CS5112: Algorithms and Data Structures for Applications

Streaming algorithms

Ramin Zabih

Some content from: Wikipedia;
J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets, http://www.mmds.org

Cornell University

CORNELL TECH

Administrivia

- Prelim feedback
- Reduction (duck) problem
  - NP-completeness did not figure in the problem
    - Answers that used NP-completeness were usually incorrect
  - An NP-hard problem does not really have a lower bound, but we gave students credit for writing $O(1)$ or $O(n)$
    - The $O(n \log n)$ lower bound is for a problem that can solve sorting
    - Writing a polynomial lower bound is incorrect, such lower bounds include $O(n^3), O(n^{5112}), \ldots$

Non-binding grade estimates from prelim

- The below is **NOT** a promise, just a guess
- Assuming the class follows the same curve as in 2018:
  - About $\frac{1}{2}$ A's of some kind, mostly A-
    - prelim score 80-100
  - About $\frac{1}{3}$ B's of some kind
    - prelim score 60-80
  - About $\frac{1}{6}$ lower grades
    - prelim score <60

Lecture Outline

- Online data processing
- Histogram properties
  - Majority estimation via Boyer Moore
  - Misra-Gries generalization
- Hashing-based streaming algorithms
  - Flajolet-Martin
  - Bloom filters

Online algorithms and approximations

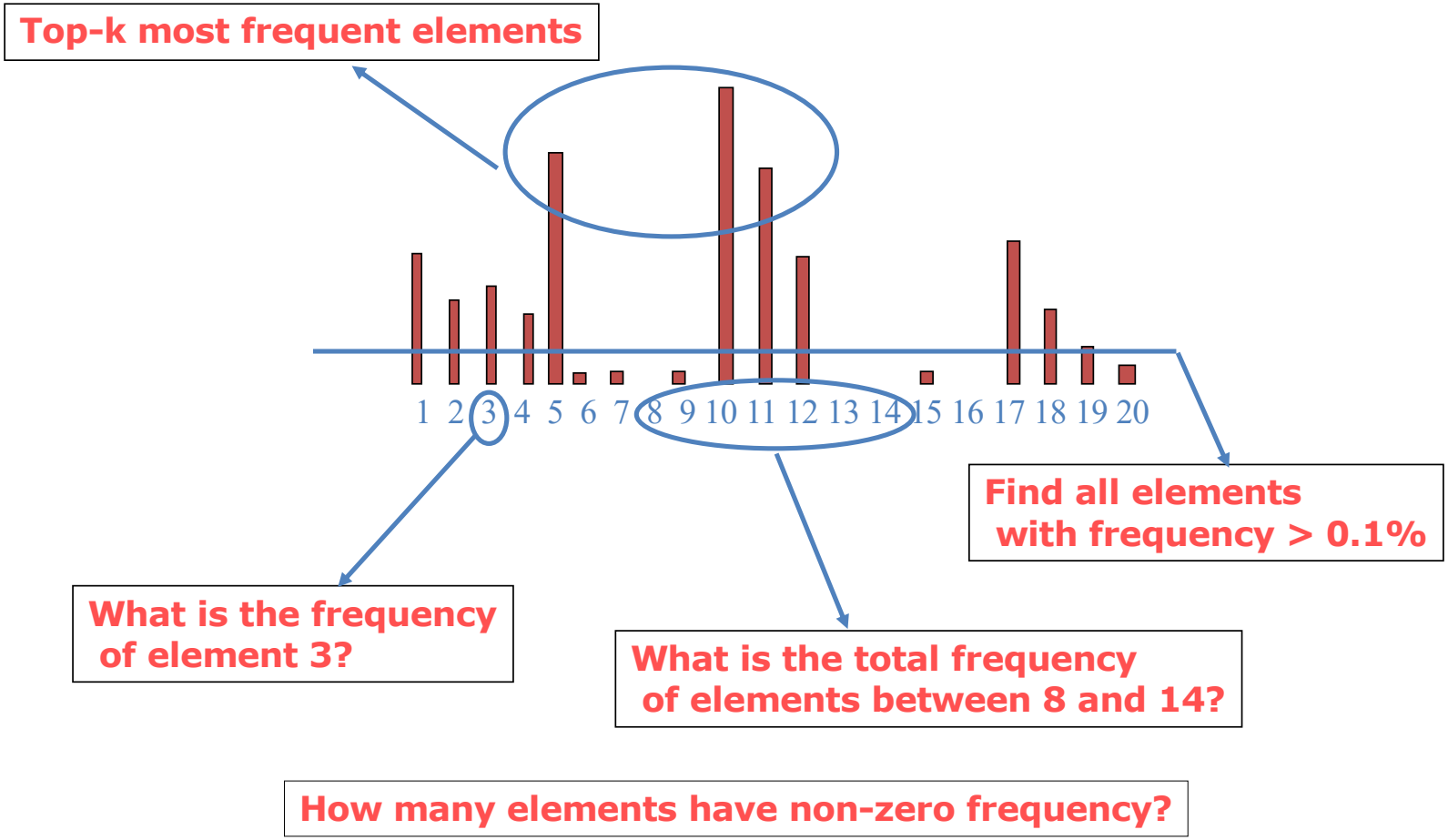
- Many AI/ML applications hinge on understanding the distribution of your input data
  - Classification is just one example (is this a cat video?)
- Classically we assume that all the input data is available
  - You can run an offline algorithm over it
  - Then do NN classification, etc.
- This assumption is often wrong: data comes streaming in
  - And you cannot afford to store the entire data set

