Assignment 1

Prepared by MahmoudSoliman

1) A concise pdf file listing the time and space computational complexity of: linear search, binary search, hashing, Bloom filter, Cuckoo filter. Use appropriate parameters to describe the complexity, e.g., Bloom filter uses n (estimated number of elements/logins), m (bit array of m bits), and k (number of hash functions). You do not need to justify the complexity; only to explain each parameters used and provide the formula with appropriate references.

Time and Space Complexity Analysis

Algorithm	**Time Complexity (Search)**	**Space Complexity**	**Parameters**
Linear Search	O(n)	O(n)	n = number of elements
Binary Search	O(log n)	O(n)	n = number of elements
Hashing	O(1) (average), O(n) (worst)	O(n)	n = number of elements
Bloom Filter	O(k)	O(m)	n = estimated number of elements m = size of bit array k = number of hash functions
Cuckoo Filter	O(1)	O(n)	n = number of elements

References:

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms*. MIT press.
- "Bloom Filters - Introduction and Implementation." GeeksforGeeks, 28 Nov. 2023, www.geeksforgeeks.org/bloom-filters-introduction-and-implementation/.
- "Cuckoo Filter: Practically Better Than Bloom." Cse.wustl.edu, https://www.cse.wustl.edu/~jain/cse573-14/ftp/mdfg.pdf.

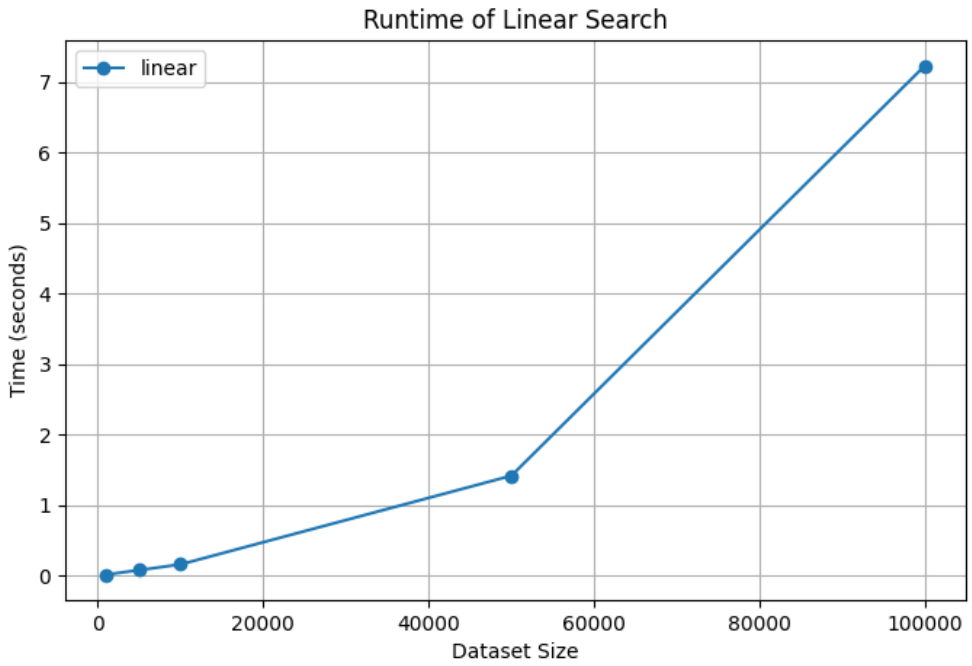
Parameter Explanations:

