Bloom Filters: Randomness, Probability, and the Mathematics of Approximate Membership

Tom Rocks Maths Competition

April 2026

Abstract. *A Bloom filter is a space-efficient probabilistic data structure that answers the question "is this element in the set?" not with a guaranteed* yes *or* no*, but with either a* definite no *or a* probably yes*. This essay develops the full mathematical theory from scratch: what a hash function is, how the Bloom filter is constructed, a rigorous derivation of the false positive probability, the optimisation of the number of hash functions, and — most beautifully — an information-theoretic argument showing that the Bloom filter uses only* $\ln 2 \approx 1.44$ *times the absolute minimum number of bits that any data structure could ever require. The same constant* $\ln 2$ *that emerges from the symmetry condition* $p = \frac{1}{2}$ *reappears in the information-theoretic bound, revealing a deep coherence at the heart of the design.*

Introduction: The Membership Problem

Imagine you are a web browser trying to protect users from malicious websites. There are roughly 500 million known dangerous URLs. Every time a user clicks a link, you need to check whether that URL is on the list — ideally in milliseconds. Storing all 500 million URLs in a hash set is possible, but it costs gigabytes of RAM. Can we do better?

This is the *membership problem*: given a set S of n items drawn from a universe $\mathcal{U}$, build a data structure that answers queries of the form "is $x \in S$?" as quickly and compactly as possible.

The Bloom filter, introduced by Burton Howard Bloom in 1970 [1], offers a striking answer: yes, dramatically so — at the cost of occasionally answering "probably yes" when the true answer is "no." It *never* produces false negatives (it will never claim an item is absent when it is present), but it permits false positives with a tunable, small probability ε. For the malware-detection use case, this is perfectly acceptable: the browser can send the small fraction of "probably dangerous" URLs for a full server-side check.

What makes Bloom filters not just practically useful but mathematically remarkable is the reason they are so efficient. As we shall see, an optimally configured Bloom filter uses only 1.44 times the minimum number of bits that any correct data structure — however ingeniously designed — could possibly use. This is not a heuristic claim; it follows from information theory. The same logarithm that appears in the optimal design appears in the theoretical lower bound, and the ratio is exactly $1/\ln 2$.

Hash Functions

A **hash function** takes an input — any word, number, URL, or arbitrary string — and outputs an integer in some fixed range. Formally, $h : \mathcal{U} \to \{0, 1, \ldots, m-1\}$ for some m.

Two key properties make hash functions useful:

- **Determinism**: the same input always yields the same output.
- **Speed**: the output is computed in constant time, $O(1)$, regardless of the size of the input.

A simple example and its flaw

A simple hash function adds up the ASCII values of all characters in a string, then takes the result modulo m. For instance, with $m = 10$:

$$h(\text{“cat”}) = 99 + 97 + 116 = 312 \equiv 2 \pmod{10},$$

$$h(\text{“dog”}) = 100 + 111 + 103 = 314 \equiv 4 \pmod{10}.$$

The immediate problem is that the output depends only on the *multiset* of characters, not their order. Thus $h(\text{“act”}) = 99 + 99 + 116 = 312 \equiv 2$ as well. The words "cat" and "act" collide: they map to the same slot. This creates **crowding** — multiple distinct inputs sharing the same output — and leads to false positives in any membership test based

solely on this hash. The Bloom filter's ingenious solution is to use *multiple independent* hash functions, making it exponentially harder to get a false positive by chance.

The Bloom Filter: Construction

A Bloom filter for a set of n items consists of:

- An array B of m bits, all initialised to 0.
- k independent hash functions $h_1, h_2, \ldots, h_k$, each mapping any input uniformly to one of the m positions $\{0, 1, \ldots, m-1\}$.

Insertion

To insert item x, compute $h_1(x), h_2(x), \ldots, h_k(x)$ and set each of those k bit positions to 1:

$$B[h_i(x)] \leftarrow 1 \quad \text{for } i = 1, 2, \ldots, k.$$

Query

To query whether $x \in S$, compute the same k positions. If *all* of them are 1, return "probably yes". If *any* is 0, return "definitely not".