Welford's online mean algorithm

- Suppose we know that cats come from a Gaussian distribution but we have too many cats to store all their weights
- Can we estimate the mean (and variance) online?
- Online algorithms are very important for modern applications
- For the mean we have

$$\mu_n = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Some natural histogram queries



Why approximation?

- Suppose you want an (exact) histogram of your data
- This requires space linear in the number of data points
- This is intractable for many internet applications!
  - Examples:
    - IP addresses for DDOS detection
    - Most popular page/item to buy
    - Is this web site worth caching or is it a ‘one hit wonder’?

Streaming algorithms

- We will use a small amount of space but solve the problem approximately
  - But, not precisely the same problem
- Instead of computing the histogram we will look at several key properties of a histogram we can efficiently approximate
  - Typically with constant or logarithmic space and time

Relevant quantities to compute

- **Majority**: if there is a single item comprising more than half the input stream, find it
- **Frequent items**: find all items that comprise more than a given percent ϕ of the input stream
  - Approximate version: find all items that comprise between $\phi - \epsilon$ and ϕ percent of the input stream (exact when $\epsilon = 0$)
  - Recent version: find and update the most recent popular items
- **Distinct items**: how many different items are there?
- **Membership**: have we seen this item before?

1. Boyer-Moore majority algorithm

- Majority: one element is more than half the population
  - Easy offline. How do we do this online??
- Algorithm according to Moore:

> As we sweep we maintain a pair consisting of a *current candidate* and a counter. Initially, the current candidate is unknown and the counter is 0.
>
> When we move the pointer forward over an element *e*:
>
> - If the counter is 0, we set the current candidate to *e* and we set the counter to 1.
> - If the counter is not 0, we increment or decrement the counter according to whether *e* is the current candidate.
>
> When we are done, the current candidate is the majority element, if there is a majority.

Source

Boyer-Moore algorithm example

Source

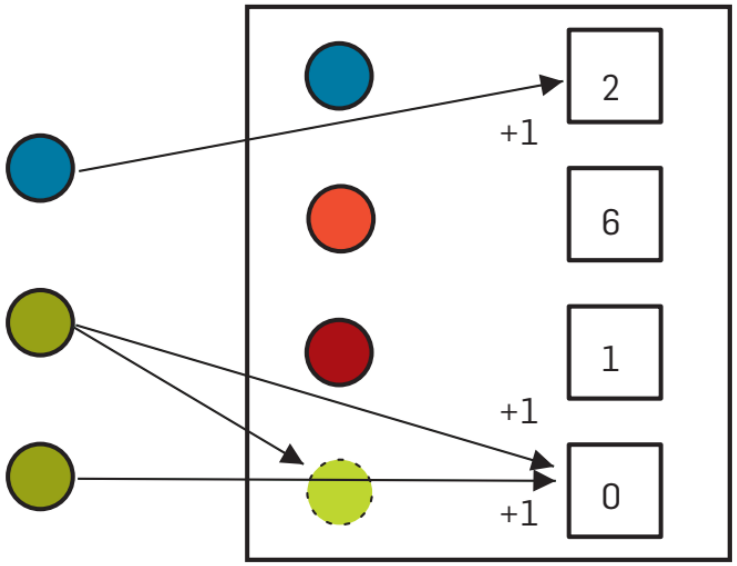
CORNELL TECH

2. Misra-Gries

- Generalization of Boyer-Moore majority algorithm
- Store $k-1$ counters, for a parameter k
  - Larger k means more space and accuracy
- Any item that appears more than $\frac{n}{k}$ times in the input stream of size n will be present when the algorithm terminates
- If $k = 1/\epsilon$ then each count is at most ϵn below its true value

Misra-Gries algorithm in action

Figure 2. Counter-based data structure: the blue (top) item is already stored, so its count is incremented when it is seen. The green (middle) item takes up an unused counter, then a second occurrence increments it.



Algorithm 1: FREQUENT(k)

```
n ← 0;
T ← ∅;
foreach i do
    n ← n + 1;
    if i ∈ T then
        c_i ← c_i + 1;
    else if |T| < k−1 then
        T ← T ∪ {i};
        c_i ← 1;
    else forall j ∈ T do
        c_j ← c_j − 1;
        if c_j = 0 then T ← T \ {j};
```

Finding the Frequent Items in Streams of Data, Cormode and Hadjieleftheriou, CACM 52(10), October 2009

CORNELL TECH

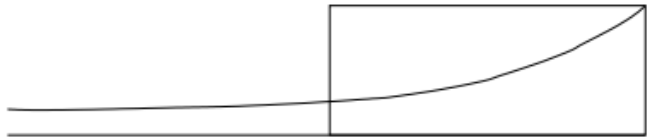
3. Find popular recent items

- Want to be able to naturally update this over time
  - Think of popular: movies, shopping items, web pages, etc.
- We could run, e.g., Misra-Gries on a sliding window
  - This is both impractical and wrong
- Wrong because the importance of an item should not “fall off a cliff” when it moves outside of our window
- Our input will be, for an item, a series of counts
  - 100 people bought this yesterday, 50 people the day before, etc.

