The Login Checker Problem - Benchmarking various lookup algorithms in different data structures

HAMZA MUHAMMAD ANWAR, University of British Columbia Okanagan, Canada

This is the report for the first assignment of the COSC 520 (Advanced Algorithms) course. The objective of the assignment was to: (1) review hash tables, (2) explore advanced data structures, and (3) numerically compare the algorithms. A benchmarking of different algorithms was done by checking their existence using linear and binary search, hashing in hash tables, bloom filters, and cuckoo filters. The time and space complexities were analyzed, and the results were plotted. The GitHub repository containing my implementations of this assignment can be found here: https://github.com/Hamza-Awr/COSC-520-Assignment-1.git.

Additional Key Words and Phrases: Algorithms, Lookup, Linear Search, Binary Search, Hash Tables, Hashing functions, Bloom Filters, Cuckoo Filters

1 Introduction

In assignment 1 of COSC 520, we were tasked to solve the login checker problem. The login checker problem is to quickly check that a login name has not already been taken so that all logins are unique. Assume you have millions of logins (aka usernames), and a user creates a new account. You need to ensure the new login is not a duplicate of an existing one. We were required to:

(1) store all logins in a list, and check existence using linear search;
(2) store all logins in a sorted array, and check existence using binary search;
(3) store logins in a hash table, and check existence by hashing.
(4) store logins in two advanced data structures: bloom and cuckoo filters, and check their existence

The algorithms and code necessary to generate them were all implemented using Python.

2 Linear Search

In linear search, we linearly check the existence of elements by going through them in the array one by one.

2.1 Time complexity

The only parameter in linear search is n: the number of elements in the list. The time complexity is:

- Best case: $O(1)$, when the target is the first element of the list
- Average case: $O(n/2) = O(n)$, when the target is in the middle, but that is still $O(n)$
- Worst case: $O(n)$ when the target is the last element of the list

2.2 Space complexity

The space complexity of linear search is $O(n)$, as it needs to store an n amount of elements. The auxiliary space complexity is $O(1)$, as it uses a constant number of variables to compare and index. For example, to store the current element of the list in the loop and to compare it with the ones that need to be searched.

3 Binary Search

In binary search, the search space is halved every time by going to the middle of the list, checking if the target is greater or less than the middle, and discarding the unneeded half. This is done

Author's Contact Information: Hamza Muhammad Anwar, hanwar02@student.ubc.ca, University of British Columbia Okanagan, Kelowna, British Columbia, Canada.

iteratively until the target is found. Binary search requires the list to first be sorted. The only parameter in linear search is n: the number of elements in the **sorted** list. The time complexity is:

- Best case: $O(1)$, if the target is found in the middle first step.
- Average/Worst case: $O(\log n)$, since the search space halves each iteration [4].

3.1 Space complexity

The space complexity of linear search is $O(n)$, as it needs to store an n amount of elements. The auxiliary space complexity of binary search (iterative) is also $O(1)$, as it uses a constant number of variables to compare and index.

4 Hashing using Hash Tables

In simple terms, hash tables store values as follows: the element that needs to be stored is passed through a hash function, which converts that element (called a key) into a value. The key is then stored in that value's index (called a bucket) in a hash table. Think of it like a smart locker system. When that key needs to be searched, simply compute the index again from the key and go to that index to check the value stored there [2].

4.1 Time complexity

In hash tables, the parameters are:

- n: number of elements stored.
- m: number of slots (buckets).
- $\alpha = n/m$ load factor (average elements per bucket).

The time complexity is:

- Best case: $O(1)$, assuming a good hashing function is used and the load factor is low
- Worst case: $O(n)$ when all elements are hashed to the same bucket

4.2 Space complexity

The space complexity is $O(m + n)$ since we need m buckets and store n elements.

