COSC 520 Assignment 1 - The Login Checker Problem

ZHENBANG HE, Uniersity of British Columbia, Canada

Abstract

Additional Key Words and Phrases: Algorithm analysis, Algorithm design, Computational theory

ACM Reference Format:
Zhenbang He. 2025. COSC 520 Assignment 1 - The Login Checker Problem. 1, 1 (October 2025), 8 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Efficient data retrieval and username queries are critical in computer science and data-intensive applications, like online shopping websites. This report provides a concise but detailed overview of the time and space complexities of commonly used data structures and algorithms: linear search, binary search, hashing, Bloom filters, and Cuckoo filters. We describe the key principles, parameters, and justify the complexity results while highlighting the trade-offs.

We also implemented each algorithm in Python. In our experiments, we generated datasets with different numbers of usernames and tested the performance of each method in terms of both time consumption and space (memory) consumption. Figures were plotted to show the average/total running time, memory consumption, and correctness of each algorithm. We found that Bloom filters and Cuckoo filters exhibit significantly lower memory consumption while maintaining similar query speeds. Their correctness was also close to that of the deterministic search methods.

2 Linear Search

2.1 Principles and Brief Introduction

Linear search works by checking each element in the dataset sequentially until the desired element is found or the end of the dataset is reached. It does not require preprocessing or sorting.

2.2 Complexity Derivation

Time Complexity: Each lookup may require examining up to n elements in the worst case. Therefore, time complexity is $O(n)$. In the best case, only one comparison is required, giving $O(1)$. The average case assumes the item is equally likely to appear anywhere, resulting in $O(n/2) = O(n)$.

Space Complexity: Only a constant number of additional variables are required (indices and comparisons), so the space complexity is $O(1)$.

Author's Contact Information: Zhenbang He, zbhe96@student.ubc.ca, Uniersity of British Columbia, Kelowna, British Columbia, Canada.

3 Binary Search

3.1 Principles and Brief Introduction

Binary search applies a divide-and-conquer strategy on sorted arrays. At each step, the search space is halved.

3.2 Complexity Derivation

Time Complexity: Each comparison halves the problem size from n to $n/2$, then $n/4$, and so on. After k steps, the size becomes $n/2^k$. The process ends when $n/2^k = 1$, giving $k = \log_2 n$. Hence, time complexity is $O(\log n)$. The best case is when the middle element is the target ($O(1)$).

Space Complexity: Iterative binary search uses only a few variables for indices ($O(1)$). Recursive binary search requires additional stack frames for each recursive call, giving $O(\log n)$.

4 Hashing

4.1 Principles and Brief Introduction

Hashing maps keys to slots in a table using a hash function. Ideally, this distributes keys uniformly across m slots.

4.2 Complexity Derivation

Time Complexity: Under uniform distribution and low load factor ($\alpha = n/m$), expected time for lookup, insertion, and deletion is $O(1)$. In the worst case, all n keys map to the same slot, requiring linear traversal, so the worst case is $O(n)$.

Space Complexity: A hash table requires m slots for storage plus space for n elements. Thus, space complexity is $O(m + n)$.

5 Bloom Filter

5.1 Principles and Brief Introduction

A Bloom filter uses a bit array of size m and k hash functions. Each element sets k bits in the array. Queries check these positions.

5.2 Complexity Derivation

Time Complexity: Insertion and query each require k independent hash computations and array accesses, giving $O(k)$. Since k is usually constant relative to n, this is often considered $O(1)$.

Space Complexity: The space required is proportional to the length m of the bit array, independent of n, hence $O(m)$. The false positive probability formula arises from the probability that a bit remains unset after n insertions.

5.3 Automatic Parameter Determination

The efficiency of a Bloom filter depends heavily on choosing suitable values for m and k. For a target false positive probability p, the optimal number of hash functions k and bit array size m can be derived as:

$$k = \frac{m}{n} \ln 2,$$

$$m = -\frac{n \ln p}{(\ln 2)^2}.$$

These formulas allow automatic tuning of parameters given n and p. In practice, systems estimate n in advance and choose m and k accordingly to balance space efficiency and false positive probability.

