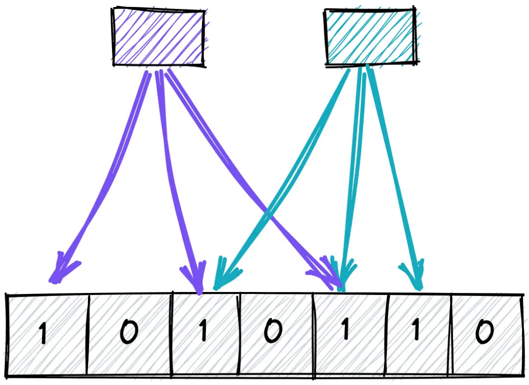
INFO—F413
Randomized Algorithms — Bloom Filters
Project Report

Berthion Antoine — 566199

3 November 2025



Abstract

This project presents the implementation of a *hash factory* and a *Bloom filter*. We provide both **theoretical** and **experimental** analyses of their performance, establishing formal *bounds* and *guarantees* on efficiency. Additionally, we discuss **design choices**, **implementation details**, and the *trade-offs* encountered during development.

1 Introduction

In numerous computational contexts, determining whether an element belongs to a given set is fundamental. Efficiently solving this *membership problem* is critical in applications ranging from large-scale databases to network systems. A standard approach relies on **hash tables**, which support constant-time ($O(1)$) membership queries by storing hashed representations of elements. However, when dealing with very large datasets, even storing all hash values can become memory-intensive.

To address these limitations, **randomized algorithms** offer a practical alternative. These methods accept a small probability of error in exchange for reduced space or faster execution, striking a balance between precision and efficiency.

Bloom filters exemplify this approach. They answer membership queries with either *definitely not present* or *possibly present*. This behavior is particularly useful in large-scale systems: a Bloom filter can pre-check whether an element exists in a database. If the filter indicates absence, the costly lookup can be skipped; if it indicates presence, the query proceeds, with a small chance of a false positive.

2 Formalisation

We now formalize the membership problem, the probabilistic framework, and the structure of Bloom filters.

2.1 Membership Problem

Let U denote a finite universe of possible elements, and let $S \subseteq U$ be the set to store. The goal is to design a data structure supporting the query operation:

$$\texttt{query}(x) = \begin{cases} \texttt{true}, & \text{if } x \in S, \\ \texttt{false}, & \text{if } x \notin S. \end{cases} \tag{1}$$

Deterministic structures such as hash tables provide exact answers in $O(1)$ expected time but require memory proportional to $|S|$. When $|S|$ is very large, storing all elements becomes impractical, motivating probabilistic alternatives.

2.2 Probabilistic Relaxation

A **randomized algorithm** introduces controlled uncertainty to improve efficiency. In this setting, we allow a small probability ε of returning a *false positive*—indicating that an element is present when it is not—but no *false negatives*. Formally, for a randomized structure B:

$$\Pr[B.\texttt{query}(x) = \texttt{true} \mid x \notin S] = \varepsilon, \quad \Pr[B.\texttt{query}(x) = \texttt{false} \mid x \in S] = 0. \tag{2}$$

The parameter ε quantifies the trade-off between memory usage and accuracy.

2.3 Bloom Filter Model

A **Bloom filter** stores S using a bit array A of size m and k independent hash functions

$$h_i : U \to \{0, \ldots, m-1\}, \quad 1 \le i \le k.$$

For each element $x \in S$, the filter sets all bits $A[h_i(x)] \leftarrow 1$. A membership query checks whether all corresponding bits are set:

$$B.\texttt{query}(x) = \begin{cases} \texttt{true}, & \text{if } A[h_i(x)] = 1 \text{ for all } i, \\ \texttt{false}, & \text{otherwise.} \end{cases} \tag{3}$$

If at least one bit is zero, the element is definitely absent; if all bits are one, it is *probably present*, with error probability bounded by ε.

