CS 4973: Advanced Algorithms	Spring 2025

Lecture 14 — February 26, 2025

Prof. Prashant Pandey *Scribe: Dhruv Chauhan*

1 Overview

In the last lecture we finished talking about filters as a data structure by discussing adaptive filters. For data structures throughout the course (up until this point), we assumed a large universe of size U (normally 2^{64}), and tried to store the n number of keys in a data structure of size $O(n)$.

In this lecture we will move on to another type of data structure called sketches, which is motivated by the problem that now n, the number of keys, is also too large for us to store in memory or even on disk.

2 Frequency Estimation in a Stream

Goal: Count all items, so that we can return the frequency of any query $q \in [U]$

The universe $[U]$ the keys come from is too big to store in memory (typically $U = 2^{64}$).
NOTE: usually lgU is a constant (in the above example, $lgU = 64$).

Stream: $X = (x_1, x_2, \ldots, x_n)$ where $x_i \in [U]$ for all $i \in [n]$

Count: $c(q) = \sum_{x \in X} R(x, q)$
Where R is an indicator variable which is 1 if $x = q$ and 0 otherwise.

Frequency: $f(q) = \frac{c(q)}{n}$ (the ratio of count to total keys in this context)

Space bound: The lower bound to accurately store the exact count and/or frequency of keys is $\Omega(n)$ space.

Assumptions:

- We cannot store whole data in memory (but can use sublinear space - lgn, lg^2n)
- For each key in the stream, we can only process it once (some use cases allow for two passes, but we will focus on once)

Use case: Even though a router cannot keep a count for the number of times each IP address is requested (limited memory), it would be useful to detect high-frequency IP addresses to prevent DoS attacks.

3 Approximate Version of Frequency

$\hat{f}_\varepsilon$, the ε-approximation of f, such that for all $q \in [U]$ is defined as

$$f(q) - \varepsilon \leq \hat{f}_\varepsilon \leq f(q) + \varepsilon$$

4 Heavy Hitter in Stream

A related problem to frequency estimation is finding the ϕ-heavy hitter(s) in a stream.
ϕ-heavy hitter: $y \in [U]$ is a ϕ-heavy hitter *iff* $f(y) > \phi$

The ε-approximation of ϕ-heavy hitter $\hat{H}_\varepsilon^\phi$ is defined as

- $y \in \hat{H}_\varepsilon^\phi$ if $f(y) > \phi$
- $y \notin \hat{H}_\varepsilon^\phi$ if $f(y) > \phi - \varepsilon$

If $\phi - \varepsilon \leq f(y) \leq \phi$, then y may or may not be in $\hat{H}_\varepsilon^\phi$,

4.1 Majority Finding

A version of ϕ-heavy hitting, where $\phi = 0.5$, is called majority finding. The goal is to find y where $f(y) > 0.5$(if it exists).

The intuition behind solving this in one pass is to keep track of an element, every time you see the same element, increase a counter you kept track of by 1. If we see a different element, we decrease said counter by 1. Once the counter is 0, we should set the tracked element as the element most recently seen. The idea is that if an element is the majority element, once the counter resets (which will only happen by seeing at least as many non-majority elements as the majority element), the majority element is still the majority in the rest of the stream.

Below is the algorithm for majority finding (Boyer & Moore [1])

Algorithm 1 Majority(X)

```
y ← NaN
c ← 0
for all x ∈ X do
    if x == y then                    ▷ if next key same as current key increment counter
        c ← c + 1
    else if c == 0 then ▷ if counter is 0, then make current key as candidate majority element
        y ← x
        count ← 1
    else                              ▷ if next key different than current key decrement counter
        count ← count − 1
    end if
end for

return y
```

NOTE: If there is a majority element, this algorithm will return it. If there is no majority element, the algorithm will return junk. To fix this, we can do a second pass to verify that the key being returned is in fact the majority element.

4.2 Find ϕ-heavy hitters

An extension of majority finding, the new problem to solve is finding all keys that are ε-approximation of ε-heavy hitter ($\phi = \varepsilon$ for simplicity), i.e, finding keys $y \in \hat{H}_\varepsilon^\varepsilon$ ($y \in \hat{H}_\varepsilon^\varepsilon$ if $f(y) > \varepsilon$). The below algorithm by Misra-Gries [2] can be thought of as an extension of the above algorithm, and solves this problem.

