Assignment 1: Algorithmic Approach to the Login Problem

RILEY EATON, University of British Columbia, Canada

1 Computational Complexity

This section analyzes the time and space complexity of five different data structures used for login checking: linear search, binary search, hash tables, Bloom filters, and Cuckoo filters.

1.1 Linear Search

Linear search stores elements in an unsorted array or list and searches sequentially through all elements.

Parameters:

- n = number of stored logins

Time Complexity:

- Insert: $O(n)$ - must check all existing elements to verify uniqueness
- Search: $O(n)$ - worst case requires examining all elements

Space Complexity: $O(n)$ - stores exactly n elements

Linear search provides no optimization for lookups, making it inefficient for large datasets [3].

1.2 Binary Search

Binary search maintains elements in a sorted array, enabling logarithmic search time through the divide-and-conquer approach.

Parameters:

- n = number of stored logins

Time Complexity:

- Insert: $O(n)$ - $O(\log n)$ for binary search to find position, but $O(n)$ for array shifting to maintain sorted order
- Search: $O(\log n)$ - divides search space in half at each step

Space Complexity: $O(n)$ - stores exactly n elements in sorted order

Binary search significantly improves lookup performance but insertion remains costly due to the need to maintain sorted order [3].

1.3 Hash Tables

Hash tables use a hash function to map keys to array indices, providing constant-time average case operations.

Parameters:

- n = number of stored logins
- m = size of hash table
- $\alpha = n/m$ = load factor

Time Complexity:

- Insert: $O(1)$ average case, $O(n)$ worst case with collisions
- Search: $O(1)$ average case, $O(n)$ worst case with collisions

Author's Contact Information: Riley Eaton, University of British Columbia, Kelowna, BC, Canada.

Space Complexity: $O(n)$ - with good hash functions and proper load factor management

Hash tables provide excellent average-case performance when the load factor α is kept below a threshold (typically 0.7) [3]. Python's `set` implementation uses open addressing with a load factor that triggers resizing.

1.4 Bloom Filters

A Bloom filter is a probabilistic data structure that uses multiple hash functions to map elements into a bit array, allowing for space-efficient membership testing with possible false positives [1].

Parameters:

- n = estimated number of elements to insert
- m = size of bit array (in bits)
- k = number of hash functions
- p = desired false positive probability

Time Complexity:

- Insert: $O(k)$ - requires k hash function evaluations
- Search: $O(k)$ - requires k hash function evaluations

Space Complexity: $O(m)$ bits

The optimal number of hash functions is given by:

$$k = \frac{m}{n} \ln 2 \tag{1}$$

The optimal bit array size for a desired false positive probability p is:

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

The false positive probability after inserting n elements is approximately:

$$p \approx \left(1 - e^{-kn/m}\right)^k \tag{3}$$

Bloom filters trade accuracy for space efficiency, making them ideal when false positives are acceptable but false negatives are not [2]. My implementation uses a backing hash table to verify positive results and eliminate false positives.

1.5 Cuckoo Filters

Cuckoo filters extend Bloom filters by storing fingerprints of items using cuckoo hashing, enabling deletions while maintaining space efficiency [5].

Parameters:

- n = number of elements to insert
- m = number of buckets
- b = bucket size (entries per bucket)
- f = fingerprint size (in bits)
- α = load factor (typically ≤ 0.95)

Time Complexity:

- Insert: $O(1)$ average case, with a small probability of failure requiring rehashing
- Search: $O(1)$ - checks at most 2 buckets with b entries each
- Delete: $O(1)$ - unlike Bloom filters, supports deletion

Space Complexity: $O(n \cdot f)$ bits, where f is typically 4-16 bits

The false positive rate is approximately:

$$\epsilon \approx \frac{2b}{2^f} \tag{4}$$

where f is the fingerprint size in bits and b is the bucket size. For a load factor α, the total capacity is $C = \alpha \cdot b \cdot m$ [5].

Cuckoo filters provide similar space efficiency to Bloom filters while supporting deletion and offering better lookup performance for certain parameters.

1.6 (Bonus) Tries

A trie (prefix tree) is a tree-based data structure where each node represents a character and paths from root to leaf represent complete strings, making it well-suited for string operations [3].