2.4 Hash Function Family

Hash functions play a central role in Bloom filters. In this project, we use the **Carter–Wegman universal hash family**, defined as:

$$h_{a,b}(x) = ((a \cdot x + b) \bmod p) \bmod m, \tag{4}$$

where $a \in \{1, \ldots, p-1\}$ and $b \in \{0, \ldots, p-1\}$ are chosen randomly, and p is a large prime. This family is **pairwise universal**:

$$\Pr_{a,b}[h_{a,b}(x) = h_{a,b}(y)] \le \frac{1}{m}, \quad x \ne y, \tag{5}$$

ensuring low collision probability and an approximately uniform distribution of hash values.

3 Implementation

This section describes the key design choices in our implementation.

3.1 Hash Function Generation

To obtain sufficiently independent hash functions, the coefficients a and b in Equation 4 are generated pseudo-randomly. Each pair (a, b) is drawn uniformly from

$$a \in \{1, \dots, p-1\}, \qquad b \in \{0, \dots, p-1\},$$

with p a large prime.

To facilitate efficient modular arithmetic while avoiding overflow, we fix a **Mersenne prime** $p = 2^{61} - 1$, which is prime and well-suited for 64-bit computations.[1] This choice balances numerical stability and performance.

3.2 Bloom Filter

The Bloom filter is implemented as a **bit array**, providing compact storage and constant-time access. The structure is parameterized by:

- m — the total number of bits in the filter;
- k — the number of hash functions h_i from Section 2.4.

Each insertion sets bits at positions $h_i(x)$, and membership queries check whether all relevant bits are set, as formalized in Section 2.

4 Bounds and Optimality

Before proceeding to experimentation, we recall the theoretical bounds that govern Bloom filter performance. These results provide reference points for the empirical behaviour observed later.

4.1 False Positive Probability

The false positive probability of a Bloom filter with m bits, k independent hash functions, and n inserted elements is given by:

$$P_{\text{fp}} = \left(1 - e^{-kn/m}\right)^k \tag{6}$$

as first derived by Bloom [1] and further analysed in later studies [2]. This expression shows how the false positive rate depends exponentially on the load factor n/m and the number of hash functions k.

4.2 Optimal Parameters

The number of hash functions can be tuned to minimize P_{fp}. The optimal choice, obtained analytically, is:

$$k_{\text{opt}} = \frac{m}{n} \ln 2 \tag{7}$$

which corresponds to a situation where approximately half of the bits in the array are set to one. At this point, the minimal achievable false positive rate becomes:

$$P_{\text{fp,min}} \approx (0.6185)^{m/n} \tag{8}$$

These results express the fundamental space–accuracy trade-off of Bloom filters and define the theoretical limits our implementation can aim to approach.

[1] A *Mersenne prime* is a prime of the form $2^k - 1$, convenient for modular arithmetic due to its binary structure.

5 Experiments

This section outlines the objectives, methodology, and results of our experimental analysis. We aim to empirically validate the theoretical properties of Bloom filters, particularly their false-positive behaviour and the influence of key parameters such as the number of hash functions and the filter's load factor.

5.1 Methodology

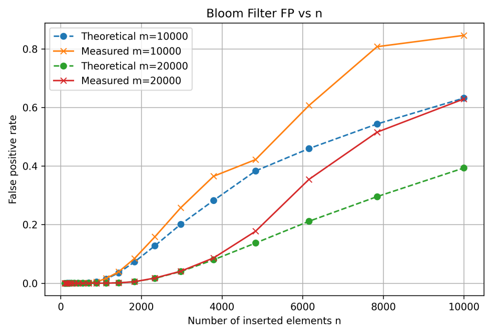
To assess the performance and limitations of our implementation, we conducted two main experiments:

1. **Growth analysis:** We measured the false-positive rate as the Bloom filter becomes increasingly filled, by gradually inserting n elements and testing for membership of elements not in the set.
2. **Parameter analysis:** We examined how varying the number of hash functions k affects the false-positive probability, keeping the Bloom filter size m and number of inserted elements n fixed.

These experiments allow us to compare empirical results against the theoretical false-positive probability and to evaluate how closely our implementation aligns with this model.

5.2 Results

The results of our experiments are shown in Figure 1.