There are **no false negatives**: if x was inserted, all k of its bits were set to 1 and none can have been reset (the filter never resets bits). However, there can be **false positives**: a new item $y \notin S$ might happen to hash to k positions that were all set to 1 by other inserted items.

A worked example

Take $m = 10$ bits and $k = 3$ hash functions. Insert two items:

$$\text{"alice"}: \quad h_1 = 1,\ h_2 = 4,\ h_3 = 7.$$

$$\text{"bob"}: \quad h_1 = 3,\ h_2 = 4,\ h_3 = 9.$$

The bit array after both insertions:

Index	0	1	2	3	4	5	6	7	8	9
Bit	0	1	0	1	1	0	0	1	0	1

Now query "carol", whose hash values are $h_1 = 0$, $h_2 = 2$, $h_3 = 3$. Position 0 is 0, so the filter correctly returns "definitely not in set."

But suppose "dave" hashes to $1, 3, 4$. All three bits are 1 — yet "dave" was never inserted. This is a **false positive**: the filter says "probably yes" when the truth is "no." The art of Bloom filter design is controlling exactly how likely this is.

The False Positive Probability

We now derive the probability of a false positive rigorously.

After inserting n items

Each hash function, applied to any input, picks one of m positions uniformly at random. After inserting n items using k hash functions, there have been kn total hash operations, each independently setting one bit to 1.

The probability that a single specific bit is *not* set by any one hash operation is:

$$1 - \frac{1}{m}.$$

After all kn operations, the probability that a specific bit is *still* 0 is:

$$P(\text{bit} = 0) = \left(1 - \frac{1}{m}\right)^{kn}.$$

For large m, we use the standard approximation $\left(1 - \frac{1}{m}\right)^m \approx e^{-1}$, so:

$$P(\text{bit} = 0) = \left[\left(1 - \frac{1}{m}\right)^m\right]^{kn/m} \approx e^{-kn/m}.$$

Therefore, the probability that a bit is 1 is:

$$P(\text{bit} = 1) = 1 - e^{-kn/m}.$$

False positive probability

A false positive occurs when a queried item $y \notin S$ has all k of its hash positions equal to 1. Since each position is independently 1 with probability $(1 - e^{-kn/m})$:

$$\boxed{P(\text{false positive}) = \left(1 - e^{-kn/m}\right)^k.}$$

Let us introduce the shorthand $p = e^{-kn/m}$, the probability that a specific bit remains 0. Then:

$$P(\text{false positive}) = (1 - p)^k.$$

Optimising the Number of Hash Functions

Given fixed m and n, what value of k minimises the false positive probability? This is where the mathematics becomes genuinely beautiful.

We want to minimise $f(k) = (1 - p)^k$, but p itself depends on k through $p = e^{-kn/m}$. It is more convenient to minimise $\ln f(k) = k \ln(1 - p)$. Taking the natural logarithm of p:

$$\ln p = -\frac{kn}{m} \implies k = -\frac{m}{n} \ln p.$$

Substituting into $\ln f$:

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

For fixed m and n, we minimise $g(p) = -(\ln p)(\ln(1-p))$ over $p \in (0,1)$.

Setting $g'(p) = 0$:

$$g'(p) = -\frac{\ln(1-p)}{p} + \frac{\ln p}{1-p} = 0.$$

This gives $(1-p)\ln(1-p) = p \ln p$. Let $f(t) = t \ln t$; we need $f(1-p) = f(p)$. Since $f(t) = t \ln t$ takes the same value at two points a and b in $(0,1)$ only when $a = b$ (as f is strictly convex on $(0,1)$ with a unique minimum), we conclude:

$$1 - p = p \implies \boxed{p = \tfrac{1}{2}.}$$

This is the **symmetry condition**: the false positive probability is minimised when exactly half the bits in the array are set to 1. From $p = e^{-kn/m} = \frac{1}{2}$:

$$\frac{kn}{m} = \ln 2 \implies \boxed{k = \frac{m}{n} \ln 2.}$$

The minimum false positive probability at this optimal k is:

$$P(\text{false positive})_{\min} = (1-p)^k = \left(\frac{1}{2}\right)^k = 2^{-k} = 2^{-(m/n)\ln 2} = 2^{-m \ln 2/n}.$$