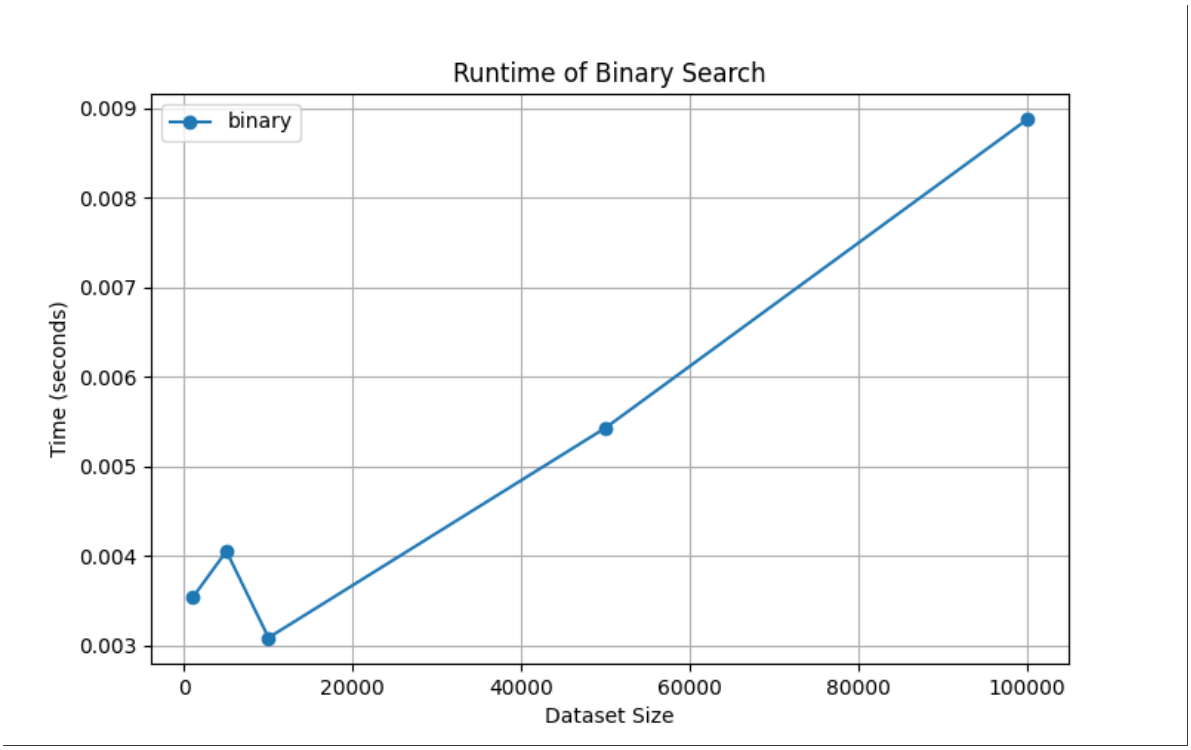
- **n:** The number of elements (logins) stored in the data structure.
- **m:** The size of the bit array in a Bloom filter.
- **k:** The number of hash functions used in a Bloom filter.

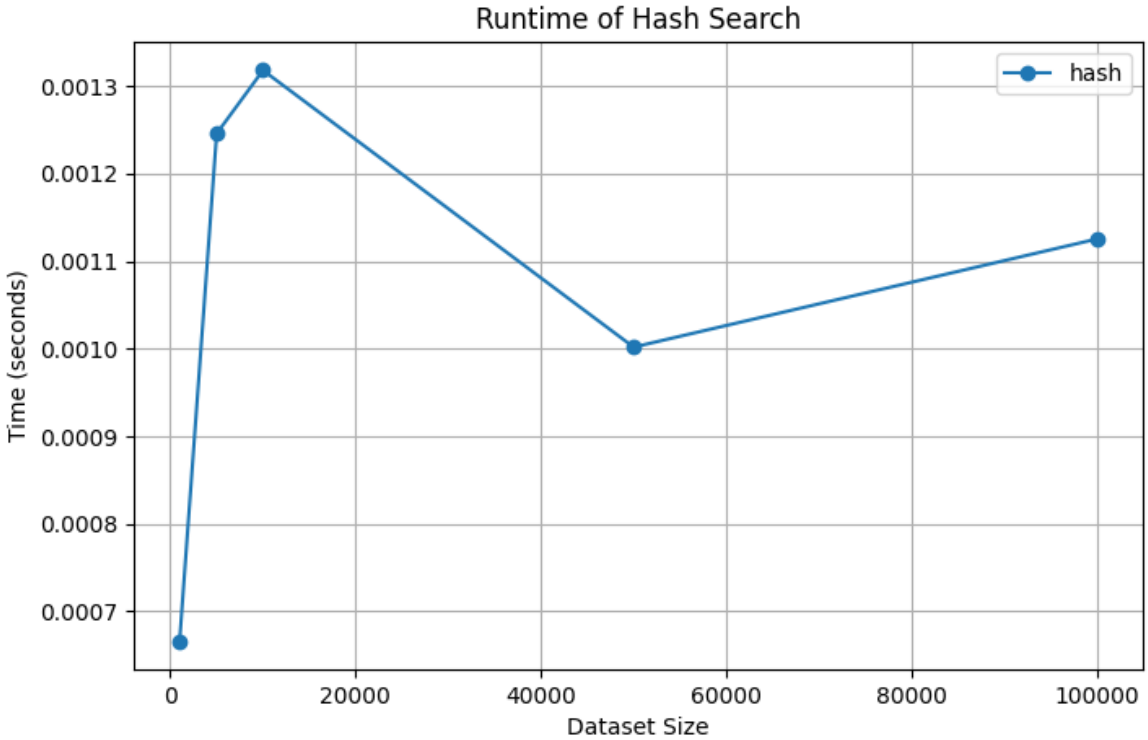
2) Plots showing the run time complexity for large enough data. You can generate strings dataset using synthesized functions. Aim for at least one million data. Then, run your program with different implementations using linear search, binary search, hashing, Bloom filter, Cuckoo filter, and compare their run time in a plot. Include this analysis in your pdf file and explain if it supports your analysis. Remember to upload your dataset on the web and include a link to the dataset in the pdf file.

