations are free to implement optimizations whereby the `equals` invocation is avoided, for example, by first comparing the hash codes of the two keys. (The `Object.hashCode()` specification guarantees that two objects with unequal hash codes cannot be equal.) More generally, implementations of the various Collections Framework interfaces are free to take advantage of the specified behavior of underlying `Object` methods wherever the implementor deems it appropriate.

Some map operations which perform recursive traversal of the map may fail with an exception for self-referential instances where the map directly or indirectly contains itself. This includes the `clone()`, `equals()`, `hashCode()` and `toString()` methods. Implementations may optionally handle the self-referential scenario, however most current implementations do not do so.

This interface is a member of the Java Collections Framework.

Since:

1.2

See Also:

`HashMap`, `TreeMap`, `Hashtable`, `SortedMap`, `Collection`, `Set`

"Dictionaries" / "Maps"

maintain key-value pairs
(k_1, v_1), (k_2, v_2), ...

- insertion
- deletion
- lookup

5.5. Dictionaries

Another useful data type built into Python is the *dictionary* (see Mapping Types — dict). Dictionaries are sometimes found in other languages as "associative memories" or "associative arrays". Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type: strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can't use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like `append()` and `extend()`.

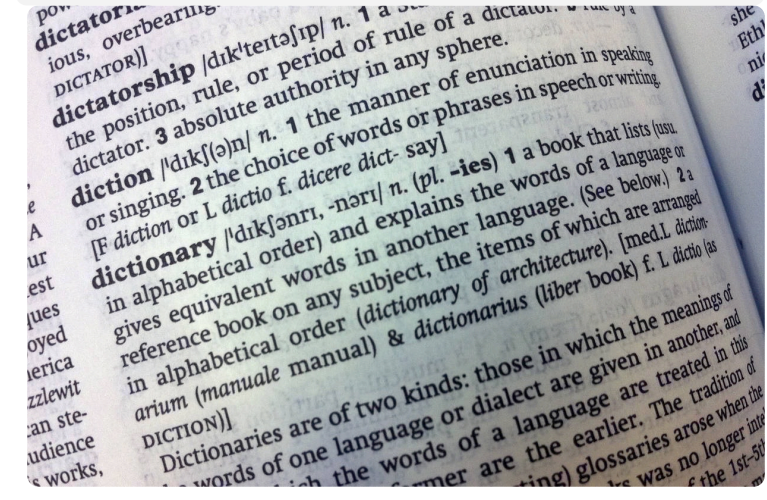
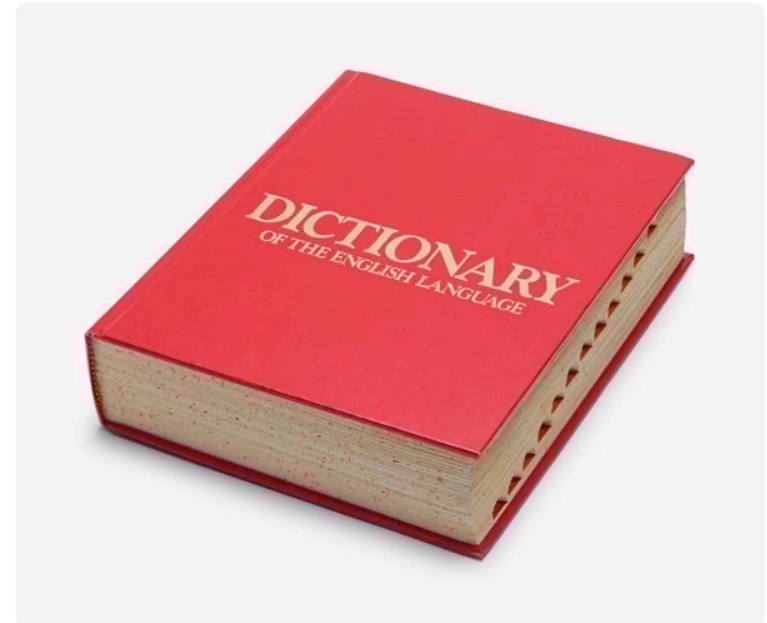
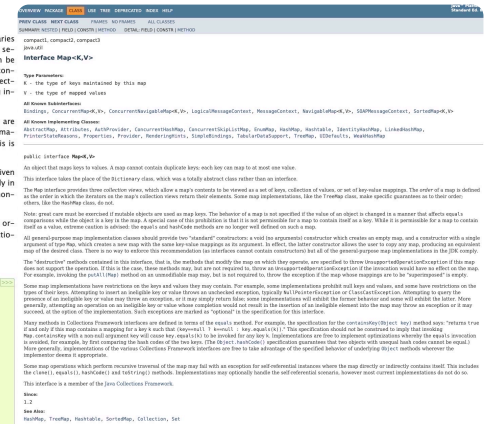
It is best to think of a dictionary as a set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}`. Placing a comma-separated list of *key:value* pairs within the braces adds initial *key:value* pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some *key* and extracting the value given the *key*. It is also possible to delete a *key:value* pair with `del`. If you store using a *key* that is already in use, the old value associated with that *key* is forgotten. It is an error to extract a value using a non-existent *key*.

Performing `list(d)` on a dictionary returns a list of all the keys used in the dictionary, in insertion order if you want it sorted, just use `sorted(d)` instead). To check whether a single *key* is in the dictionary, use the `in` keyword.

Here is a small example using a dictionary:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

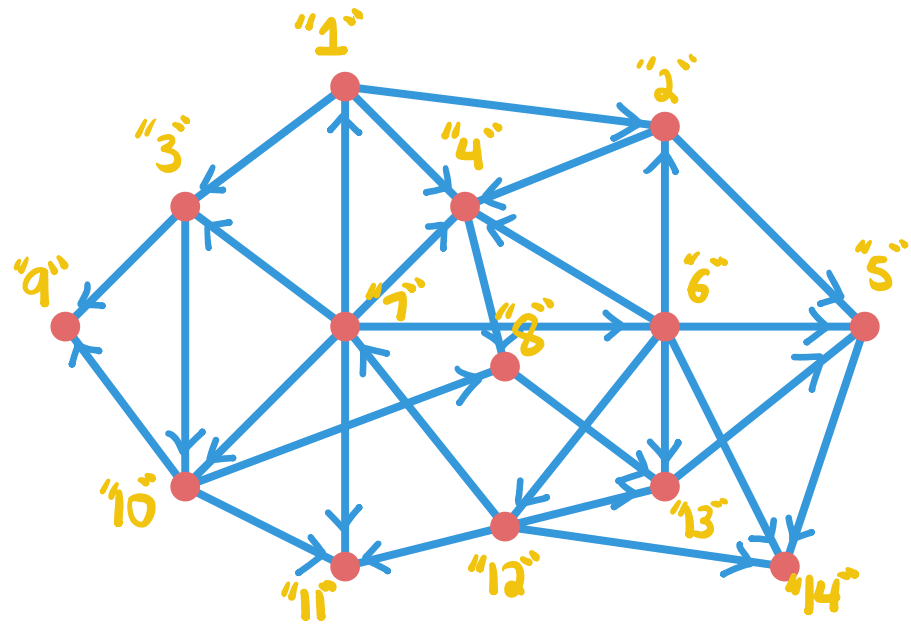


Very special case

keys = contiguous integers $\{1, \dots, n\}$

e.g. index vertices of
a graph $v_1, \dots, v_n$

(or after reindexing anything, really)