We can also write this compactly as:

$$P(\text{false positive})_{\min} = \left(\frac{1}{2}\right)^{(m/n)\ln 2} \approx (0.6185)^{m/n}.$$

Two competing effects

There is a beautiful tension in increasing k:

1. **Effect 1 — more checks, harder to fool.** With more hash functions, a false positive requires *more* positions to all be accidentally 1. Each additional hash function pushes the false positive rate down.
2. **Effect 2 — more insertions, more bits set.** Each new hash function sets one more bit per inserted item. The filter fills up faster, making it more likely any given position is 1, which pushes the false positive rate back up.

The optimal $k = (m/n)\ln 2$ is exactly where these two effects are perfectly balanced — the point of symmetry $p = \frac{1}{2}$.

Numerical illustration

For $n = 1{,}000{,}000$ items and $m \approx 7.4$ MB of memory:

k	False positive rate	Note
1	$\sim 10\%$	
3	$\sim 2.1\%$	
7	$\sim 0.82\%$	$\leftarrow$ optimal
10	$\sim 1.1\%$	rate rises again
20	$\sim 4.1\%$	far from optimal

Notice how the false positive rate rises again after the optimal $k = 7$. This confirms the two competing effects: past the optimum, the array fills up too fast and the filter becomes less discriminating.

Memory and the Information-Theoretic Argument

So far we have treated m as given. But how large should m be for a target false positive rate ε? And could some other data structure do better?

Memory required by an optimal Bloom filter

Setting $P(\text{false positive}) = \varepsilon$ with the optimal k:

$$\varepsilon = 2^{-k} = 2^{-(m/n)\ln 2}.$$

Taking $\log_2$:

$$\log_2 \varepsilon = -\frac{m}{n}\ln 2 \implies m = -\frac{n \log_2 \varepsilon}{\ln 2} = \frac{n \log_2(1/\varepsilon)}{\ln 2}.$$

So the Bloom filter uses:

$$\boxed{m_{\text{Bloom}} = \frac{n \log_2(1/\varepsilon)}{\ln 2} \approx 1.44\, n \log_2 \frac{1}{\varepsilon} \text{ bits.}}$$

The information-theoretic lower bound

Now we ask: is this optimal? Could a cleverer structure do better?

Suppose the universe has U possible items and we want to store a subset S of exactly n items. Any data structure that can answer membership queries must be able to *distinguish* between different possible sets S, because two different sets would otherwise be indistinguishable and would produce wrong answers for some queries.

The number of distinct n-element subsets of a universe of size U is $\binom{U}{n}$. To distinguish them all, the data structure must have at least:

$$\log_2 \binom{U}{n} \text{ bits of state.}$$

By Stirling's approximation and the entropy formula, for $n \ll U$:

$$\log_2 \binom{U}{n} \approx n \log_2 \frac{U}{n}.$$

When we allow a false positive rate of ε, some pairs of sets can be "confused" because they differ only on elements that will be false positives anyway. A careful analysis (due to Carter, Floyd, Gill, Markowsky, and Wegman [2]) shows that the minimum number of bits required by any data structure — not just Bloom filters — that achieves a false positive rate of at most ε is:

$$m_{\min} = n \log_2 \frac{1}{\varepsilon} \text{ bits.}$$

This is Shannon's floor. No matter how clever your design, no matter how exotic your approach, you cannot do better.

The Bloom filter is nearly optimal

Comparing the two:

	Bits used
Theoretical minimum (any data structure)	$n \log_2(1/\varepsilon)$
Optimal Bloom filter	$\dfrac{n \log_2(1/\varepsilon)}{\ln 2} = \dfrac{1}{\ln 2} \cdot n \log_2(1/\varepsilon)$
Ratio	$\dfrac{1}{\ln 2} \approx 1.4427$

The Bloom filter uses exactly $\frac{1}{\ln 2} \approx 1.44$ times the information-theoretic minimum. In concrete terms:

- For $n = 10^6$ items at $\varepsilon = 1\%$: theoretical minimum ≈ 8.3 MB; Bloom filter ≈ 11.9 MB; a hash set storing actual items: > 50 MB.