6 Cuckoo Filter

6.1 Principles and Brief Introduction

The Cuckoo filter is based on cuckoo hashing and stores fingerprints of items. Each item can reside in one of two candidate buckets. If both are full, an eviction occurs and the displaced item is moved to its alternative bucket.

6.2 Complexity Derivation

Time Complexity: Lookup requires checking at most two buckets, so it is $O(1)$. Insertion is also expected $O(1)$, though it may involve a bounded sequence of evictions. Deletion simply clears the fingerprint from one of the two buckets, $O(1)$.

Space Complexity: Each element is represented by a small fingerprint of f bits, and with n elements stored, the total space is $O(nf)$.

6.3 Automatic Parameter Determination

For Cuckoo filters, key parameters include fingerprint size f, bucket size b, and the number of buckets m. The false positive rate p is approximately $2b/2^f$. To achieve a target p, we can automatically compute:

$$f = \lceil \log_2(2b/p) \rceil.$$

The number of buckets m is typically chosen so that $n/(mb)$ (the load factor) remains below a threshold (e.g., 0.95) to ensure high insertion success probability. Thus, m can be determined from n and b as:

$$m = \left\lceil \frac{n}{\alpha b} \right\rceil,$$

where α is the target load factor.

7 Implementation

7.1 Overview

To evaluate the performance characteristics of classical and probabilistic search algorithms, we implemented five representative approaches: Linear Search, Binary Search, Hash Table, Bloom Filter, and Cuckoo Filter. All implementations were written in `Python` to leverage its readability, rich standard library, and extensive ecosystem for numerical analysis and visualization. The codebase was structured with modularity and extensibility in mind, allowing each algorithm to be instantiated, benchmarked, and analyzed independently while sharing a consistent experimental interface.

7.2 Design Principles

Each algorithm was implemented in a plain, transparent coding style to ensure that the measured performance reflects the intrinsic algorithmic behavior rather than optimization artifacts. The source code follows object-oriented programming (OOP) principles, with each algorithm encapsulated as a dedicated class. Every class exposes standardized methods such as `add()`, `delete()`(where applicable), `contains()`, and `check()` , enabling fair comparison under identical experimental conditions.

The project structure separates algorithmic logic, dataset generation, and experimental evaluation into distinct modules. This modular organization allows flexible replacement or extension of algorithms without disrupting the benchmarking pipeline. Furthermore, common utilities, such as performance logging and memory profiling, are shared across all modules to maintain consistency.

7.3 Dataset Generation and Loading

A key component of our system is the custom dataset generator and loader, designed to support experiments on datasets containing up to 100 million records. The dataset generator produces synthetic username data, where each entry is a unique identifier in the form of a `UUID4` string. This approach guarantees global uniqueness and randomness, minimizing potential bias introduced by data patterns.

To handle large-scale data efficiently, the generator supports parallel execution using Python's `multiprocessing` library. The data generation process automatically partitions the workload into multiple chunks (or "trunks"), which can be processed concurrently. Once generation is complete, the dataset loader aggregates and merges these trunks, enabling seamless loading of large datasets into memory for testing. This design dramatically reduces total data preparation time and ensures scalability across different machine configurations.

7.4 Software Infrastructure and Reproducibility

All experiments were executed on a uniform software environment to ensure reproducibility. The environment includes Python 3.10, the `numpy` and `matplotlib` packages for numerical computation and visualization, and `psutil` for runtime memory monitoring. All implementation and experiment scripts are publicly accessible in the following repository: https://github.com/m-iDev-0792/COSC520-Assignments. The repository includes full source code, configuration files, dataset generation scripts, and benchmark code used for plotting and analysis.

