Task 1:

Computational Complexity of Login Checker Methods

1. **Linear Search [3]**
    - **Time Complexity**: O(n)
    - **Space Complexity**: O(n)
    - **Parameters**:
        - n: Number of stored logins.
    - *Explanation*: Checks each login sequentially, leading to linear time. Stores all logins in a list.
2. **Binary Search [4]**
    - **Time Complexity**: O(log n)
    - **Space Complexity**: O(n)
    - **Parameters**:
        - n: Number of stored logins (sorted array).
    - *Explanation*: Divides the search space by half in each step. Requires a sorted array.
3. **Hashing (Hash Table) [5]**
    - **Time Complexity**: O(1) average case, O(n) worst case (due to collisions).
    - **Space Complexity**: O(n)
    - **Parameters**:
        - n: Number of stored logins.
    - *Explanation*: Uses a hash function to map logins to buckets. Average case assumes uniform hashing.
4. **Bloom Filter [6]**
    - **Time Complexity**: O(k)
        - $k = \frac{m}{n} ln2$
    - **Space Complexity**: O(m) bits

- $m = -\frac{nlnP}{(ln2)^2}$

- **Parameters**:
  - n: Estimated number of logins.
  - m: Size of the bit array.
  - k: Number of hash functions.
  - P: the desired probability of false positives
- *Explanation*: Employs k hash functions to set bits in an m-sized array. Space depends on m (typically proportional to n for a target false positive rate P).

5. **Cuckoo Filter [7]**
   - **Time Complexity**: O(1)
   - **Space Complexity**: O(f · n) bits (where f is a constant fingerprint size)
   - **Parameters**:
     - n: Number of logins.
     - f: Fingerprint size in bits (determines false positive rate).
   - *Explanation*: Stores compact fingerprints in a cuckoo hash table. Checks two buckets per lookup. Space scales linearly with n and f.

Task 2 [8]:

To analyze the time complexity, we generate 10 million random login (each with 8 characters), and I tested it on 1000 different inquiries. Half the inquiries are in the dataset and the other half are new login names that don't exist in the dataset.

Dataset link: https://github.com/AhmadMustafa015/COSC-520---Assignments/blob/main/Assignment%201/dataset.csv

Figure 1 presents the time complexity comparison for linear search, binary search, hashing, Bloom filter, and Cuckoo filter. Due to its inefficiency, the simulation for linear search was restricted to 1 million logins, as running it for larger datasets would be excessively time-consuming. The results show that linear search had the longest runtime, scaling linearly with the number of logins. In contrast, the other algorithms maintained a nearly constant performance regardless of dataset size. However, binary search began to show an increase in execution time once the number of logins surpassed 2 million.