5 Bloom Filters

A bloom filter is an advanced, space-efficient, and probabilistic data structure that uses a k number of hashing functions to store elements in it. An element is hashed using both functions, and the index of a bit array (of m size) is given as output. The bits at those indices are set to one. To check whether that element exists, we hash it and then check whether the bits at the indices specified by the hashing function are 1. If all are 1, then the element *probably* exists. If any of them are 0, then the element definitely does not exist. Thus, Bloom filters are associated with a probability of false positives, but this can be mitigated by controlling the parameters m and k.

5.1 Time complexity

In Bloom filters, the parameters are:

- n: number of inserted elements.
- m: number of bits in the bit array.
- k number of hash functions.

The time complexity is:

- $O(k)$ fixed, performs constant k hashes. The best and worst cases are the same, as the number of hashes to perform does not depend on how full the filter is [1, 6].

5.2 Space complexity

The space complexity is $O(m)$ since we only need the bit array of length m.

6 Cuckoo Filters

A cuckoo filter, like a Bloom filter, is a probabilistic data structure that uses multiple hashing functions. They are an improved version as they support deletion and can achieve lower space overhead than space-optimized Bloom filters [3, 5].

A cuckoo filter uses cuckoo hashing to store fingerprints of items. These fingerprints are also calculated by a hash function. Two possible buckets are computed for the fingerprint using two hashing functions, and the fingerprint is stored in either of them. If one is full, it goes to the other, and if that is also full, it evicts the item present in that bucket and replaces it [7]. This is cuckoo hashing. The evicted item then goes to another bucket calculated using the other hashing function [3].

6.1 Time complexity

In cuckoo filters, the parameters are:

- n: number of elements.
- b: number of buckets.
- f: fingerprint size (bits per item).
- s: bucket size (number of slots per bucket).

The time complexity is:

- $O(1)$; only needs to check two buckets to look up the value.

6.2 Space complexity

The space complexity is $O(n)$ since we need to store an n amount of fingerprints of size f each, which makes it $O(fn)$, but since f is usually a constant, $O(fn) \approx O(n)$.

7 Complexity Summary

A summary of the time and space complexities of all the methods is given in Table 1.

Method	Lookup Time Complexity	Lookup Space Complexity
Linear Search	$O(n)$	$O(n)$
Binary Search	$O(\log n)$	$O(n)$ (iterative)
Hashing	$O(1)$	$O(n)$
Bloom Filter	$O(k)$ (k = number of hash functions)	$O(m)$ (m = number of bits in bit array)
Cuckoo Filter	$O(1)$	$O(n)$

Table 1. Summary of Time and Space Complexities of Different Search Methods

8 Implementation

The code was implemented in Python. The datasets were synthesized each time by randomly generating strings of five random ASCII characters. The number of usernames searched was kept constant at 1000 for each algorithm and data structure. It was not scaled with the value of n as that would give a false comparison between the algorithms, due to their differing time complexities.

The maximum value of n used was 10 million. I tried to insert 1 billion usernames; the code would not execute because my machine would run out of memory. However, 10 million provides a dataset large enough to observe the asymptotic trends. The generated time plots already match the theoretical complexities (linear, logarithmic, constant, etc.) described previously in the report, showing the expected behaviour for each algorithm. Note that only the time complexities of Lookup were measured.

For the Hash Table and Bloom Filter, I used the MurmurHash3 hashing function. This hashing function is fast to execute as it's not cryptographic. I reasoned that for this assignment, high security was not a necessity; therefore, a cryptographic hashing function is not required. For the cuckoo filter, however, I used SHA256 (for the fingerprints and the two other hashing functions). I tried MurmurHash3, but it was taking a significantly longer amount of time than SHA256.

The complete plot of time complexities for each algorithm is shown in Figure 1.