Weighted average in a sliding window

- Computing the average of the last k inputs can be viewed as a dot product with a constant vector $v = \left[\frac{1}{k}, \frac{1}{k}, \ldots, \frac{1}{k}\right]$
- Sometimes called a box filter
  - Easy to visualize
- This is also a natural way to smooth, e.g., a histogram
  - To average together adjacent bins, $v = [\frac{1}{3}, \frac{1}{3}, \frac{1}{3}]$
- This kind of weighted average has a famous name

Decaying windows



- Let our input at time t be $\{a_1, a_2, \ldots, a_t\}$
- With a box filter over all of these elements we computed

$$\frac{1}{t}\sum_{i=0}^{t-1} a_{t-i}$$

- Instead let us pick a small constant c and compute

$$\hat{a}_t = \sum_{i=0}^{t-1} a_{t-i}(1-c)^i$$

Easy to update this

- Update rule is simple, let the current dot product be $\hat{a}_t$

$$\hat{a}_{t+1} = (1 - c)\hat{a}_t + a_{t+1}$$

- This downscales the previous elements correctly, and the new element is scaled by $(1 - c)^0 = 1$
- This avoids falling off the edge
- Gives us an easy way to find popular items

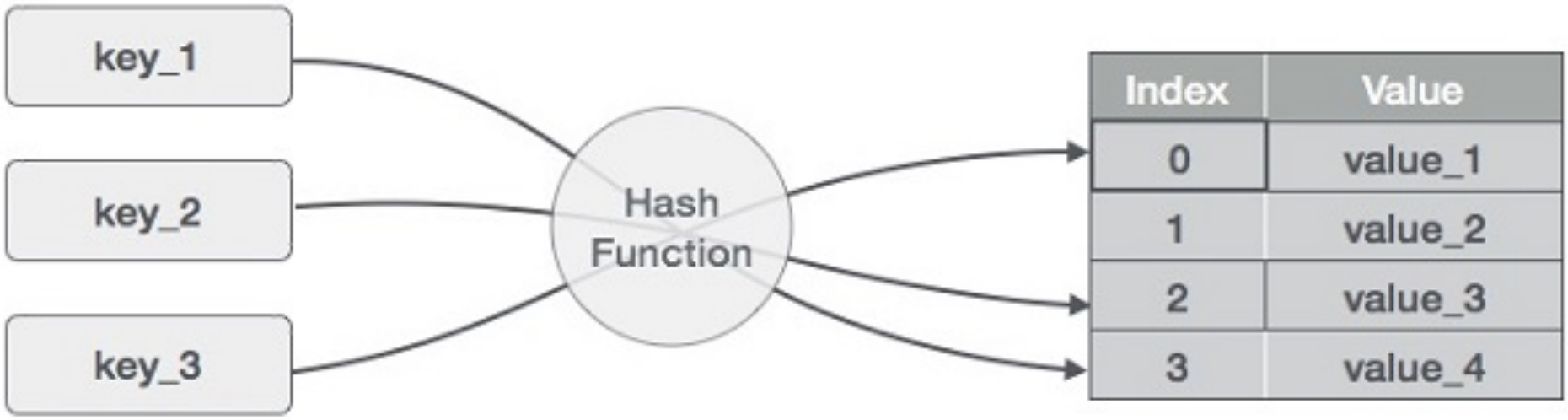
4. Popular items with decaying windows

- We keep a small number of weighted sum counters
- When a new item arrives for which we already have a counter, we update it using decaying windows, and update all counters
- How do we avoid getting an unbounded number of counters?
- We set a threshold, say ½, and if any counter goes below that value we throw it away
- The number of counters is bounded by $\frac{2}{c}$

How many distinct items are there?

- This tells you the size of the histogram, among other things
- To solve this problem exactly requires space that is linear in the size of the input stream
  - Impractical for many applications
- Instead we will compute an efficient estimate via hashing

Hashing in one diagram



What makes a good hash function?

- Almost nobody writes their own hash function
  - Like a random number generator, very easy to get this wrong!
- Deterministic
- Uniform
  - With respect to your input!
  - Technical term for this is entropy
- (Sometimes) invariant to certain changes
  - Example: punctuation, capitalization, spaces

Examples of good and bad hash functions

- Suppose we want to build a hash table for CS5112 students
- Is area code a good hash function?
- How about zip code?
- Social security numbers?
  - https://www.ssa.gov/history/ssn/geocard.html
- What is the best and worst hash function you can think of?