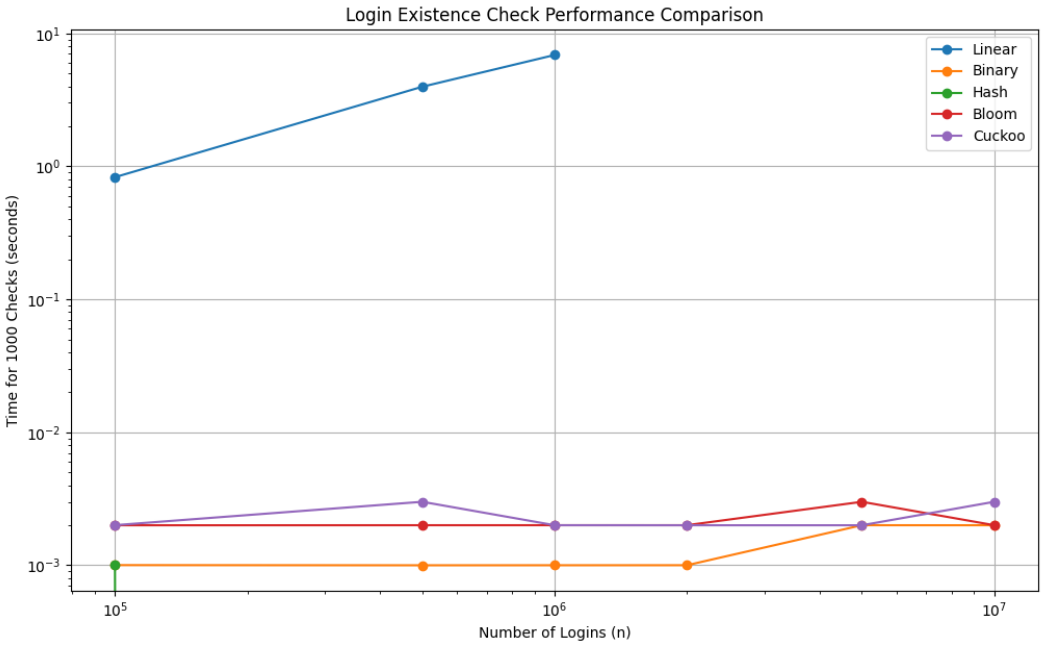

Figure 1: Time complexity comparison for linear search, binary search, hashing, Bloom filter, and Cuckoo filter. For different datasets sizes.

Task 3 [8]:

GitHub repo link: https://github.com/AhmadMustafa015/COSC-520---Assignments/blob/main/Assignment%201/task_3.py

Bonus [9]:

Data Structure: Trie (Prefix Tree)

A Trie is a tree-like data structure in which each node represents a single character of a string. All descendants of a node share a common prefix of the string associated with that node. For storing login names, each path from the root to a terminal node corresponds to a complete login.

- **Nodes and Edges:**
  Each node typically contains an array or a dictionary mapping characters to child nodes. There is also a flag (or similar marker) at nodes to indicate that a valid login ends at that point.
- **Storage:**
  Inserting and searching for strings only requires storing one copy of each common prefix, which can be memory efficient when many strings share prefixes. However, for datasets with very little common prefix, the memory usage might be higher than that of a hash table.

How It Works

1. **Insertion:**
   To insert a login, you start at the root and follow the path corresponding to each character in the string. If a character's node does not exist, you create a new node. Once you have processed all characters, you mark the final node as representing the end of a valid login.
2. **Search:**
   To check whether a login already exists, you again traverse the tree following the characters of the login. If you reach a node that does not exist or if the terminal marker is not set for the final character, the login is not present in the Trie. Otherwise, it is.
3. **Advantages:**
   - **Fast Lookup:** The time to insert or search is proportional to the length of the string L (i.e., O(L)), which is effectively constant if L is bounded by a small maximum (as is typical with login names).
   - **Prefix Queries:** Tries are particularly efficient for prefix queries, which can be useful if you also need to support auto-completion or other similar operations. Although the hash table has a relatively faster lookup speed, it

only supports the exact match of the whole string. The trie solution is more flexible to support more applications, such as auto-complete. Also, we can easily print all the words in the dictionary in alphabetic order with a trie.

Therefore, if we want a full-text lookup application, the hash table is better as it has a faster lookup speed. However, if we want to return all the words that match a prefix, the trie is our solution.

Complexity

- **Time Complexity:**
  - **Insertion:** O(L)
  - **Search:** O(L)
    where L is the length of the string (login). Since login names are generally of modest length, these operations are very efficient in practice.
- **Space Complexity:**
  - In the worst case, the Trie requires O(N×L) space, where N is the number of logins and L is the average length.
  - However, due to the shared prefixes, the effective space usage is often much lower.

References

1. *Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors.*
2. *Fan, B., Andersen, D. G., Kaminsky, M., & Mitzenmacher, M. (2014). Cuckoo Filter: Practically Better Than Bloom.*
3. *https://www.geeksforgeeks.org/linear-search*
4. *https://www.geeksforgeeks.org/binary-search*
5. *https://iq.opengenus.org/time-complexity-of-hash-table*
6. *https://bdupras.github.io/filter-tutorial*
7. *https://iq.opengenus.org/cuckoo-filter/*
8. *Generative AI: OpenAI O3-mini*
9. *https://www.geeksforgeeks.org/trie-insert-and-search*