Solution: Array $A[1 \dots n]$

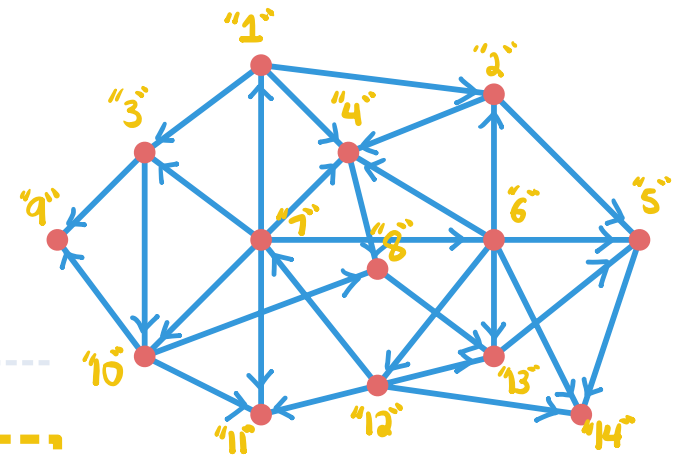


$O(1)$ access, $O(n)$ space

keys = contiguous integers $\{1, \dots, n\}$

e.g. index vertices of a graph $v_1, \dots, v_n$

(or after reindexing anything, really)



Solution: Array $A[1 \dots n]$

k_1	k_2	k_3	k_4	k_5	k_6	k_7	k_8	k_9	k_{10}	k_{11}	$\dots$
-------	-------	-------	-------	-------	-------	-------	-------	-------	----------	----------	---------

$O(1)$ access, $O(n)$ space

Perfect for static settings where we can reindex
the input once

Bad for dynamic settings where keys change

More general:

(U for "universe", U large)

keys = n integers out of $\{1, \dots, U\}$



e.g. strings = #'s
via bit encodings

HELLO
ASCII 72 69 76 76 79
 $\Rightarrow 72 + 69 \times 128 + 76 \times 128^2 + 76 \times 128^3 + 79 \times 128^4$

Idea 1: Array $A[1, \dots, U]$

$O(1)$ lookup / insert / delete

$O(U)$ space ($\gggg O(n)$)

keys = n integers out of $\{1, \dots, U\}$

(U for "universe", U large)



e.g. strings = #'s
via bit encodings

HELLO
ASCII 72 69 76 76 79

$$\Rightarrow 72 + 69 \times 128 + 76 \times 128^2 + 76 \times 128^3 + 79 \times 128^4$$

Idea 1: Array $A[1, \dots, U]$

$O(1)$ lookup / insert / delete

$O(U)$ space ($\gggg O(n)$)

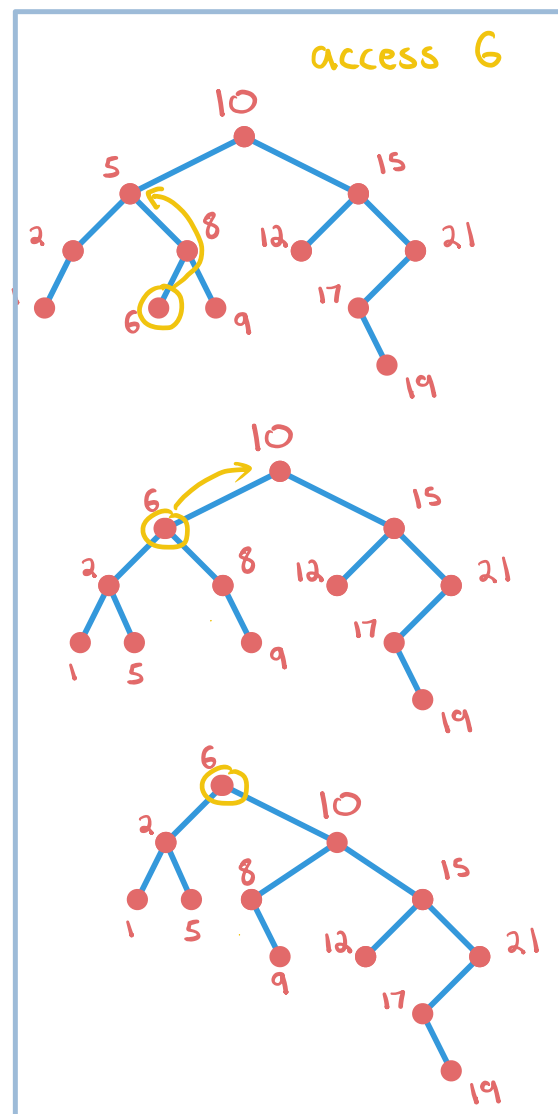
Idea 2: self-balanced trees

$O(n)$ space

$O(\log n)$ time per
lookup, insertion, deletion

(maybe amortized)