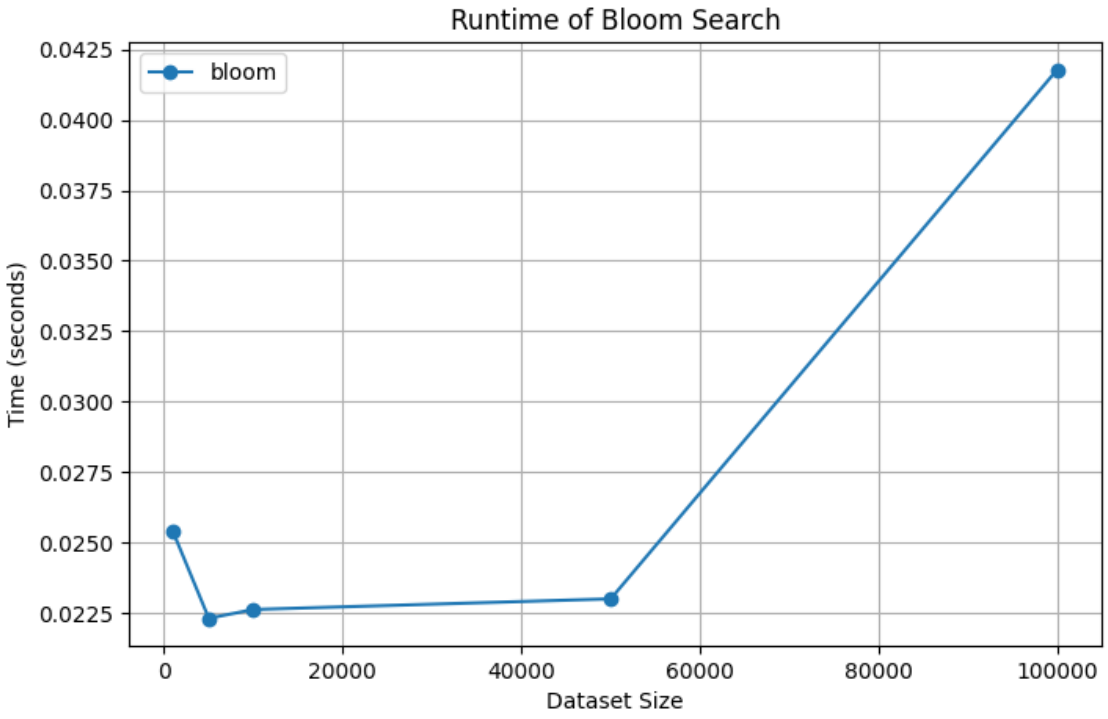
```
Running linear search with dataset size: 1000
run_experiment took 0.0157 seconds
Running linear search with dataset size: 5000
run_experiment took 0.0792 seconds
Running linear search with dataset size: 10000
run_experiment took 0.1602 seconds
Running linear search with dataset size: 50000
run_experiment took 1.4172 seconds
Running linear search with dataset size: 100000
run_experiment took 7.2301 seconds
Running binary search with dataset size: 1000
run_experiment took 0.0035 seconds
Running binary search with dataset size: 5000
run_experiment took 0.0041 seconds
Running binary search with dataset size: 10000
run_experiment took 0.0031 seconds
Running binary search with dataset size: 50000
run_experiment took 0.0054 seconds
Running binary search with dataset size: 100000
run_experiment took 0.0089 seconds
Running hash search with dataset size: 1000
run_experiment took 0.0007 seconds
Running hash search with dataset size: 5000
run_experiment took 0.0012 seconds
Running hash search with dataset size: 10000
run_experiment took 0.0013 seconds
Running hash search with dataset size: 50000
run_experiment took 0.0010 seconds
```

