*Casus ubique valet; semper tibi pendeat hamus:*
*Quo minime credas gurgite, piscis erit.*

*[Luck affects everything. Let your hook always be cast.*
*Where you least expect it, there will be a fish.]*

— Publius Ovidius Naso [Ovid], *Ars Amatoria*, Book III (2 AD)

*There is no sense being precise*
*when you don't even know what you're talking about.*

— Attributed to John von Neumann

# 6 Filtering and Streaming

The randomized algorithms and data structures we have seen so far *always* produce the correct answer but have a small probability of being slow. In this lecture, we will consider randomized algorithms that are always fast, but return the wrong answer with some small probability.[1] More generally, we are interested in tradeoffs between the (likely) efficiency of the algorithm and the (likely) quality of its output.

Specifically, we introduce an ***error rate*** $\delta$ and analyze the running time required to guarantee the output is correct with probability $1-\delta$. For "high probability" correctness, we need $\delta < 1/n^c$ for some constant $c$. In practice, it may be sufficient (or even necessary) to set $\delta$ to a small constant; for example, setting $\delta = 1/1000$ means the algorithm produces correct results at least 99.9% of the time.

## 6.1 Bloom Filters

Bloom filters are a natural variant of hashing proposed by Burton Bloom in 1970 as a mechanism for supporting membership queries in sets. In strict accordance with Stigler's Law of Autonomy, Bloom filters are identical to ***Zatocoding***, a coding system for library cards developed by Calvin Mooers in 1947. (Mooers was the person who coined the phrase "information retrieval".) A probabilistic analysis of Zatocoding appears in the personal notes of cybernetics pioneer W. Ross Ashby from 1960.

A Bloom filter (or Zatocode) for a set $X$ of $n$ items from some universe $\mathcal{U}$ allows one to test whether a given item $x \in \mathcal{U}$ is an element of $X$. Of course we can already do this with hash tables in $O(1)$ expected time, using $O(n)$ space. Bloom (and Mooers) observed that by allowing false positives—occasionally reporting $x \in X$ when in fact $x \notin X$—we can still answer queries in $O(1)$ expected time using considerably less space. False positives make Bloom filters unsuitable as an exact membership data structure, but because of their speed and low false positive rate, they are commonly used as filters or sanity checks for more complex data structures.

A Bloom filter consists of an array $B[0\,..\,m-1]$ of bits, together with $k$ hash functions $h_1, h_2, \ldots, h_k \colon \mathcal{U} \to \{0, 1, \ldots, m-1\}$. For purposes of theoretical analysis, we assume the hash functions $h_i$ are mutually independent, ideal random functions. This assumption is of course unsupportable in practice, but may be necessary to guarantee theoretical performance. Unlike many other types of hashing, nobody knows whether the same theoretical guarantees can be achieved using practical hash functions with more limited independence.[2] Fortunately, the actual

[1] Some textbooks (and Wikipedia) use the terms "Las Vegas" and "Monte Carlo" algorithms to respectively describe these two types of randomized algorithms. Most algorithms researchers don't.

[2] PhD thesis, anyone?

real-world behavior of Bloom filters appears to be consistent with this unrealistic theoretical analysis.

A Bloom filter for a set $X = \{x_1, x_2, \dots, x_n\}$ is initialized by setting the bit $B[h_j(x_i)]$ to 1 for all indices $i$ and $j$. Because of collisions, some bits may be set more than once, but that's fine.

```
MakeBloomFilter($X$):
  for $h \leftarrow 0$ to $m-1$
      $B[h] \leftarrow 0$
  for $i \leftarrow 1$ to $n$
      for $j \leftarrow 1$ to $k$
          $B[h_j(x_i)] \leftarrow 1$
  return $B$
```

Given a new item $y$, the Bloom filter determines whether $y \in X$ by checking each bit $B[h_j(y)]$. If any of those bits is 0, the Bloom filter correctly reports that $y \notin X$ However, if all bits are 1, the Bloom filter reports that $y \in X$, although this is not necessarily correct.

```
BloomMembership($B, y$):
  for $j \leftarrow 1$ to $k$
      if $B[h_j(y)] = 0$
          return False
  return Maybe
```

One nice feature of Bloom filters is that the various hash functions $h_i$ can be evaluated in parallel on a multicore machine.