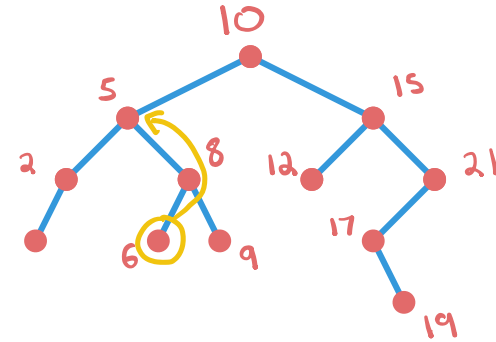
e.g. splay trees



Arrays



Search Trees



Time

$O(1)$

$O(\log n)$

Space

$O(1)$

$O(n)$

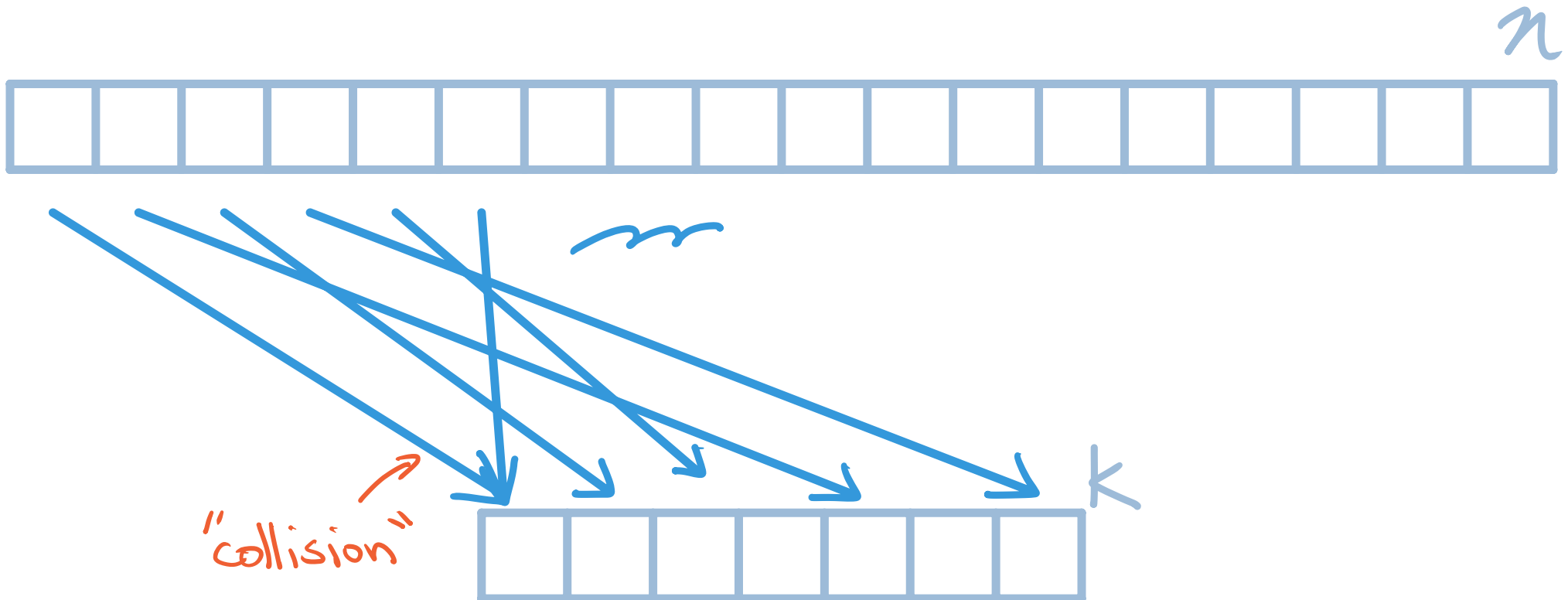
Question: can we get the **best** of both worlds?

Yes, with randomization

② Hash Functions

Randomly ^(initialized) constructed function

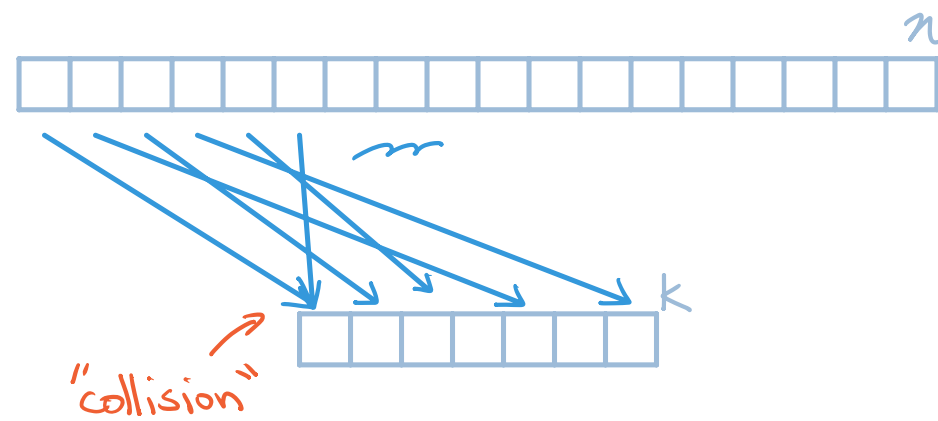
big $h: [n] \rightarrow [k]$ rel. small



② Hash Functions

Randomly constructed function

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



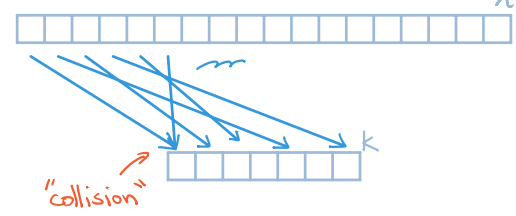
"Ideal Hash Function"