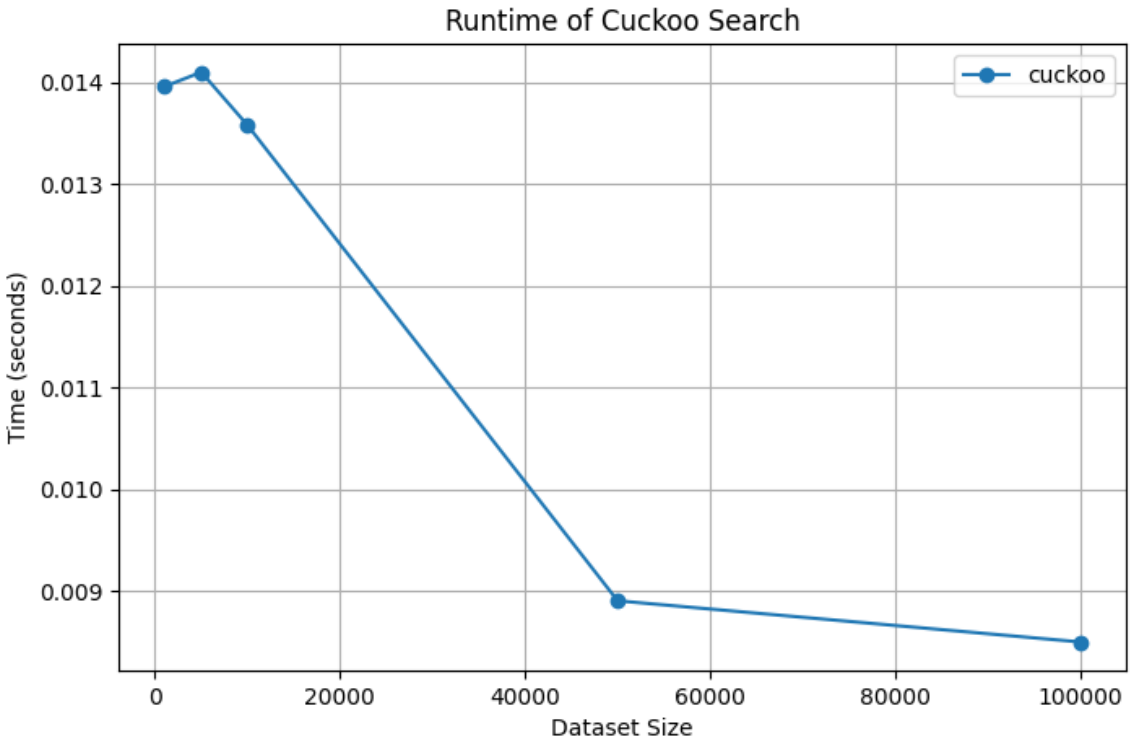
```
Running hash search with dataset size: 100000
run_experiment took 0.0011 seconds
Running bloom search with dataset size: 1000
run_experiment took 0.0254 seconds
Running bloom search with dataset size: 5000
run_experiment took 0.0223 seconds
Running bloom search with dataset size: 10000
run_experiment took 0.0226 seconds
Running bloom search with dataset size: 50000
run_experiment took 0.0230 seconds
Running bloom search with dataset size: 100000
run_experiment took 0.0418 seconds
Running cuckoo search with dataset size: 1000
run_experiment took 0.0140 seconds
Running cuckoo search with dataset size: 5000
run_experiment took 0.0141 seconds
Running cuckoo search with dataset size: 10000
run_experiment took 0.0136 seconds
Running cuckoo search with dataset size: 50000
run_experiment took 0.0089 seconds
Running cuckoo search with dataset size: 100000
run_experiment took 0.0085 seconds
```