## 6.2 False Positive Rate

Let's estimate the probability of a false positive, as a function of the various parameters $n$, $m$, and $k$. For all indices $h$, $i$, and $j$, we have $\Pr[h_j(x_i) = h] = 1/m$, so ideal randomness gives us

$$Pr[B[h] = 0] \;=\; \left(1 - \frac{1}{m}\right)^{kn} \;\approx\; e^{-kn/m}$$

for every index $h$. Using this exact probability is rather unwieldy; to keep things sane, we will use the close approximation $\boldsymbol{p := e^{-kn/m}}$ instead.

The expected number of 0-bits in the Bloom filter is approximately $mp$; moreover, Chernoff bounds imply that the number of 0-bits is close to $mp$ with very high probability. Thus, the probabililty of a false positive is very close[3] to

$$(1-p)^k = (1 - e^{-kn/m})^k.$$

If all other parameters are held constant, then the false positive rate increases with $n$ (the number of items) and decreases with $m$ (the number of bits). The dependence on $k$ (the number of hash functions) is a bit more complicated, but we can derive the best value of $k$ for given $n$ and $m$ as follows. Consider the logarithm of the false-positive rate:

$$\ln\left((1-p)^k\right) = k \ln(1-p) = -\frac{m}{n} \ln p \ln(1-p).$$

By symmetry, this expression is minimized when $\boldsymbol{p = 1/2}$. We conclude that, the optimal number of hash functions is $\boldsymbol{k = \ln 2 \cdot (m/n)}$, which would give us the false positive rate

[3] This analysis, originally due to Bloom, assumes that certain events are independent even though they are not; as a result, the estimate given here is slightly below the true false positive rate.

$(1/2)^{\ln 2(m/n)} \approx (0.61850)^{m/n}$. Of course, in practice, $k$ must be an integer, so we cannot achieve precisely this rate, but we can get reasonably close (at least if $m \gg n$).

Finally, the previous analysis implies that we can achieve any desired false positive rate $\delta > 0$ using a Bloom filter of size

$$m = \left\lceil \frac{\lg(1/\delta)}{\ln 2} n \right\rceil = \Theta(n \log(1/\delta))$$

that uses with $k = \lceil \lg(1/\delta) \rceil$ hash functions. For example, we can achieve a 1% false-positive rate using a Bloom filter of size $10n$ bits with 7 hash functions; in practice, this is *considerably* fewer bits than we would need to store all the elements of $S$ explicitly. With a $32n$-bit table (equivalent to one integer per item) and 22 hash functions, we get a false positive rate of just over $2 \cdot 10^{-7}$.

★★★ Deletions via counting filters? Key-value pairs via Bloomier filters? Other extensions?

## 6.3 Streaming Algorithms

A ***data stream*** is an extremely long sequence $S$ of items from some universe $\mathcal{U}$ that can be read only once, in order. Good examples of data streams include the sequence of packets that pass through a network router, the sequence of searches at google.com, the sequence of all bids on the New York Stock Exchange, and the sequence of humans passing through the Shinjuku Railway Station in Tokyo. Standard algorithms are not appropriate for data streams; there is simply too much data to store, and it arrives too quickly for any complex computations.

A ***streaming algorithm*** processes each item in a data stream stream as it arrives, maintaining some summary information in a local data structure. The basic structure of every streaming algorithm is the following:

```
DoSomething(S):
  ⟨⟨initialize⟩⟩
  while S is not done
      x ← next item in S
      ⟨⟨do something fast with x⟩⟩
  return ⟨⟨something⟩⟩
```

Ideally, neither the running time per item nor the space used by the data structure depends on the overall length of the stream; viewed as algorithms in the traditional sense, all streaming algorithms run in constant time! Somewhat surprisingly, even within this very restricted model, it is possible to compute interesting properties of the stream using randomization, provided we are willing to tolerate some errors in the output.