8 Experiments

8.1 Experimental Setup

We conducted a series of controlled experiments to systematically compare the performance of all five algorithms under varying data scales. The datasets consisted of unique username entries ranging from small (10,000 items) to extremely large (100,000,000 items). The tested dataset sizes are listed as: [10,000, 40,000, 80,000, 100,000, 400,000, 800,000]. We set the expected false positive rate to 1% in both Bloom filter and Cuckoo filter algorithms.

Each algorithm was tested under two experimental rounds to evaluate its scalability and responsiveness to query load:

- **Round 1: Fixed Query Volume.** In this round, each algorithm executed exactly 10,000 search queries, regardless of dataset size. This setup isolates per-query efficiency from dataset scaling effects.
- **Round 2: Proportional Query Volume.** In this round, 10% of the total dataset items were queried. This configuration captures the performance impact under realistic load conditions that scale with data volume.

The experiments were conducted on my MacBook Pro laptop, which has a 2.3 GHz 8-Core Intel Core i9 CPU and 32 GB 2667 MHz DDR4 RAM.

8.2 Metrics and Data Collection

Four primary performance metrics were collected and analyzed:

1. **Memory Usage:** The average memory consumption was recorded using the `pympler` library during data loading and query execution phases.
2. **Total Query Time:** The aggregate time required to execute all queries in each round.
3. **Average Query Time:** The mean latency per query, computed as total query time divided by the number of queries.

(4) **Correctness:** The percentage of accurate results returned by the algorithm compared with ground-truth lookups. For probabilistic algorithms (Bloom and Cuckoo Filters), correctness was measured in terms of false positive rates.

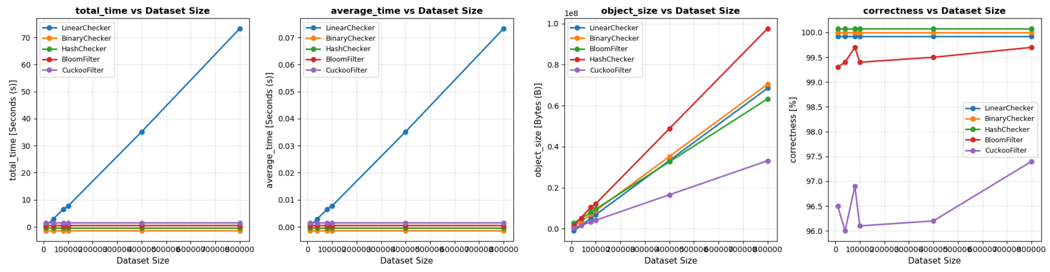

Fig. 1. Performance comparison of the five algorithms across varying dataset sizes with 10K queries. (Please note that lines of other four algorithms except linear search overlap in the left two subfigures.)

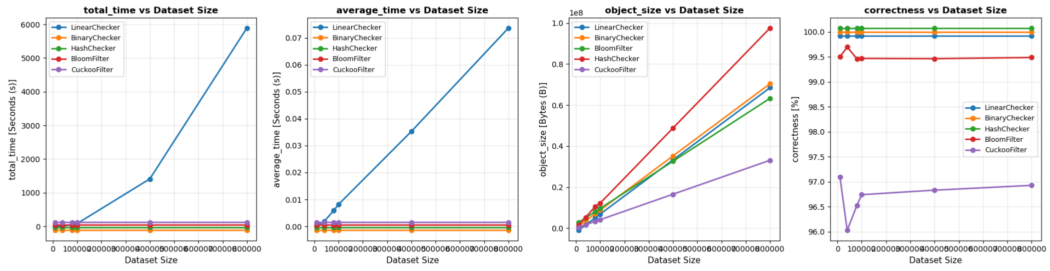

Fig. 2. Performance comparison of the five algorithms across varying dataset sizes with 10% queries of the dataset size. (Please note that lines of other four algorithms except linear search overlap in the left two subfigures.)