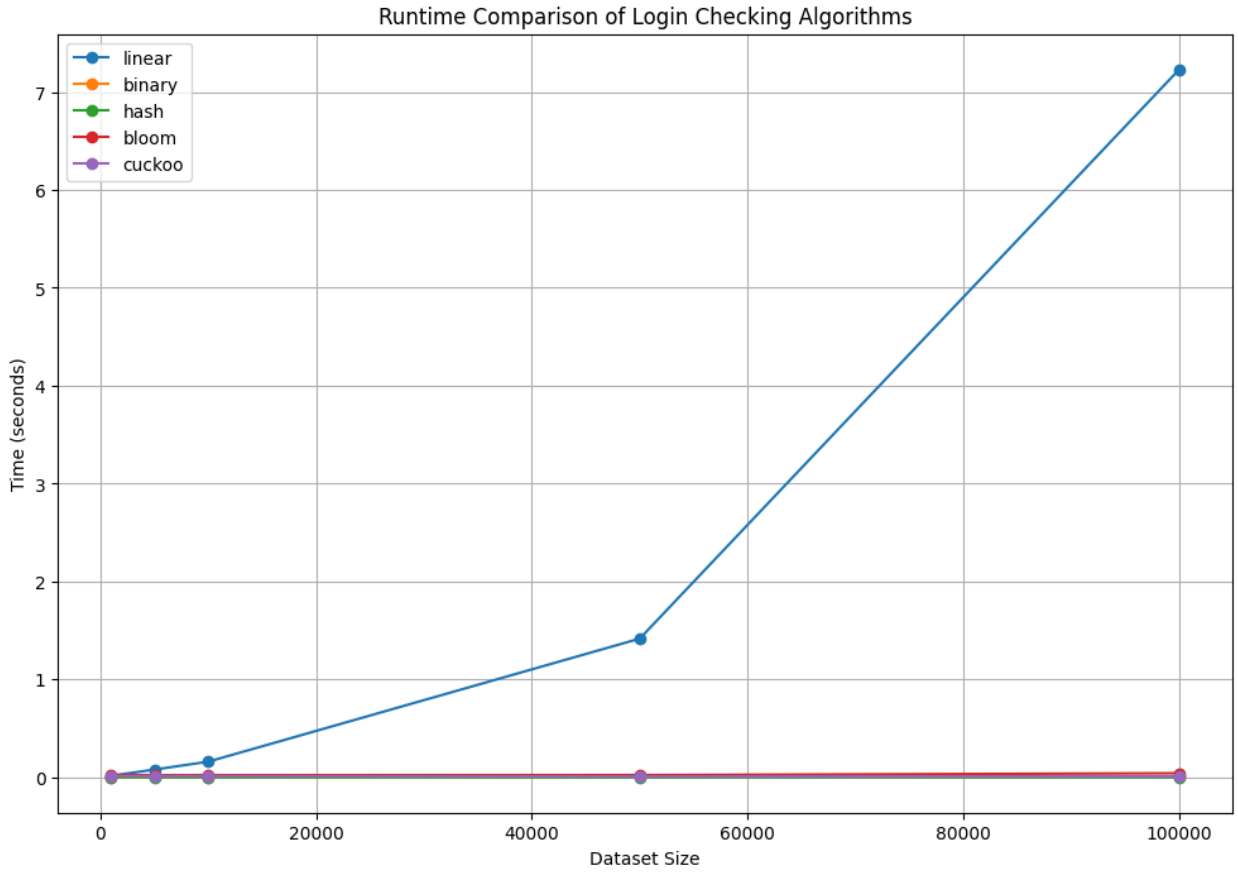



Runtime of Binary Search
binary
Time (seconds)
Dataset Size
0.009
0.008
0.007
0.006
0.005
0.004
0.003
0
20000
40000
60000
80000
100000


Runtime of Hash Search
hash
Time (seconds)
Dataset Size
0.0013
0.0012
0.0011
0.0010
0.0009
0.0008
0.0007
0
20000
40000
60000
80000
100000


Runtime of Bloom Search
bloom
Time (seconds)
0.0425
0.0400
0.0375
0.0350
0.0325
0.0300
0.0275
0.0250
0.0225
0
20000
40000
60000
80000
100000
Dataset Size


Runtime of Cuckoo Search
cuckoo
Time (seconds)
0.014
0.013
0.012
0.011
0.010
0.009
0
20000
40000
60000
80000
100000
Dataset Size



3) Python code comparing the hashing, Bloom filter, and Cuckoo filter. Add the link of your GitHub repository for this assignment to the pdf file.