Parameters:

- n = number of stored logins
- m = average length of login strings
- σ = alphabet size (typically ≈ 100 for alphanumeric + symbols)

Time Complexity:

- Insert: $O(m)$ - traverse/create m nodes
- Search: $O(m)$ - traverse path of length m
- Delete: $O(m)$ - locate and remove the string

Space Complexity: $O(n \cdot m \cdot \sigma)$ worst case, but much better with shared prefixes

Tries offer deterministic performance without hash collisions and support efficient prefix operations ($O(m + k)$ to find all logins with a given prefix). However, space overhead is higher than hash tables when strings lack common prefixes, and $O(m)$ time is slower than hash table's $O(1)$ for typical login lengths.

1.7 Complexity Comparison

Table 1 summarizes the time and space complexity of all five approaches.

Table 1. Computational Complexity Comparison

Data Structure	**Insert**	**Search**	**Space**
Linear Search	$O(n)$	$O(n)$	$O(n)$
Binary Search	$O(n)$	$O(\log n)$	$O(n)$
Hash Table	$O(1)^*$	$O(1)^*$	$O(n)$
Bloom Filter	$O(k)$	$O(k)$	$O(m)$ bits
Cuckoo Filter	$O(1)^*$	$O(1)$	$O(nf)$ bits

*Average case complexity; worst case is $O(n)$

n = number of elements, k = number of hash functions

m = bit array size, f = fingerprint size in bits

2 Python Experiments

2.1 Test Data Generation

All experiments use realistic login data generated using the Faker library [4], which produces human-like usernames following common patterns:

- Standard usernames (e.g., `john_smith`)
- First/last names with numeric suffixes (e.g., `alice123`)
- Email prefixes (e.g., `user.name`)
- Random alphanumeric combinations

A dataset of ~6.5 million unique logins was pre-generated and stored in a text file[1] to ensure consistency across all test runs. This approach eliminates variability from data generation and allows for reproducible results, while not having to waste time re-generating logins for each test.

2.2 Test Size Selection

The experimental design uses five test sizes: **100, 500, 1,000, 2,000, and 5,000** logins. This progression was chosen to reveal algorithmic scaling behavior across different dataset sizes.

This range was chosen to reveal scaling behavior: $n = 100$ establishes baseline performance where all algorithms are reasonable, while $n = 5000$ approaches practical limits for inefficient algorithms (linear search performs ~12.5 million comparisons). The intermediate sizes capture the transition where algorithmic differences become measurable. Testing beyond 5,000 elements would provide diminishing returns—linear search becomes prohibitively slow, while constant-time structures show only noise rather than algorithmic differences.

2.3 Experimental Procedure

For each test size n: insert n unique logins from pre-generated data, then perform n lookups (50% existing, 50% non-existent). Two metrics are tracked: **wall-clock time** (real-world performance including overhead) and **comparison count** (implementation-independent validation of theoretical complexity). This yields 25 trials total (5 structures × 5 sizes).

2.4 Implementation Details

All implementations use Python 3.12 with the following specifics:

- **Linear Search:** Python list with sequential iteration
- **Binary Search:** Sorted Python list with manual binary search implementation
- **Hash Table:** Python's built-in `set` (hash table with open addressing)
- **Bloom Filter:** `pybloom-live` library with $n = 1,000,000$ capacity and 0.001 error rate
- **Cuckoo Filter:** `cuckoo-filter` library with table size 10,000, bucket size 4, fingerprint size 8 bits

Both probabilistic filters (Bloom and Cuckoo) use a backing hash table to verify positive results, eliminating false positives at the cost of additional space overhead. This hybrid approach ensures correctness while maintaining the performance benefits of probabilistic filtering for negative lookups.

2.5 Experiment Results

This section presents the empirical results from performance benchmarking all five login checking implementations across dataset sizes ranging from 100 to 5,000 elements.

2.5.1 Wall-Clock Performance.

Figure 1 shows the wall-clock execution time for both insertion and lookup operations across all algorithms.

Linear search shows dramatic performance degradation (0.49s insertion, 0.57s lookup at $n = 5000$), compressing other algorithms into near-flat lines. Figure 2 provides a zoomed view.