## 6.4 The Count-Min Sketch

As an example, consider the following problem: At any point during the stream, we want to estimate the number of times that an arbitrary item $x \in \mathcal{U}$ has appeared in the stream so far. This problem can be solved with a variant of Bloom filters called the ***count-min sketch***, first published by Graham Cormode and S. Muthu Muthukrishnan in 2005.

The count-min sketch consists of a $w \times d$ array of counters (all initially zero) and $d$ hash functions $h_1, h_2, \ldots, h_d : \mathcal{U} \to [m]$, drawn independently and uniformly at random from a 2-uniform family of hash functions. Each time an item $x$ arrives in the stream, we call CMIncrement($x$). Whenever we want to estimate the number of occurrences of an item $x$ so far, we call CMEstimate($x$).

```
$\text{CMIncrement}(x)$:
    for $i \gets 1$ to $d$
        $j \gets h_i(x)$
        $Count[i,j] \gets Count[i,j] + 1$
```

```
$\text{CMEstimate}(x)$:
    $est \gets \infty$
    for $i \gets 1$ to $d$
        $j \gets h_i(x)$
        $est \gets \min\{est, Count[i,j]\}$
    return $est$
```

If we set $w := \lceil e/\varepsilon \rceil$ and $d := \lceil \ln(1/\delta) \rceil$, then the data structure uses $O(\frac{1}{\varepsilon}\log\frac{1}{\delta})$ space and processes updates and queries in $O(\log\frac{1}{\delta})$ time.

Let $f_x$ be the true frequency (number of occurrences) of $x$, and let $\hat{f}_x$ be the value returned by CMEstimate. It is easy to see that $f_x \le \hat{f}_x$; we can never return a value smaller than the actual number of occurrences of $x$. We claim that $\mathbf{Pr}[\hat{f}_x > f_x + \varepsilon N] < \delta$, where $N$ is the total length of the stream. In other words, our estimate is never too small, and with high probability, it isn't a significant overestimate either. (Notice that the error here is additive; the estimates or truly infrequent items may be much larger than their true frequencies.)

For any items $x \ne y$ and any index $j$, we define an indicator variable $X_{i,x,y} = [h_i(x) = h_i(y)]$; because the hash functions $h_i$ are universal, we have

$$\mathrm{E}[X_{i,x,y}] = \Pr[h_i(x) = h_i(y)] = \frac{1}{w}.$$

Let $X_{i,x} := \sum_{y \ne x} X_{i,x,y} \cdot f_y$ denote the total number of collisions with $x$ in row $i$ of the table. Then we immediately have

$$Count[i, h_i(x)] \;=\; f_x + X_{i,x} \;\ge\; f_x.$$

On the other hand, linearity of expectation implies

$$\mathrm{E}[X_{i,x}] \;=\; \sum_{y \ne x} \mathrm{E}[X_{i,x,y}] \cdot f_y \;=\; \frac{1}{w}\sum_{y \ne x} f_y \;\le\; \frac{N}{w}.$$

Now Markov's inequality implies

$$\begin{aligned}
\Pr[\hat{f}_x > f_x + \varepsilon N] &= \Pr[X_{i,x} > \varepsilon N \text{ for all i}] && \text{[definition]}\\
&= \Pr[X_{1,x} > \varepsilon N]^d && \text{[independence of } h_i\text{'s]}\\
&\le \left(\frac{\mathrm{E}[X_{1,x}]}{\varepsilon N}\right)^d && \text{[Markov's inequality]}\\
&\le \left(\frac{N/w}{\varepsilon N}\right)^d = \left(\frac{1}{w\varepsilon}\right)^d && \text{[derived earlier]}
\end{aligned}$$

Now setting $w = \lceil e/\varepsilon \rceil$ and $d = \lceil \ln(1/\delta) \rceil$ gives us $\Pr[\hat{a}_x > a_x + \varepsilon N] \le (1/e)^{\ln(1/\delta)} = \delta$, as claimed.

## 6.5 Estimating Distinct Items

★★★

Write this, possibly as a simpler replacement for the count-min sketch. AMS estimator: $2^{z+1/2}$ where $z = \max\{zeros(x) \mid x \in S\}$? Or stick with the Flajolet-Martin/Bar-Yossef-et-al estimator in the exercises? Median amplification.