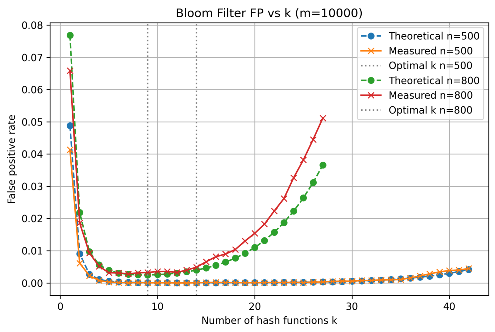


Figure 1: Experimental results: false-positive rate vs number of elements n (left) and vs number of hash functions k (right).

In the **growth analysis**, the false-positive rate increases steadily as the number of inserted elements n grows, matching theoretical expectations. For small n, the false-positive probability is negligible; as the load factor n/m increases, collisions in the bit array become more frequent, causing the rate to rise. Empirical measurements follow the theoretical curve closely, confirming the correctness of the implementation.

In the **parameter analysis**, varying the number of hash functions k produces the expected U-shaped behaviour. Too few hash functions leave many bits unset, increasing false positives, while too many hash functions oversaturate the bit array, also raising the rate. The minimum occurs near the theoretical optimal $k_{\text{opt}} = \frac{m}{n} \ln 2$ (see Equation 7), which our measurements confirm. Slight deviations at high load factors are likely due to finite-size effects and limited randomness.

Overall, the experiments support the theoretical model and demonstrate that the Bloom filter behaves predictably across different parameters.

6 Use of LLMs

In this brief section, we discuss the use of large language models (LLMs) in the context of this project. No LLM was used for the implementation or for understanding the project itself. However, this report was reviewed for syntax and grammar by DeepL as well as a GPT model. It should be noted that none of the information contained in this report was generated by anyone other than the author.

7 Conclusion

This project presented the design, implementation, and evaluation of a Bloom filter using a universal hash family. Both theoretical analysis and experimental results confirm that the filter behaves as expected: the false-positive rate increases with load and exhibits the characteristic U-shaped dependence on the number of hash functions. Optimal parameters closely match theoretical predictions, validating the correctness and efficiency of our implementation.

Overall, Bloom filters provide a compact and effective probabilistic solution to the membership problem, illustrating the power of randomized algorithms in practical applications.

Appendix A: Hash Factory Implementation

Python implementation of universal hash functions and the factory to generate them.

```python

import random

class UniversalHash:
    def __init__(self, p: int, a: int, b: int, m: int):
        self._p = p
        self._a = a
        self._b = b
        self._m = m

    def __call__(self, value: int) -> int:
        if value < 0:
            raise ValueError(f"Value must be non-negative (got {value}).")
        return ((self._a * value + self._b) % self._p) % self._m


class HashFactory:
    @staticmethod
    def create(m: int, seed: int = None) -> UniversalHash:
        rnd = random.Random(seed)
        p = (1 << 61) - 1
        a = rnd.randint(1, p - 1)
        b = rnd.randint(0, p - 1)
        return UniversalHash(p, a, b, m)
```

Appendix B: Bloom Filter Implementation

Python implementation of the Bloom filter using universal hash functions.

```python

from bitarray import bitarray
from . import UniversalHash, HashFactory

class BloomFilter:
    def __init__(self, m, hash_functions):
        self._m = m
        self._hf = hash_functions
        self._bit_array = bitarray([0] * m)

    @classmethod
    def New(cls, m, k, seed=None):
        hash_functions = [
            HashFactory.create(m, seed + i if seed is not None else None)
            for i in range(k)
        ]
        return cls(m, hash_functions)

    def add(self, x):
        for index in (hf(x) % self._m for hf in self._hf):
            self._bit_array[index] = 1

    def query(self, x):
        return all(self._bit_array[hf(x) % self._m] for hf in self._hf)
```

References

[1] B. H. Bloom. *Space/Time Trade-offs in Hash Coding with Allowable Errors.* Communications of the ACM, vol. 13, no. 7, pp. 422–426, 1970.

[2] A. Z. Broder and M. Mitzenmacher. *Network Applications of Bloom Filters: A Survey.* Internet Mathematics, vol. 1, no. 4, pp. 485–509, 2004.