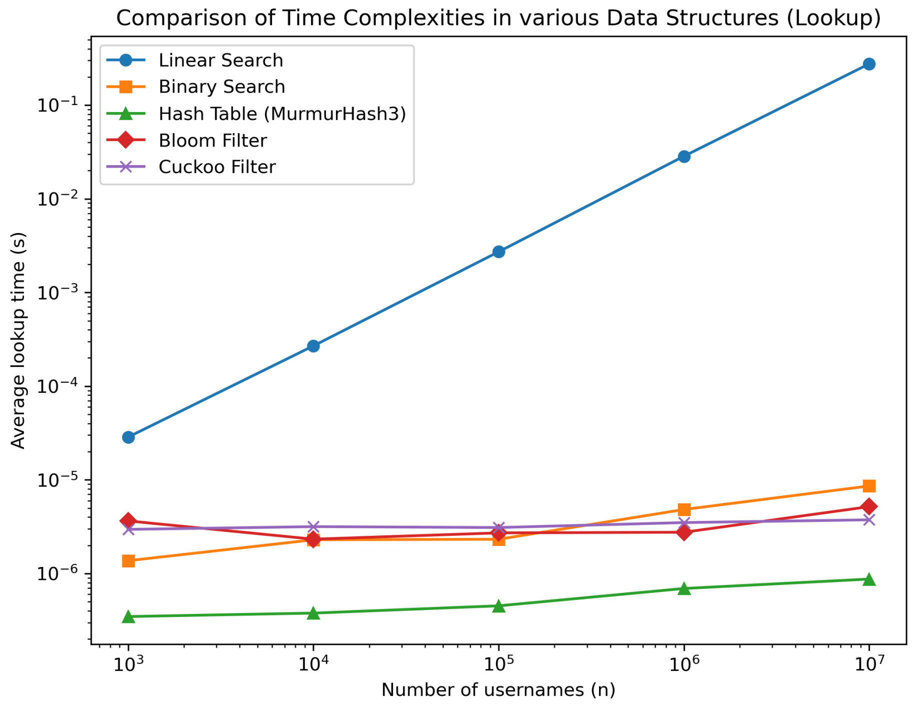

Fig. 1. A plot showing the comparison of Lookup time complexities of various data structures and algorithms.

9 Analysis

Figure 1 demonstrates the average lookup times obtained from the implemented algorithms compared to their theoretical time complexities.

9.1 Linear Search ($O(n)$)

- Each lookup scans the list of usernames one by one until it finds the target.
- As the number of usernames n increases, the time per lookup grows roughly proportionally, producing a straight-line trend on the log-log plot, consistent with $O(n)$ behaviour.

9.2 Binary Search ($O(\log n)$)

- Binary search repeatedly divides the search space of sorted usernames in half.
- Doubling the input size adds roughly one additional comparison per lookup ($\log_2(2n) - \log_2(n) = 1$. My implementation multiplies the input size by 10 each time, which gives roughly 3 extra comparisons. This matches the rough logarithmic growth observed in the plot.

9.3 Hash Table ($O(1)$)

- Using MurmurHash3 with chaining ensures that lookups generally require constant time, regardless of n.
- The average lookup time remains nearly flat across increasing n, with negligible fluctuations of time, reflecting constant-time behaviour.

9.4 Bloom Filter ($O(k)$)

- Each lookup computes k hash functions and checks the corresponding bits.
- Since k is fixed in the implementation, the lookup time does not increase with n.

9.5 Cuckoo Filter ($O(1)$)

- Lookups check only two possible buckets for the fingerprint of the item.
- As with the hash table, the lookup time stays roughly constant even as n grows, confirming the expected $O(1)$ behaviour.

9.6 Overall Analysis

The plotted results validate the theoretical complexities: linear search grows linearly, binary search grows logarithmically, and hash-based structures (hash table, Bloom filter, Cuckoo filter) maintain nearly constant lookup times. Minor variations for large n are likely due to memory access patterns, CPU caching, or hash collisions, but the overall trends confirm the theoretical analysis.

10 GitHub Repository

The GitHub repository containing my implementations of this assignment can be found here: https://github.com/Hamza-Awr/COSC-520-Assignment-1.git. The synthesized datasets are in the code for each algorithm itself. All instructions on the setup and running of the code can be found in the readme.md file.

Acknowledgments

Most of the code was written by ChatGPT, and it was properly reviewed and executed by the author to ensure the correct results are obtained.