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

w/ each $h(i)$ indep. unif. random in $[k]$

↑
requires $\Omega(\log(k^n)) = \Omega(n \log k)$ space $\ddot{\smile}$

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

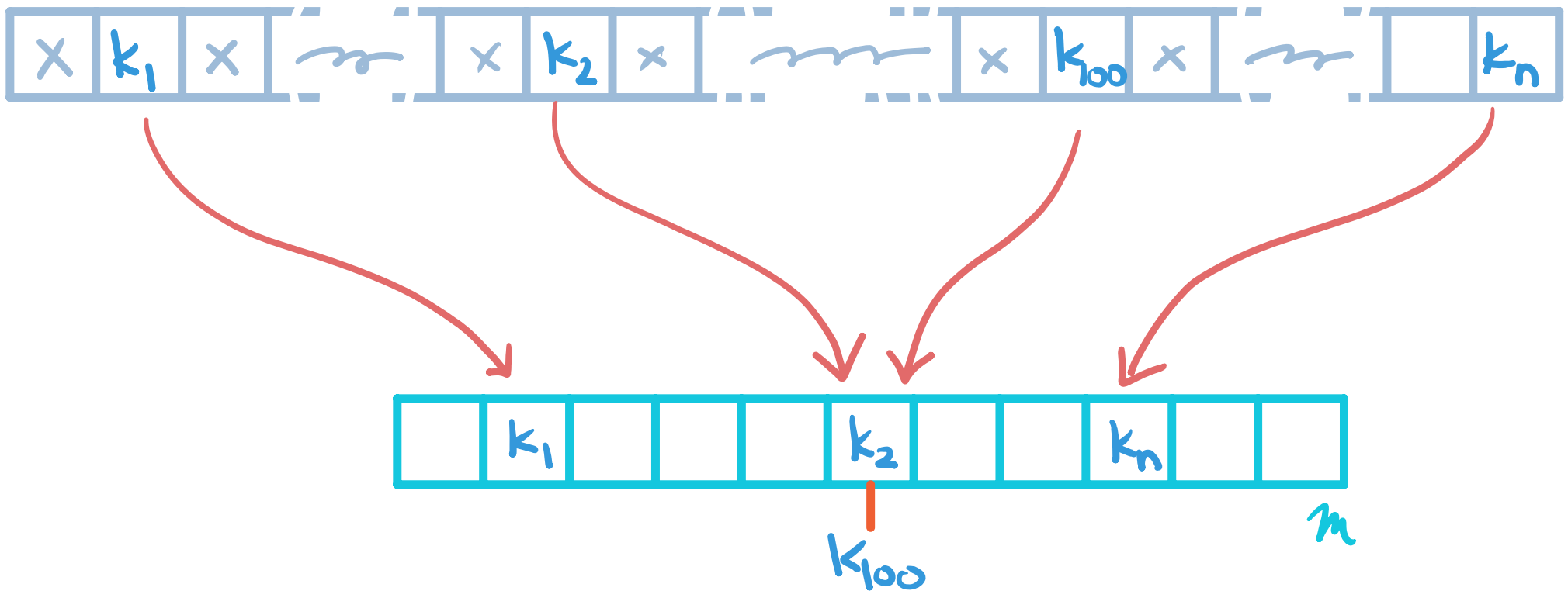
where

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

(proofs left as exercise)

Hashing

$$k_i \xrightarrow{\text{(is stored in)}} h(k_i) \bmod m$$



Goal: hash into a smaller array and achieve array-like performance

One problem: collisions

$$h(k_1) = h(k_2) \quad (\text{same slot})$$

Some ways to handle collisions

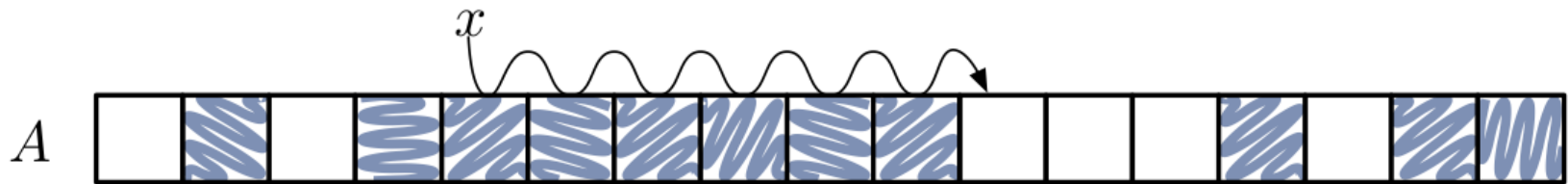
① make array big enough that there are (probably) no collisions

② "chaining": linked list at each cell

③ "cuckoo hashing" use two hash functions h_1 and h_2 .

if $h_1(k)$ and $h_2(k)$ already occupied, try to reassign one of the conflicts to their other hash, etc.

④ if $A[h(k)]$ filled, try $A[h(k)+1]$, $A[h(k)+2]$, ...



6
(and more)

"linear probing"

Chaining + Universal Hash Function

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

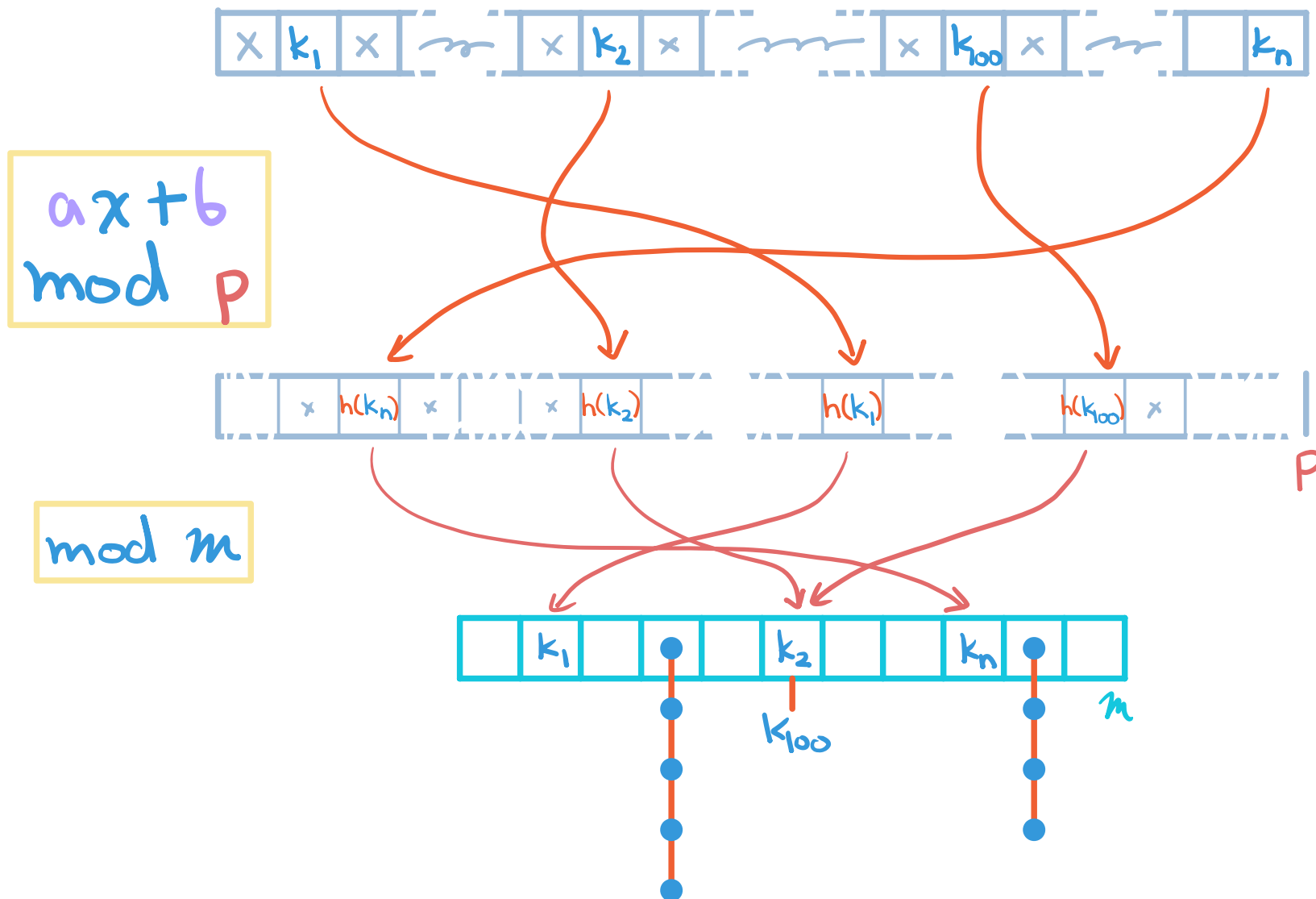
Universal Hash Functions

$h: [n] \rightarrow k$ 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 $\{1, \dots, p-1\}$ unifs. random
- b in $\{0, \dots, p-1\}$ unifs. random (proof left as exercise)