Github link: https://github.com/MahmoudOsama97/TheLoginCheckerProblem

Pdf link:
https://github.com/MahmoudOsama97/TheLoginCheckerProblem/blob/main/Assignment1.pdf

Folder structure:

```
login-checker/
├── README.md           # Instructions and project description
├── Assignment1.pdf    # Project dependencies
├── Assignment1.ipynb    # colab notebook
├── src/                # Source code
│   ├── dataset.py      # Dataset generation
│   ├── login_checker.py # Implementations of all algorithms
│   ├── main.py         # Main script for running experiments and plotting
│   └── utils.py        # Utility functions
├── workspace/                # Source code
│   ├── username_dataset.txt         # dataset
│   └── .png        # plots for all algorithms
└── tests/              # Unit tests
    └── test.py
```

General Steps to Use the project:

Option1:

1. **Clone the Repository: git clone https://github.com/MahmoudOsama97/TheLoginCheckerProblem.git**
2. **Run the Code: python3 src/main.py**

Option2:

Use colab notebook directly by uploading it to google colab or by running it in VS Code

Note: code is a combination from:

1) **Being manually written**
2) **browsed from github repos and geeksforgeeks site**
3) **used Gemini in some parts to understand the algorithms and have example code.**

Bonus

Alternative Method: Quotient Filter

Data Structure: Quotient Filter

How it Works:

The Quotient filter is a probabilistic data structure that provides a compact way to store and query approximate set membership, similar to Bloom and Cuckoo filters. It's particularly well-suited when you need good performance and a low false positive rate, even with a high number of elements.

Here's a simplified explanation of how it works:

1. **Hashing: Each item (username in our case) is hashed using a hash function, producing a hash value.**
2. **Quotienting:**
    - **The hash value is split into two parts: a *quotient* and a *remainder*.**
    - **The quotient is used to determine the *slot* index in the main table (similar to how you'd use the hash value directly in a regular hash table).**
    - **The remainder is stored in the slot.**
3. **Slot Structure: Each slot in the Quotient filter table can store three things:**
    - **Remainder: A few bits of the original hash value (the remainder part).**
    - **Occupied bit: Indicates whether the slot was ever occupied by an element whose hash value had the corresponding quotient.**
    - **Continuation bit: Indicates whether the current remainder is part of a *run* of remainders that belong to the same quotient (explained below).**
    - **Shifted bit: Indicates that the remainder has been shifted to the right due to a collision (explained below).**
4. **Handling Collisions:**
    - **If two items have the same quotient (i.e., they map to the same slot index), the Quotient filter uses a modified form of linear probing to resolve collisions.**
    - **However, instead of just storing the remainder in the next available slot, it maintains *runs* of remainders with the same quotient.**

- **The continuation bit is used to chain together remainders belonging to the same run.**
- **The shifted bit is used to indicate that a remainder is not in its "canonical slot" (the slot determined by its quotient) because it has been shifted due to collisions.**

5. **Membership Query:**
    - **To check if an item is in the filter, calculate its hash, quotient, and remainder.**
    - **Go to the slot indicated by the quotient.**
    - **If the *occupied bit* of that slot is not set, the item is definitely not in the filter.**
    - **Otherwise, scan the *run* of remainders starting from that slot (following the *continuation bits*) and check if any of the remainders match the item's remainder.**
    - **If a match is found, the item is *probably* in the filter (there's a chance of a false positive).**
    - **If the end of the run is reached without a match, the item is definitely not in the filter.**

Complexity Analysis:

- **Time Complexity:**
    - **Insertion: O(1) on average, but it can degrade to O(log n) in the worst case (when there are long runs due to many collisions).**
    - **Lookup: O(1) on average, similar to insertion, with a worst-case of O(log n).**
- **Space Complexity:**
    - **O(n), where n is the number of elements.**

Advantages of Quotient Filters:

- **Good Performance: Generally provides fast insertion and lookup times (close to constant time on average).**
- **Low False Positive Rate: Achieves a lower false positive rate than Bloom filters for the same amount of space, especially at higher load factors.**
- **Dynamic Resizing: Can be resized dynamically (though resizing is a relatively expensive operation).**

- **Deletions: Unlike Bloom filters, Quotient filters support deletions (though this adds some complexity).**
- **Mergeable: Quotient filters can be merged efficiently, which is useful in distributed settings.**

Disadvantages:

- **More Complex Implementation: More complex to implement than Bloom filters.**
- **Worst-Case Performance: While the average performance is excellent, the worst-case time complexity can be O(log n) if the hash function produces many collisions or if the load factor is very high.**

Reference:

- **Original Paper: Bender, M. A., Farach-Colton, M., Johnson, R., Kraner, R., Kuszmaul, B. C., Medjedovic, D., ... & Zhu, B. (2012). Don't thrash: how to cache your hash on flash. *Proceeding***