[1] https://github.com/rileyeaton-ubc/ubc-520-a1/blob/3bb39197d86925f1904856b6149dbae908bb190b/data/logins.txt

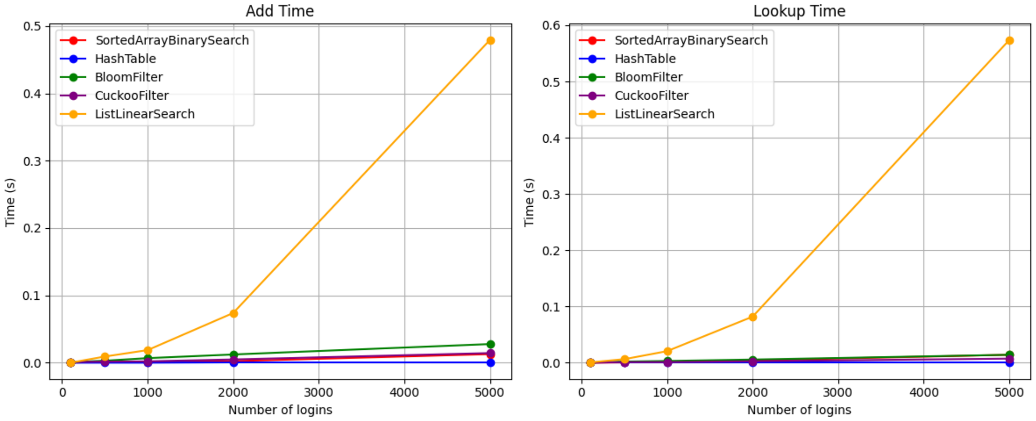

Fig. 1. Wall-clock performance comparison for all algorithms. Linear search dominates the scale, showing clear quadratic growth for insertions and linear growth for lookups.

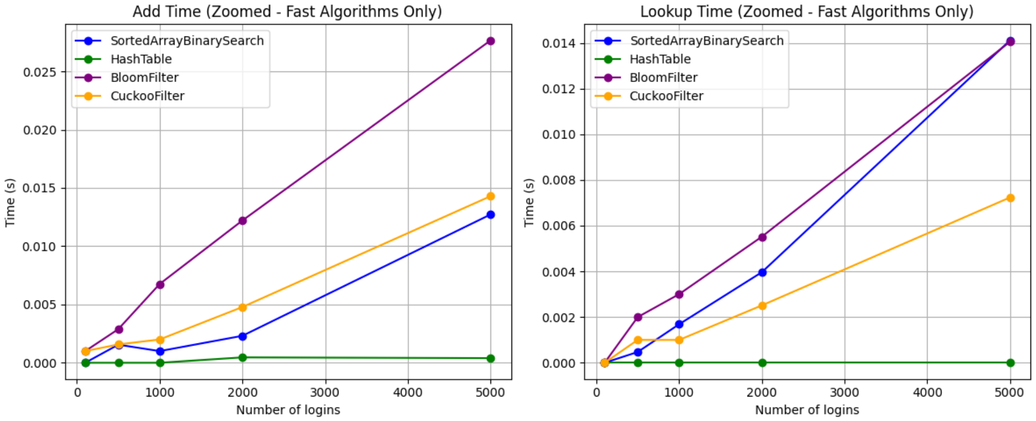

Fig. 2. Zoomed wall-clock performance for efficient algorithms. Hash table maintains near-constant time, while binary search, Bloom filter, and Cuckoo filter show varying degrees of time growth despite theoretical O(log n) and O(1) complexities.

2.5.2 Theoretical Complexity Validation. To validate that implementations correctly follow their theoretical complexity, we examine comparison counts rather than wall-clock time. Figure 3 shows average comparisons per operation.

Comparison counts validate theoretical predictions: linear search shows perfect $O(n)$ growth (50 to 2,500 comparisons), binary search exhibits logarithmic growth (6.4 to 12.2, matching $\log_2(n)$), and hash-based structures maintain constant comparisons (~1-1.5) regardless of size.

2.5.3 Key Insights and Analysis. The dual-metric approach (wall-clock time vs. comparison counts) reveals critical insights about algorithm performance:

Theoretical vs. Practical Performance Gap. Binary search demonstrates the theory-practice gap: while performing $O(\log n)$ comparisons (12.2 at $n = 5000$), wall-clock time grows nearly linearly