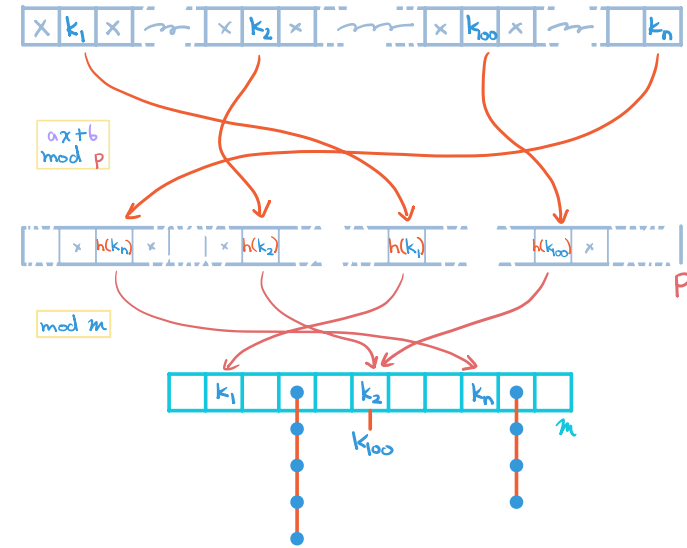


Theorem insert/delete/lookup take
 $O(1 + n/m)$ time in expectation

Proof

Chaining + universal hash

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



$n = \# \text{keys}$ $m = \text{size of array}$

Today:

② Randomized algorithms ("average" over random bits)

Probability

Hash functions,
Markov's Inequality

Algorithms

Hash tables,
Frequency Estimation,
Heavy Hitters (in streams)

Explore what
United States
is searching for
right now

allbirds

Explore

Search interest, past 24 hours



Why is allbirds trending?

G Search it



Struggling shoe retailer Allbirds
makes bizarre pivot from shoes t...

4 hours ago • CNBC



Allbirds says it's ditching footwear
and pivoting to become an AI...

30 minutes ago • CBS News

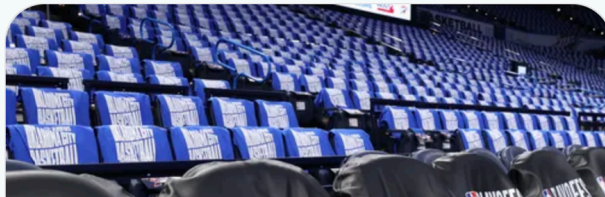


Allbirds stock soars more than
600% as the shoemaker rebrand...

48 minutes ago • Yahoo Finance

Dive deeper

Explore issues and events in detail. Curated by the Trends Data Team.



Gardening

See the sprouting Trends this spring

United States

Past 24 hours

All categories

All trends

By relevance

Export

Trends (Updated Apr 15, 2:17 PM)

Search volume

Started ⓘ

Trend breakdown ⓘ

Past 24 hours



allbirds

100K+
↑ 1,000%5 hours ago
↗ Activeallbirds stock all birds bird stock
[+ 6 more](#)

clavicular

200K+
↑ 1,000%16 hours ago
↗ Activeclavicular overdose overdose clav
[+ 11 more](#)

social security cola 2027

100K+
↑ 1,000%22 hours ago
↗ Active

2027 cola predictions



heat vs hornets

500K+
↑ 1,000%21 hours ago
↗ Activehornets vs heat hornets
charlotte hornets [+ 44 more](#)

johnny somali

50K+
↑ 1,000%16 hours ago
↗ Activejohnny somali sentence
what did johnny somali do [+ 3 more](#)

liv golf

20K+
↑ 1,000%16 hours ago
↗ Activeliv golf news liv liv golf shutting down
[+ 1 more](#)

trail blazers vs suns

200K+
↑ 1,000%17 hours ago
↗ Activedeni avdija suns suns vs trail blazers
[+ 29 more](#)

carnival tequila lawsuit verdict

20K+
↑ 1,000%9 hours ago
↗ Active

lsg vs rcb

20K+
↑ 800%4 hours ago
↗ Activelucknow super giants vs royal challengers...
rcb vs lsg [+ 3 more](#)

américa - nashville sc

100K+
↑ 1,000%15 hours ago
↗ Activeconcachampions
copa de campeones concacaf [+ 4 more](#)