8.3 Results and Analysis

The experimental results were visualized through comparative plots illustrating performance trends across data scales and algorithm types. The result of round 1 experiment is shown as Figure 1. These figures collectively revealed several key findings:

- **Time Complexity:** Linear Search exhibited a clear linear increase in query time as dataset size grew, consistent with its $O(n)$ complexity. In contrast, Binary Search, Hash Table, Bloom Filter, and Cuckoo Filter all maintained near-constant average query times, confirming sublinear or constant-time retrieval behavior.
- **Memory Efficiency:** Bloom Filter and Cuckoo Filter demonstrated exceptional memory efficiency, consuming significantly less space compared to exact methods such as Hash Tables. However, this advantage came at the expense of lower correctness, as both probabilistic methods introduced a small but measurable false positive rate.
- **Scalability:** Both probabilistic filters scaled smoothly up to the largest tested datasets (100 million items), with minimal degradation in query latency. The Hash Table also performed efficiently but consumed noticeably more memory as the dataset grew.

- **Correctness:** While Linear Search, Binary Search, and Hash Table achieved perfect correctness (100%), Bloom and Cuckoo Filters exhibited minor inaccuracies (typically below 1%$\tilde{5}$% false positives), validating the theoretical trade-off between space and accuracy.

Overall, the results suggest that while deterministic data structures remain superior for accuracy-critical applications, probabilistic filters provide substantial advantages in memory-bound environments. The experimental framework also confirms that algorithmic complexity predictions align closely with observed empirical performance.

9 Applications and Use Cases

9.1 Overview

Different search and filtering algorithms are optimized for distinct computational objectives and resource constraints. While some prioritize accuracy and simplicity, others emphasize scalability, efficiency, or space savings at the cost of probabilistic errors. This section discusses representative application scenarios for each of the five algorithms implemented in this study, highlighting their advantages, limitations, and suitability in real-world contexts.

9.2 Linear Search

Application Scenarios:

- Small-scale data processing, such as configuration lookups or linear scans of short lists.
- Sequential file processing or systems where data is streamed and not indexed in advance, especially when we don't want to sort the data (or have space to sort the data).

Discussion: Linear search offers minimal implementation complexity and requires no preprocessing or data structure overhead. Its time complexity of $O(n)$ makes it impractical for large datasets, but it remains useful in contexts where datasets are small or unsorted, and setup costs for indexing would outweigh the benefits. Furthermore, it guarantees 100% correctness and deterministic performance, making it suitable for cases where predictability is more important than speed.

9.3 Binary Search

Application Scenarios:

- Search operations on sorted arrays or ordered databases, such as numeric datasets, log files, or ranking systems.
- Range queries and threshold detection in algorithmic trading, sensor data processing, or embedded systems.
- Index-based retrieval systems, e.g., searching sorted text indexes or B-tree-like structures.

Discussion: Binary search operates with logarithmic time complexity $O(\log n)$, offering significant efficiency over linear search when data is sorted. However, it assumes immutability or infrequent updates to the dataset, since maintaining sorted order incurs additional cost. As such, binary search is well-suited for read-heavy applications with static or rarely modified datasets.

9.4 Hash Table

Application Scenarios:

- Symbol tables in compilers or interpreters for variable and function lookups.
- Caching systems such as in-memory key-value stores (e.g., Redis).
- Databases and information retrieval systems that require fast exact matching and high query throughput.

- Implementations of sets, dictionaries, and associative arrays in general-purpose programming languages.

Discussion: Hash tables provide near-constant expected time complexity $O(1)$ for insertion, deletion, and lookup, making them one of the most versatile and widely used data structures in computer science. Their efficiency depends on a well-designed hash function to minimize collisions. Although hash tables require more memory than linear or binary search, their deterministic correctness and scalability make them ideal for high-performance applications with large volumes of unique keys.

9.5 Bloom Filter

Application Scenarios:

- Network and web caching systems (e.g., to check if an item is possibly stored before performing an expensive retrieval).
- Database systems for pre-checking membership, reducing unnecessary disk lookups.
- Distributed systems and blockchain for efficient set membership testing (e.g., duplicate detection, routing tables).
- Security applications such as spam detection or malicious URL filtering, where false positives are acceptable but false negatives are not.