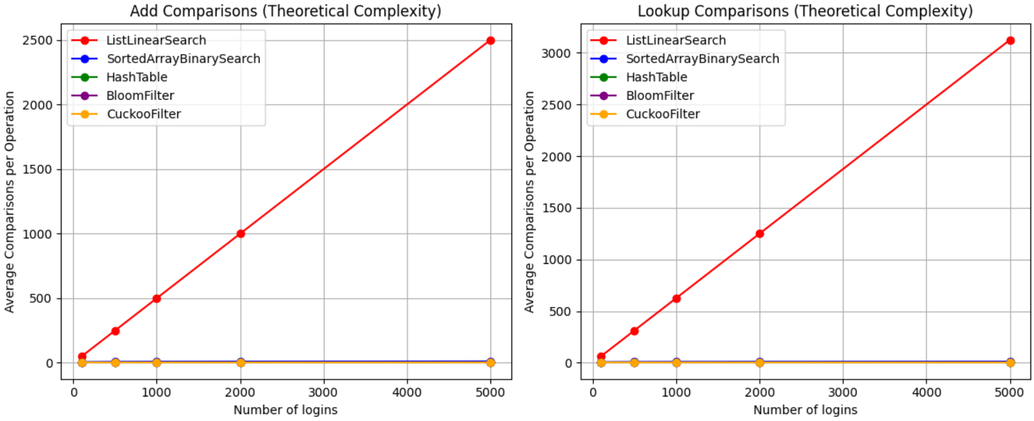

Fig. 3. Average comparison counts per operation. This metric validates theoretical complexity: linear search shows linear growth, binary search shows logarithmic growth, and hash-based approaches show constant comparisons.

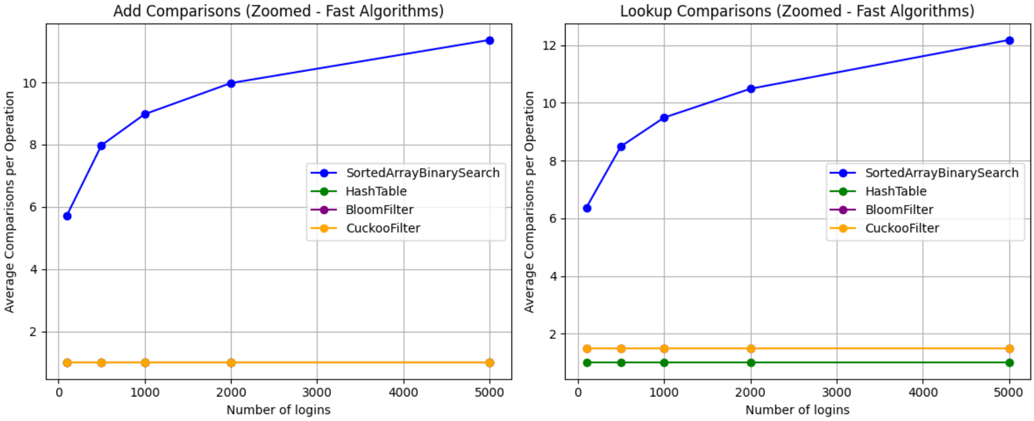

Fig. 4. Zoomed comparison counts for efficient algorithms. Binary search exhibits the characteristic logarithmic curve, while hash-based methods remain perfectly flat, validating theoretical predictions.

due to Python string comparison overhead (~40ns per character across 8.6-character strings). String operations dominate the logarithmic advantage, creating ~800 microseconds per comparison including overhead.

Implementation Overhead in Probabilistic Filters. While maintaining constant comparisons (validating $O(1)$ theory), Bloom and Cuckoo filters show linear wall-clock growth due to backing set rehashing and library overhead. At $n = 5000$: Bloom filter insertion takes 27.7ms vs. 1.0ms for pure hash table; Cuckoo filter performs better (12ms) but still grows linearly.

Hash Table as the Optimal Choice. Python's native hash table (set) demonstrates genuine $O(1)$ behavior in both comparison counts and wall-clock time, with minimal overhead and near-flat performance across all dataset sizes.

When Simpler Approaches Make Sense. Despite poor scaling, linear/binary search remain viable for small datasets ($n < 100$), memory-constrained environments, or when sorted access is required.

The Value of Empirical Validation. Theoretical complexity analysis guides scalability understanding, but constant factors and implementation overhead can dominate practical performance. Comparison counts validate theoretical correctness; wall-clock measurements expose real-world constraints. Both perspectives are necessary for informed algorithm selection.

References

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

[2] Andrei Broder and Michael Mitzenmacher. 2004. Network applications of bloom filters: A survey. *Internet Mathematics* 1, 4 (2004), 485–509.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). MIT Press, Cambridge, MA.

[4] Faker Contributors. 2024. Faker: A Python package that generates fake data. https://faker.readthedocs.io/. Accessed: 2025-10-03.

[5] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. 2014. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*. ACM, 75–88.

Acknowledgments

Anthropic's Claude was used to generate some documentation once implementation was complete, as well as reviewing the finalized report to suggest edits. The complete source code and experimental data for this project are available at https://github.com/rileyeaton-ubc/ubc-520-a1.