rocky carroll returns ncis

20K+
↑ 700%20 hours ago
↗ Active

rahm dechambeau disappointing masters

20K+
↑ 700%16 hours ago
↗ Active

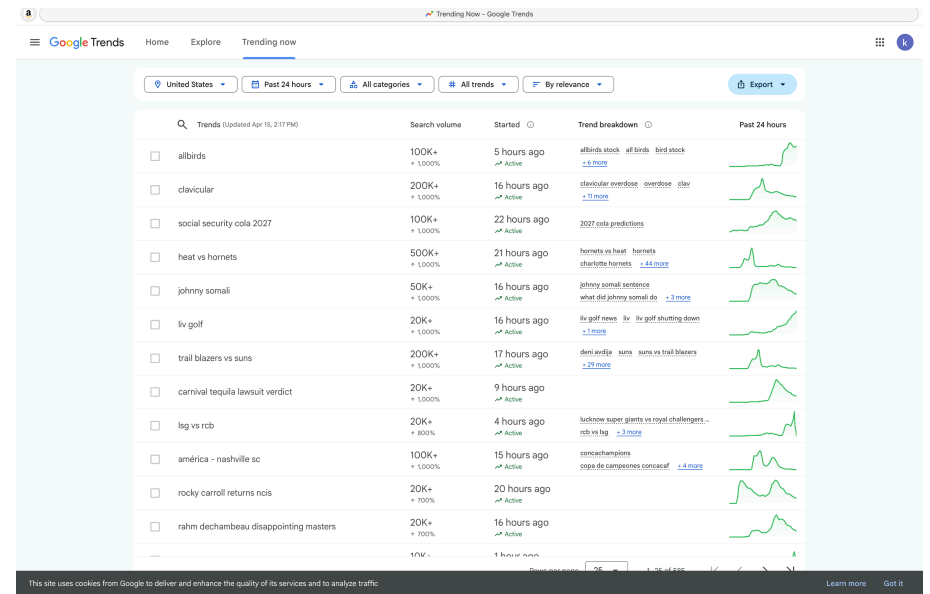
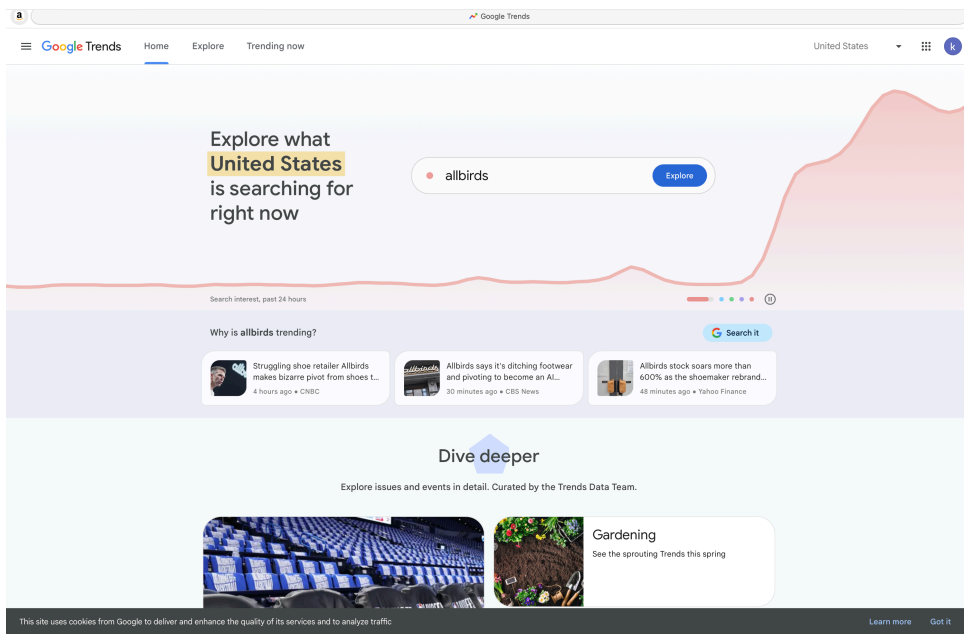
10K+

1 hour ago



Rows per page 25

1-25 of 535



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, \$

↑
small space

- 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 elements = $\{1, \dots, n\}$

- 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

rigorous, worst case bounds



Today:

Probability

Hash Functions,
Markov's Inequality

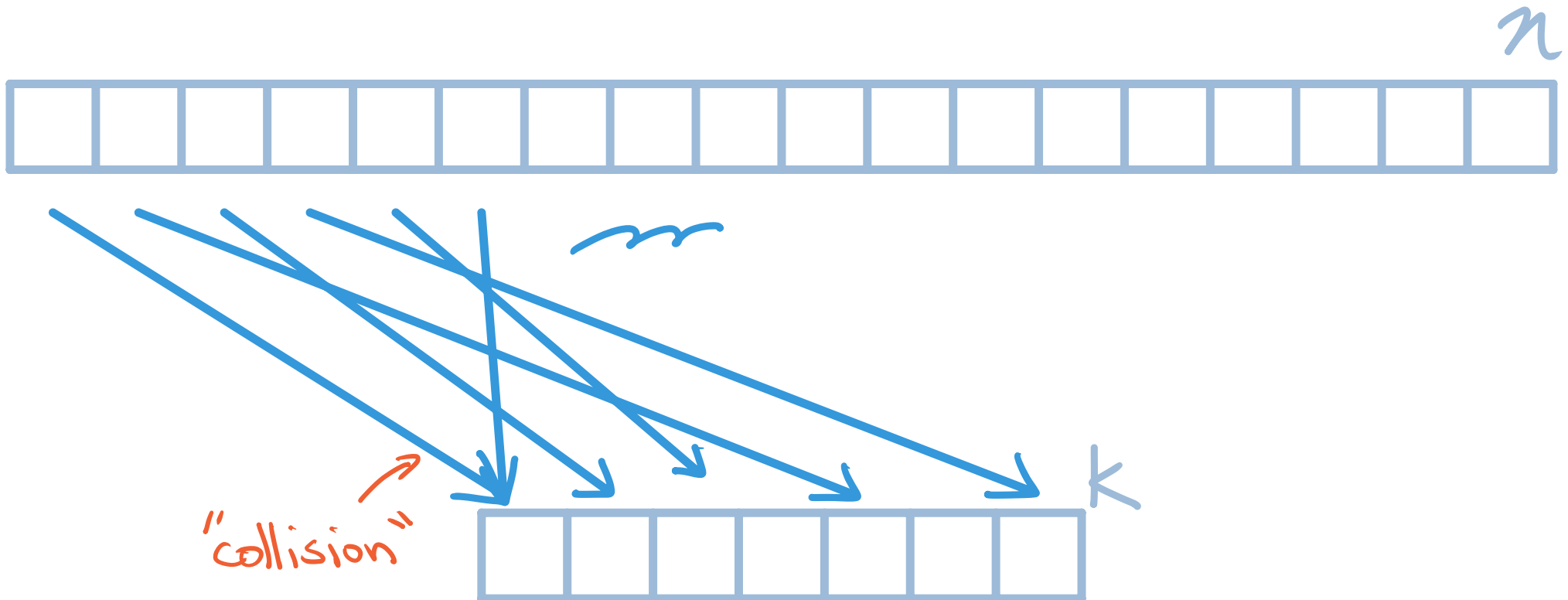
Algorithms

Frequency Estimation,
Heavy Hitters (in streams)