A Plots of the Time Complexities of Individual Data Structures

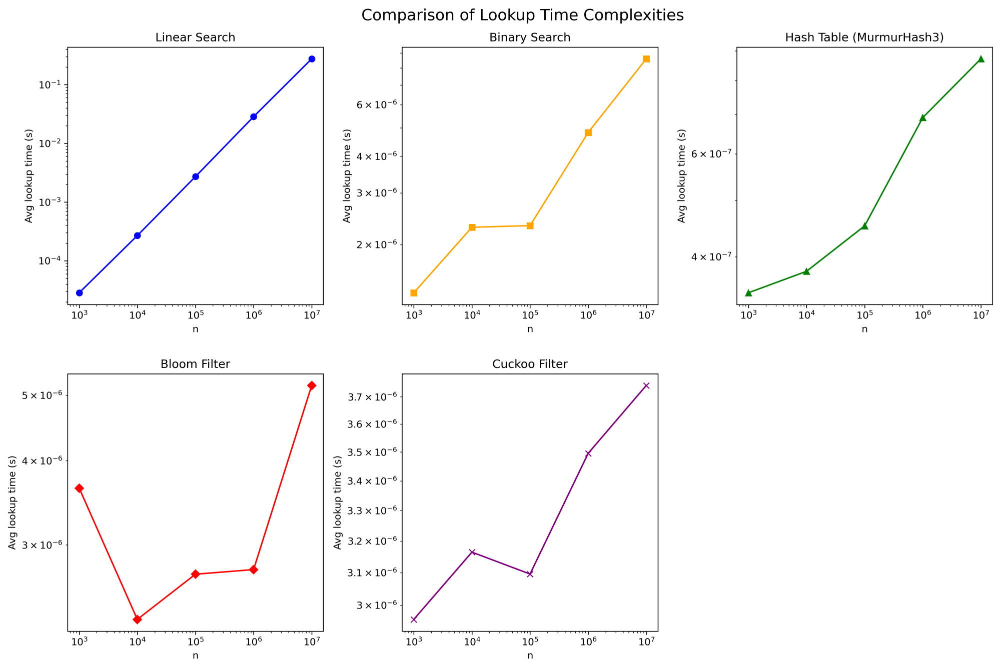

Fig. 2. The individual lookup time complexities of each data structure.

References

[1] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (July 1970), 422–426. doi:10.1145/362686.362692

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms* (2nd ed.). The MIT Press. http://www.amazon.com/Introduction-Algorithms-Thomas-H-Cormen/dp/0262032937%3FSubscriptionId%3D13CT5CVB80YFWJEPWS02%26tag%3Dws%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0262032937

[3] 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 on Conference on Emerging Networking Experiments and Technologies*. ACM, Sydney Australia, 75–88. doi:10.1145/2674005.2674994

[4] Jon Kleinberg and Éva Tardos. 2006. *Algorithm Design*. Addison Wesley.

[5] MichaelMitzenmacher. [n. d.]. Michael Mitzenmacher - Bloom Filters, Cuckoo Hashing, Cuckoo Filters, Adaptive Cuckoo Filters, and Learned Bloom Filters - MediaSpace @ Georgia Tech. https://mediaspace.gatech.edu/media/Michael+Mitzenmacher+-+Bloom+Filters,+Cuckoo+Hashing,+Cuckoo+Filters,+Adaptive+Cuckoo+Filters,+and+Learned+Bloom+Filters/1_96rboz83.

[6] Bill Mill. [n. d.]. Probabilistic Filters By Example: Cuckoo Filter and Bloom Filters. https://bdupras.github.io/filter-tutorial/.

[7] Rasmus Pagh and Flemming Friche Rodler. 2001. Cuckoo Hashing. In *Algorithms — ESA 2001*, Friedhelm Meyer auf der Heide (Ed.). Springer, Berlin, Heidelberg, 121–133. doi:10.1007/3-540-44676-1_10