Algorithm 2 Misra-Gries(X, k)

```
Y ← [NaN] * k, C ← [0] * k

for all x ∈ X do
    if ∃i(Y[i] = x) then                    ▷ if key already being counted, increase by 1
        C[i] ← C[i] + 1
    else if ∃i(C[i] = 0) then               ▷ if key not being counted, start counting it
        Y[i] ← x, C[i] ← C[i] + 1
    else        ▷ if none of the keys being counted are this key, then decrease each count
        for each i do
            C[i] ← C[i] − 1
        end for
    end if
end for
return Y, C
```

To guarantee the bounds for a ε-heavy hitter, k (the length of Y and C) should be some function of $\frac{1}{\varepsilon}$.

5 Count Sketch

For a fixed $\phi = \varepsilon$, we are able to use the Misra-Gries algorithm to find the ε-heavy hitters in a stream of keys. However, what if we wanted to find ϕ-heavy hitters for different values of ϕ. We will need to somehow store an approximation of the frequency for each key encountered.

One way of doing this is by mapping each key in U to a smaller space K using a hash function, and incrementing the count of an index when a key hashes to a particular index. When we want to find the approximate count, we will then query the index that we get by hashing the key, and return the count. This, however, is not a great approach, since a low count and high count key could collide and thus create a very bad estimate for the lower count key

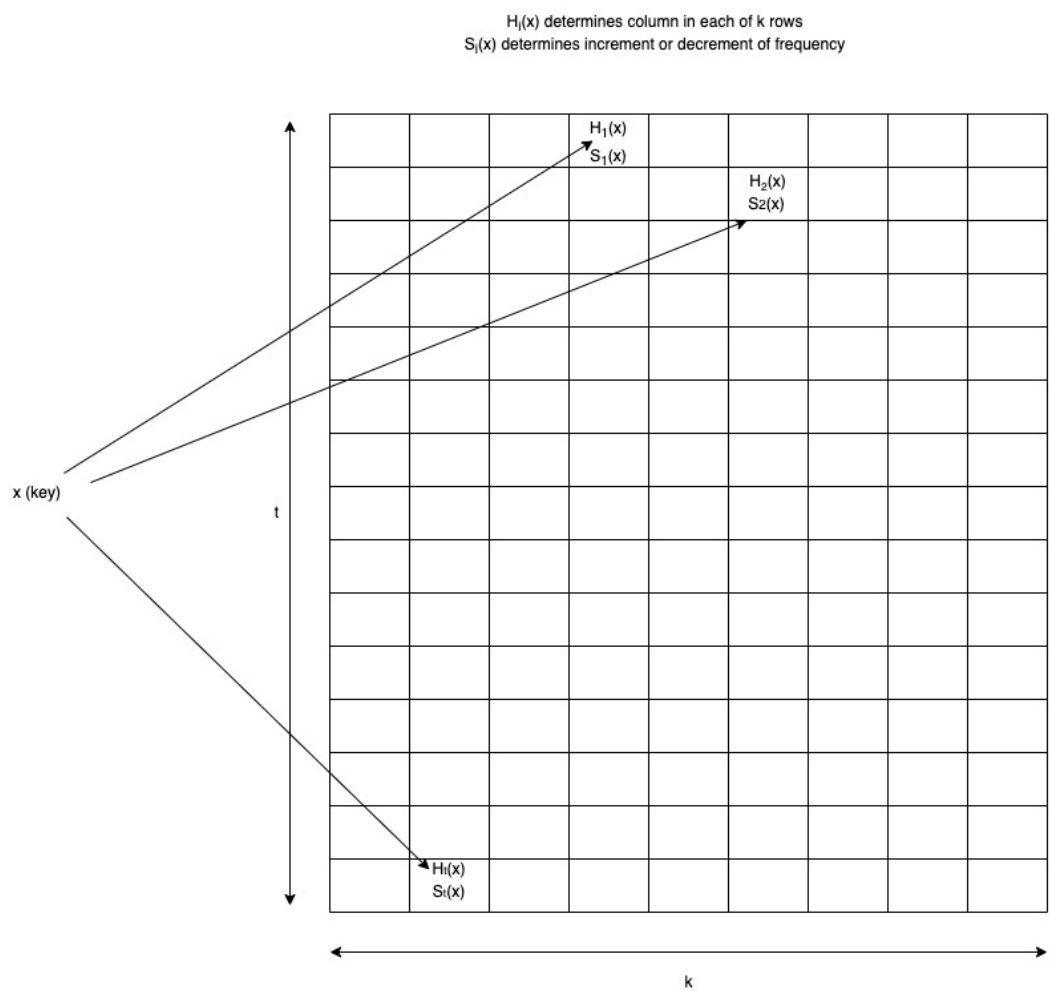
What if instead of hashing once, we hash t times, update t indices, and return the median of the frequencies when queried for the frequency. In this case, we are bounding the probability of "bad" collisions to be very low.