② Hash Functions

Randomly ^(initialized) constructed function

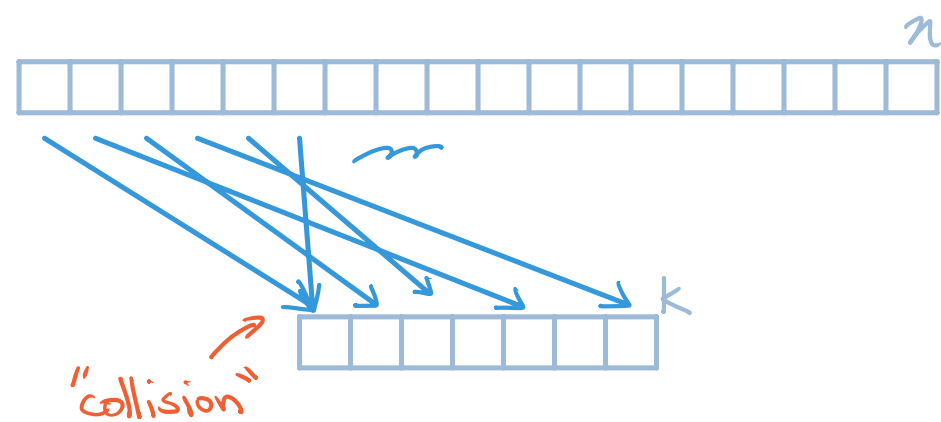
big $h: [n] \rightarrow [k]$ rel. small



② Hash Functions

Randomly constructed function

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



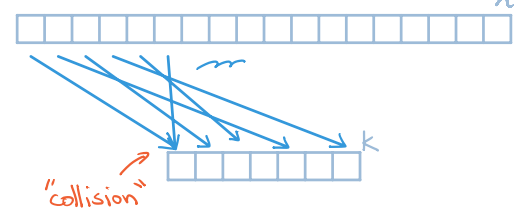
"Ideal Hash Function"

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

w/ each $h(i)$ indep. unif. random in $[k]$

↑
requires $\Omega(\log(k^n)) = \Omega(n \log k)$ space $\ddot{\smile}$

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)

Today:

② Randomized algorithms ("average" over random bits)

Probability

Hash Functions,

Universal Hash Functions

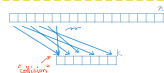
$h: [n] \rightarrow k$ s.t. for all pairs $i, j \in [n]$,
 $\Pr[h(i) = h(j)] \leq \frac{1}{k}$ (much weaker than ideal hash)

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

where • p prime, $> k$

• a in $\{1, \dots, p-1\}$ unifs. random

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



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 $\alpha \geq 1$,

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

Today:

Probability

Hash Functions,

Universal Hash Functions

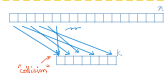
$h: [n] \rightarrow k$ s.t. for all pairs $i, j \in [n]$,
 $\Pr[h(i) = h(j)] \leq \frac{1}{k}$ (much weaker than ideal hash)

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

where • p prime, $> k$

• a in $\{1, \dots, p\}$ unis. random

• b in $\{0, \dots, p-1\}$ unis. 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, \$

↑
small space

- 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 elements = $\{1, \dots, n\}$

from now on:

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

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 $\epsilon > 0$)



Related Problem

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 $\geq \epsilon$

we will focus on estimating  up to ϵ -error

So to recap:

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

not enough space for all counters

lucky for us:

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

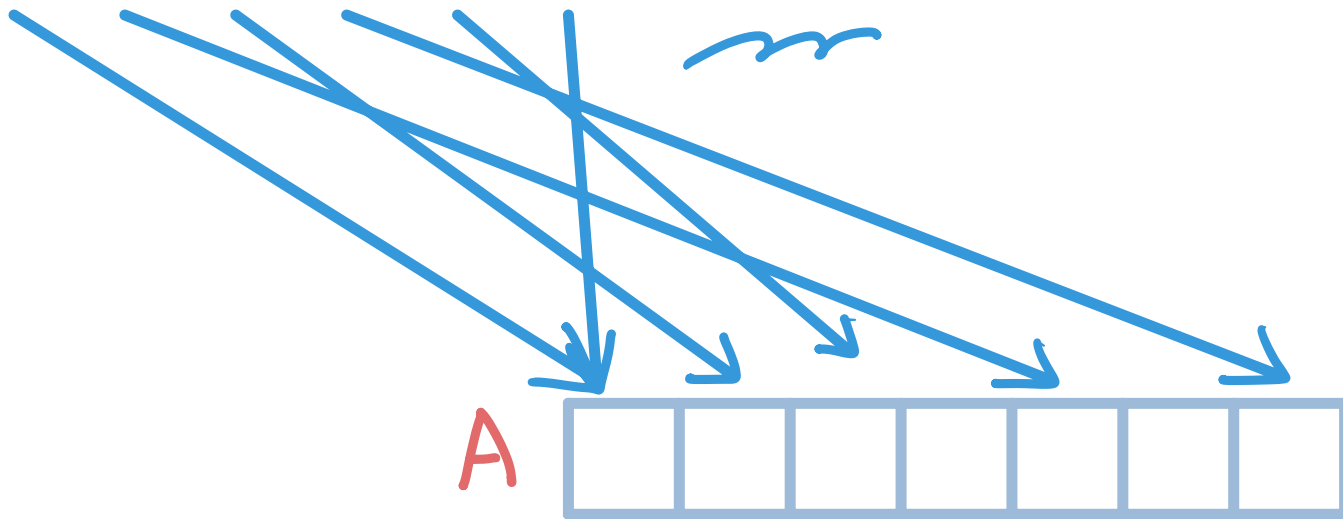
still: don't know which $\frac{1}{\epsilon}$ elements to count