Estimating larger moments?

## Exercises

1. ***Reservoir sampling*** is a method for choosing an item uniformly at random from an arbitrarily long stream of data; for example, the sequence of packets that pass through a router, or the sequence of IP addresses that access a given web page. Like all data stream algorithms, this algorithm must process each item in the stream quickly, using very little memory.

```
GETONESAMPLE(stream $S$):
  $\ell \leftarrow 0$
  while $S$ is not done
      $x \leftarrow$ next item in $S$
      $\ell \leftarrow \ell + 1$
      if RANDOM($\ell$) $= 1$
          sample ← x        (⋆)
  return sample
```

At the end of the algorithm, the variable $\ell$ stores the length of the input stream $S$; this number is *not* known to the algorithm in advance. If $S$ is empty, the output of the algorithm is (correctly!) undefined. In the following, consider an arbitrary non-empty input stream $S$, and let $n$ denote the (unknown) length of $S$.

(a) Prove that the item returned by GETONESAMPLE($S$) is chosen uniformly at random from $S$.

(b) What is the *exact* expected number of times that GETONESAMPLE($S$) executes line ($\star$)?

(c) What is the *exact* expected value of $\ell$ when GETONESAMPLE($S$) executes line ($\star$) for the *last* time?

(d) What is the *exact* expected value of $\ell$ when either GETONESAMPLE($S$) executes line ($\star$) for the *second* time (or the algorithm ends, whichever happens first)?

(e) Describe and analyze an algorithm that returns a subset of $k$ distinct items chosen uniformly at random from a data stream of length at least $k$. The integer $k$ is given as part of the input to your algorithm. Prove that your algorithm is correct.

For example, if $k = 2$ and the stream contains the sequence ⟨♠, ♥, ♦, ♣⟩, the algorithm should return the subset {♦, ♠} with probability 1/6.

2. In this problem, we will derive a simple streaming algorithm (first published by Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan in 2002) to estimate the number of distinct items in a data stream $S$.

Suppose $S$ contains $n$ unique items (but possibly several copies of each item); as usual, the algorithm does *not* know $n$ in advance. Given an accuracy parameter $0 < \varepsilon < 1$ and a confidence parameter $0 < \delta < 1$ as part of the input, our final algorithm will compute an estimate $\widetilde{N}$ such that $\Pr[|\widetilde{N} - n| > \varepsilon n] < \delta$.

As a first step, fix a positive integer $m$ that is large enough that we don't have to worry about round-off errors in the analysis. Our first algorithm chooses a hash function $h\colon \mathcal{U} \to [m]$ at random from a ***2-uniform*** family, computes the minimum hash value $\hbar = \min\{h(x) \mid x \in S\}$, and finally returns the estimate $\tilde{n} = m/\hbar$.

(a) Prove that $\Pr[\widetilde{n} > (1+\varepsilon)n] \le 1/(1+\varepsilon)$. *[Hint: Markov's inequality]*

(b) Prove that $\Pr[\widetilde{n} < (1-\varepsilon)n] \le 1-\varepsilon$. *[Hint: Chebyshev's inequality]*

(c) We can improve this estimator by maintaining the $k$ smallest hash values, for some integer $k > 1$. Let $\widetilde{n}_k = k \cdot m/\hbar_k$, where $\hbar_k$ is the $k$th smallest element of $\{h(x) \mid x \in S\}$.

Estimate the smallest value of $k$ (as a function of the accuracy parameter $\varepsilon$) such that $\Pr[|\widetilde{n}_k - n| > \varepsilon n] \le 1/4$.

(d) Now suppose we run $d$ copies of the previous estimator in parallel to generate $d$ independent estimates $\widetilde{n}_{k,1}, \widetilde{n}_{k,2}, \ldots, \widetilde{n}_{k,d}$, for some integer $d > 1$. Each copy uses its own independently chosen hash function, but they all use the same value of $k$ that you derived in part (c). Let $\widetilde{N}$ be the *median* of these $d$ estimates.

Estimate the smallest value of $d$ (as a function of the confidence parameter $\delta$) such that $\Pr[|\widetilde{N} - n| > \varepsilon n] \le \delta$.