Discussion: The Bloom filter is a probabilistic data structure that offers remarkable memory efficiency, trading absolute accuracy for space savings. It can determine whether an element is *possibly in a set* or *definitely not in a set*. This property makes it especially valuable in large-scale systems where storage is limited and exact membership testing is unnecessary. Its false positive rate can be tuned through parameter selection (number of hash functions and bit array size), enabling flexible trade-offs between memory use and accuracy.

9.6 Cuckoo Filter

Application Scenarios:

- High-performance caching systems and network intrusion detection systems (NIDS) where fast insertions and deletions are required.
- Real-time analytics platforms, especially in streaming data environments.
- Replacement for Bloom filters in applications needing dynamic updates (insertions and deletions) while maintaining similar space efficiency.

Discussion: Cuckoo filters extend the advantages of Bloom filters by supporting element deletion and improved lookup performance. They store compact hash "fingerprints" instead of entire keys, offering high accuracy with lower false positive rates compared to Bloom filters of similar size. These characteristics make Cuckoo filters attractive for dynamic or long-running systems that require frequent updates and bounded memory usage, such as online web services, telemetry pipelines, and IoT data aggregation systems.

9.7 Comparative Summary

Table 1 summarizes the typical application domains and trade-offs associated with each algorithm.

10 Conclusion

This report summarized the principles and derived the time and space complexities of linear search, binary search, hashing, Bloom filters, and Cuckoo filters. We also explained how to automatically determine optimal parameters for Bloom and Cuckoo filters given desired false positive probabilities and dataset sizes. The theoretical discussion was complemented by Python implementations

Table 1. Comparison of Algorithmic Applications and Trade-offs

Algorithm	**Typical Applications**	**Time Complexity**	**Memory Usage**	**Accuracy**
Linear Search	Small datasets, simple scans, unsorted data	$O(n)$	Low	100%
Binary Search	Sorted data retrieval	$O(\log n)$	Low	100%
Hash Table	Databases, caching, key-value stores	$O(1)$ (avg.)	Moderate	100%
Bloom Filter	Large-scale membership testing	$O(1)$	Very low	$\approx$ 99% (false positives)
Cuckoo Filter	Dynamic filters, stream systems	$O(1)$	Low	$\approx$ 99.5% (false positives)

and experiments. The results reinforce the trade-offs: probabilistic filters achieve superior space efficiency without substantially sacrificing speed or correctness.

Acknowledgments

The report referred some formulas introduced in following book [6] and links [2, 4, 5, 8**?**].Some parts of the code in this assignment were generated with the assistance of the Cursor Editor, a development environment that integrates multiple large language model (LLM) agents. The underlying backends of Cursor may include models such as ChatGPT [7], Claude [1], and others [3]. I hereby acknowledge their contribution. However, since I primarily used the Cursor Editor in a copilot-style fashion, the generated code largely extended my existing code structure and prior snippets. Therefore, the final implementation of this assignment should not be considered as being fully produced by LLMs.

References

[1] Anthropic. 2023. Claude AI. https://www.anthropic.com/claude. Accessed: 2025-10-03.
[2] Wikipedia contributors. 2025. *Cuckoo Filter*. https://en.wikipedia.org/wiki/Cuckoo_filter Accessed October 4, 2025.
[3] Cursor. 2023. Cursor: The AI Code Editor. https://cursor.sh. Accessed: 2025-10-03.
[4] B. Dupras. –. *Probabilistic Filters By Example*.
[5] GeeksforGeeks. 2025. *Bloom Filters - Introduction and Implementation*. https://www.geeksforgeeks.org/python/bloom-filters-introduction-and-python-implementation/ Last updated 23 Jul 2025, accessed October 4, 2025.
[6] Jon Kleinberg and Éva Tardos. 2006. *Algorithm Design*. Addison Wesley.
[7] OpenAI. 2022. ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/chatgpt. Accessed: 2025-10-03.
[8] L. Lim (Bill Mill's tutorial). –. *Bloom Filters by Example*.

Received 20 February 2007