Nashing has entered the chat

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

hash element i to counter $h(i)$

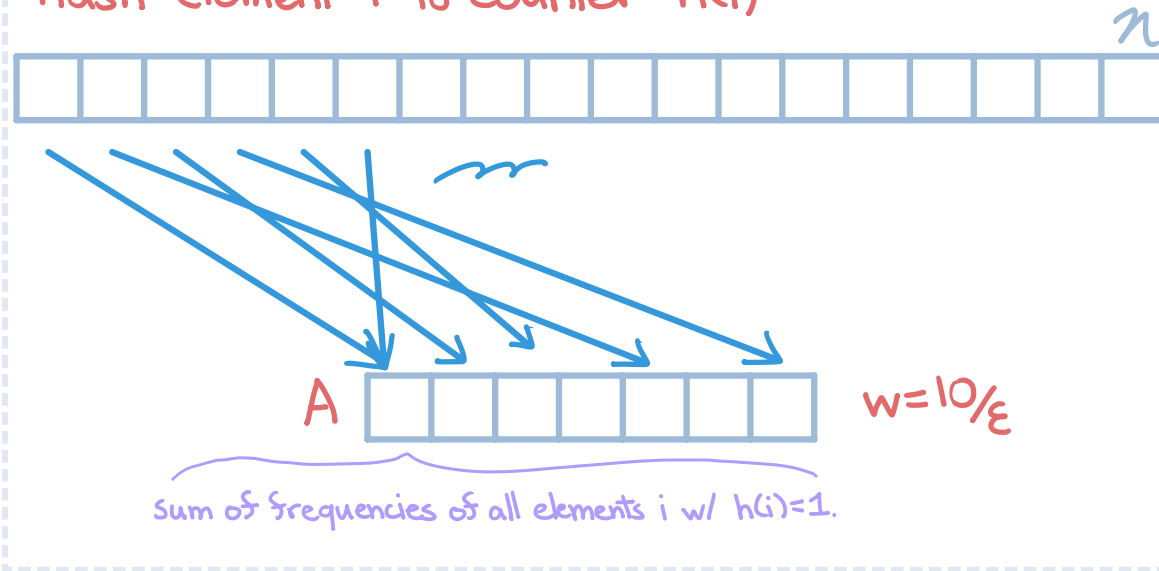
n



$w = 10/\epsilon$

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

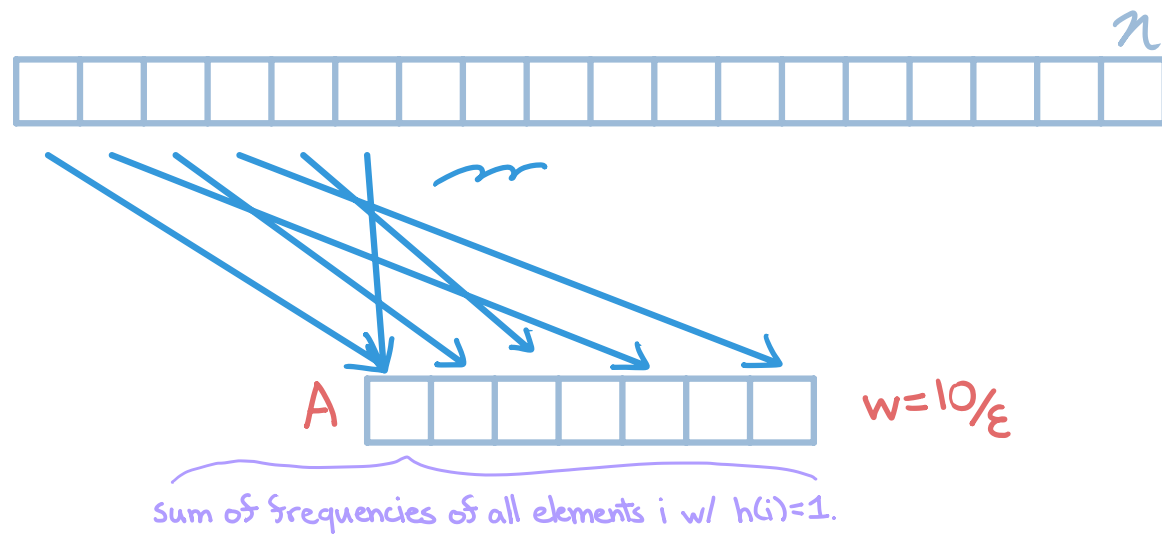
Idea: allocate $w = 10/\epsilon$ counters
hash element i to counter $h(i)$



To estimate frequency f_i of i :

return $h(i)$

(overcounts f_i)



Notation

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

f_i = (true) frequency of i

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

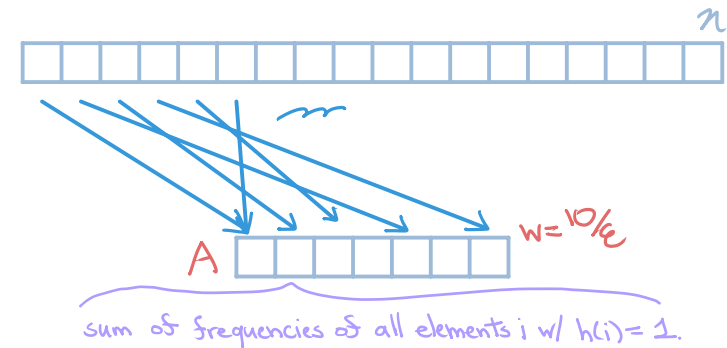
m = length of stream

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

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

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

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



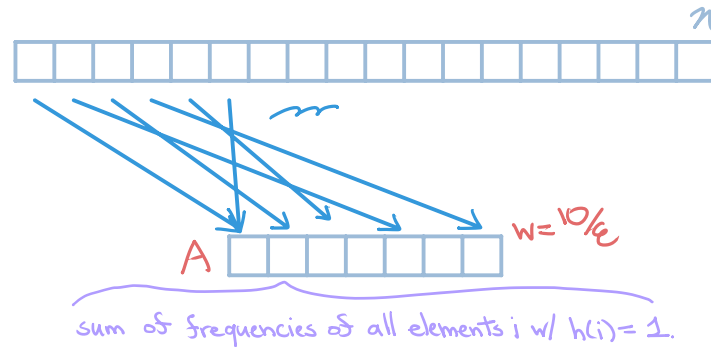
$A[1..w]$ = array of counters

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

bad
estimate

iff

ALL
hashed counters
are bad

prob $1/2^d$

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:

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

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

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
 $\uparrow$
prob $\frac{1}{2^d}$

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

Space:

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

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

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. $\geq 1 - \frac{1}{n^9}$

Probability

Hash Functions,

Universal Hash Functions

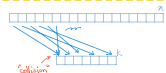
$h: [n] \rightarrow k$ s.t. for all pairs $i, j \in [n]$,
 $\Pr[h(i) = h(j)] \leq \frac{1}{k}$ (much weaker than ideal hash)

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

where • p prime, $> k$

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

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



Markov's Inequality

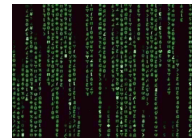
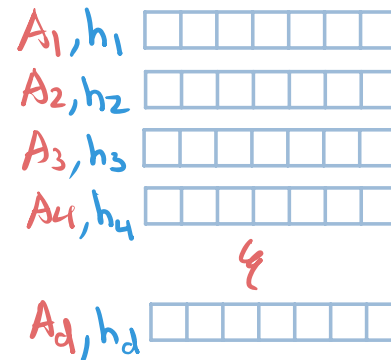
Markov's Inequality (special case)

Let $X \geq 0$ be nonnegative RV

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

Algorithms

Frequency Estimation, Heavy Hitters (in streams)



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