The Deepest Observation: Why $\ln 2$ Appears Twice

Here is the mathematical punchline. We have now seen $\ln 2$ in two entirely different roles:

1. In the **optimal** k: the false positive probability is minimised when $p = \frac{1}{2}$, i.e. when half the bits are 1. This led to $k_{\text{opt}} = (m/n) \ln 2$, with $\ln 2$ appearing from $\log_e 2 = \ln 2$.
2. In the **overhead ratio**: the Bloom filter uses $1/\ln 2$ times as many bits as the theoretical minimum — the same constant, now appearing as the reciprocal.

These are not a coincidence. Both stem from the same symmetry: the optimal operating point of the filter is when each bit is independently and uniformly 1 with probability exactly $\frac{1}{2}$. At this point, the filter is extracting the most information per bit — its bits are maximally entropic, each carrying exactly 1 bit of Shannon information. That is what connects the optimisation argument ("minimise false positive rate") to the information-theoretic argument ("minimise bits used"), and it is why the same constant governs both.

The 44% overhead is not a flaw in the design — it is the irreducible cost of using a hash-based structure rather than an arbitrary mapping. You could spend years trying to close this gap and you would fail, because it is not an engineering gap but a mathematical one.

Using the Filter: Bloom's Original Insight

Returning to where we started: the Bloom filter says, for any query:

- **Definitely not in the set**: $P(\text{existence}) = 0$.
- **Probably in the set**: $P(\text{existence}) > 0$.

By increasing the number of hash functions, we can reduce the false positive rate further. For a large number of distinct inputs, adding more hash functions drives the error toward zero — or more precisely, "probably exists" becomes "almost certainly exists."

This is the magic of the design. The actual stored values are discarded completely. All that remains is m bits and k hash functions — no strings, no URLs, no keys. Yet the answer to "is this item in the set?" is almost always correct, in a provably optimal way.

Conclusion

The Bloom filter is a rare example of a data structure that is simultaneously simple to describe, useful in practice, and provably close to a mathematical ideal. The key steps of the argument are:

1. The false positive probability is $f(k) = (1 - e^{-kn/m})^k$.
2. Minimising over k yields the symmetry condition $p = e^{-kn/m} = \frac{1}{2}$, giving the optimal $k = (m/n) \ln 2$.
3. The minimum false positive probability at this optimal k is $(1/2)^{(m/n)\ln 2}$.
4. Setting this equal to ε and solving for m gives $m = n \log_2(1/\varepsilon) / \ln 2 \approx 1.44 \cdot n \log_2(1/\varepsilon)$.
5. The information-theoretic lower bound for any data structure with false positive rate ε is $n \log_2(1/\varepsilon)$ bits.
6. Therefore the Bloom filter is exactly $1/\ln 2$ times the theoretical optimum — and the same $\ln 2$ that appears in the optimal k is also what governs the gap to the lower bound.

This is what elevates Bloom filters from a clever trick to genuine mathematics: not just that they work well in practice, but that they are provably, permanently, within 44% of the best possible answer. That closing 44% is not an engineering challenge to be overcome — it is the precise cost, dictated by the laws of information theory, of building a membership structure this way.

References

[1] B. H. Bloom, *Space/time trade-offs in hash coding with allowable errors*, Communications of the ACM, 13(7):422–426, 1970.

[2] J. L. Carter, R. Floyd, J. Gill, G. Markowsky, and M. N. Wegman, *Exact and approximate membership testers*, Proceedings of the 10th Annual ACM Symposium on Theory of Computing, 1978.

[3] M. Mitzenmacher, *Compressed Bloom filters*, IEEE/ACM Transactions on Networking, 10(5):604–612, 2002.