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 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

This comparison show that hash tables provide the best average-case performance for exact membership testing, while probabilistic filters (Bloom and Cuckoo) offer superior space efficiency at the cost of potential false positives. Linear and binary search serve as baselines, with binary search providing logarithmic lookup time but linear insertion cost due to array shifting.

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 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.

2.2.1 Justification for Size Range. **Starting Point (n=100):** This small size establishes baseline behavior where all algorithms perform reasonably well. It serves as a control point showing that even inefficient algorithms can handle small datasets, with differences measured in microseconds rather than seconds.

Early Growth (n=500): At this scale, quadratic algorithms like linear search begin to show noticeable degradation. With $O(n^2)$ insertion complexity, linear search performs approximately $500^2/2 = 125,000$ comparisons during insertion, compared to just 500 for hash-based approaches. This 250x difference becomes measurable.

Moderate Scale (n=1,000): This represents a typical small-to-medium application scenario (e.g., active users in a small web service). The gap between linear (~500,000 comparisons) and logarithmic approaches becomes substantial. Binary search requires only $\log_2(1000) \approx 10$ comparisons per lookup, demonstrating the logarithmic advantage.

Scaling Point (n=2,000): Doubling from 1,000 to 2,000 elements reveals scaling characteristics:

- Linear search: 4x increase in comparisons (from 500K to 2M)
- Binary search: minimal increase ($\log_2(2000) \approx 11$ vs. 10)
- Hash tables: constant performance regardless of size

Upper Limit (n=5,000): This size approaches practical limits for inefficient algorithms while remaining computationally feasible for benchmarking. Linear search performs approximately 12.5 million comparisons for insertion alone, creating multi-second delays that would be unacceptable in production systems. Meanwhile, constant-time structures maintain sub-second performance.

2.2.2 Why Not Larger Sizes? Testing beyond 5,000 elements faces diminishing returns:

(1) **Linear/Binary search impracticality:** At $n = 10,000$, linear search would perform 50 million comparisons, taking prohibitively long and providing no additional insight—we already know it scales poorly.
(2) **Constant-time convergence:** Hash tables, Bloom filters, and Cuckoo filters all exhibit $O(1)$ performance. Beyond 5,000 elements, their timing curves flatten, showing only minor variations due to cache effects and system noise rather than algorithmic differences.
(3) **Statistical significance:** The chosen range provides sufficient data points to establish clear trends. Five orders of magnitude captures the algorithmic behavior across practical scales.

2.3 Experimental Procedure

For each test size n, the experiment proceeds as follows:

(1) **Initialization:** Create an empty data structure instance
(2) **Insertion Phase:** Add n unique logins from the pre-generated dataset, measuring:
   - Total insertion time (wall-clock)
   - Number of comparisons performed
(3) **Lookup Phase:** Perform n lookup operations with a 50/50 mix:
   - 50% existing logins (true positives)
   - 50% non-existent logins (true negatives)
(4) **Metrics Collection:** Record timing and comparison statistics

This procedure is repeated for all five data structures at each size, yielding 25 experimental trials total.

2.4 Performance Metrics

Two primary metrics quantify performance:

Wall-Clock Time: Measures actual execution time in seconds, capturing real-world performance including constant factors, cache effects, and implementation overhead. This metric reflects what users would experience in production.

Comparison Count: Tracks the number of element comparisons or hash function evaluations. This implementation-independent metric directly reflects theoretical complexity, allowing validation of asymptotic bounds (e.g., verifying that binary search indeed performs $O(\log n)$ comparisons).

The combination of these metrics provides both theoretical validation and practical insight into algorithm performance.

2.5 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.

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 of documentation once implementation was complete.