Also, since we use the median, we want to use an unbiased estimator. Currently, since we are only incrementing the count for each index, we will overestimate the frequency of each key. Instead of this approach, once hashed to an index, we will use another hash function S to either increment or decrement the count.

To sum everything up, for each key, we will hash it t times to find t indices (to bound probability of a bad collision). To increment or decrement each of these indices, we will also use t **pairwise independent hash functions** S to determine that.

Setup:

- Have a count matrix C with $t \times k$ 0s.
- Set of t-hash functions ($H_i : [u] \to [k]$)
- Another of t-hash functions ($S_i : [u] \to [\pm 1]$)



Algorithm 3 Count-Sketch(X, t, k)

```
C ← 0^{t×k}
H ← (H_i : [u] → [k])_{i=1}^t
S ← (S_i : [u] → [±1])_{i=1}^t
for all x ∈ X do
    for i in [t] do
        j ← H_i(x)
        C_{i,j} ← C_{i,j} + S_i(x)
    end for
end for
return C, H, S
```

Query: For $q \in [U]$, calculate t different frequency approximations.

$\hat{f}_i(q) = \frac{1}{n} S_i(q) C_{i,H_i(q)}$

$\hat{f}_{CS}(q) = median_{i \in [t]} \hat{f}_i(q)$

6 Count-Min Sketch

Now, instead of an unbiased estimator, what if we have an (biased) overestimator. We do the same thing, but just remove the S hash functions and just increment the counts, and take the minimum frequency approximation instead of the median. That is what count-min sketch is.

Query: For $q \in [U]$

$\hat{f}_i(q) = \frac{1}{n} H_i(q)$

$\hat{f}_{CS}(q) = min_{i \in [t]} \hat{f}_i(q)$

Algorithm 4 Count-Min(X, t, k)

```
C ← 0^{t×k}, H ← (H_i : [u] → [k])_{i=1}^t
for all x ∈ X do
    for all i in [t] do
        C_{i,H_i(x)} ← C_{i,H_i(x)} + 1
    end for
end for
return C, H
```

NOTE: This is an overestimator since we always increment the count.

7 Hyperparameter Intuition

Since we are using hashing, we cannot always guarantee that the frequency estimations will always be within ε. Rather, we can guarantee with high probability (let's assume p), we can guarantee ε-approximation of f.

The number of columns (k) of the table controls the value of ε. The higher k is, the lower ε is, because there will be lesser collisions (as more columns to hash to).

The number of columns (t) of the table controls the value of p. The more rows there are, the higher the chance one of the indices a key hashes to has its ε-approximate frequency.

References

[1] Boyer, R.S., Moore, J.S. (1991). MJRTY—A Fast Majority Vote Algorithm. In: Boyer, R.S. (eds) Automated Reasoning. Automated Reasoning Series, vol 1. Springer, Dordrecht. *Springer, Dordrecht*, 10.1007/978-94-011-3488-0_5

[2] David Gries, J Misra. Finding Repeated Elements *J. Science of Computer Programming*, 10.1016/0167-6423(82)90012-0.