Detailed Approach and Runtime Analysis

The program is being developed to manage a URL Caching for a Web Crawler application to efficiently check whether a URL has been previously processed to avoid redundant processing. This is achieved using a simple Bloom filter with the operations:

1. **Add a URL**: Add a URL to the Bloom filter.
2. **Check a URL**: Check if a URL is possibly in the Bloom filter.

Design Considerations

A Bloom filter is chosen as the primary data structure for URL caching due to its efficient space usage and fast lookups:

a. **Space Efficiency:** Requires significantly less memory compared to other set-based data structures (e.g., HashSet) by using fixed-size bit array instead of storing entire URLs.
b. **Constant-Time Operations:** Insertions and lookups run in O(k) time, where k is the number of hash functions (in this case, k=3).
c. **Probabilistic Nature:** While false positives exist, their probability can be minimized by carefully selecting the filter size and number of hash functions.
d. **Scalability:** The implementation is tested for large-scale inputs (100,000+ URLs) to validate performance

1 Classes and Data Structure

1.1 Bloom Filter Class:

The Bloom Filter class initializes the bit array to the given size n. It also efficiently sets all bits to zero in a single step. This is the base data structure class creating the bloom filter.

Attributes	Data Type	Definition
size	integer	Size of the bloom filter bit array
bit_array	bitarray	Keeps the bit position of the URL cached

Methods	Return Type	Definition
_hash1, _hash2, _hash3	Integer	Hash functions to hash the url using custom algorithms.

1.2 Bloom Filter Caching Class:

This class manages the overall URL Caching mechanism for the application using the bloom filter. It acts as a container for the web crawler application for caching methods.

Attributes	Data Type	Definition
bloom	BloomFilter	Object to hold the bloom filter

Methods	Return Type	Definition
Add_url	None	Add bit to bitarray at hash ind
check_url	None	check bit to bitarray at hash ind

2 Functions

2.1 initiateURLCaching ():

The **initiateURLCaching** function reads commands from an input file and processes them sequentially. Each line specifies an operation (e.g., adding a url, check url exists) and its associated data. The function interprets these commands and calls the corresponding method in BloomFilterCaching Class to execute the operation, ensuring that the system works seamlessly.

2.1.1 Steps for initiate url caching

- Opens the input file.
- Reads each command, splitting it into operation (e.g., Add) and data (e.g., https://example.com/page1).
- Matches the operation to a method in BloomFilterCaching – add_url, check_url

2.1.2 Runtime Analysis:

As this function reads the input file line by line and invokes appropriate operations, the complexity depends on the number of commands in the file and the complexity of the invoked operations. For each operation, the traversal cost depends on n.

Time Complexity: O(n)

2.2 calculate_size ():

The **calculate_size** function calculate the size of the Bloom filter bit array from the given number of input elements and defined number of hash functions. It initially checks for the global value of size and returns if non-zero else proceeds with the mathematical calculations.

2.2.1 Steps for calculate size

- Opens the input file.
- Get the count of lines with ADD and call the calculate size function with number of hash functions(k) and input element count(n).
- Perform the calculation, bloom bit array size m = (k * n) / log(2) if predefined bloom size is 0.

2.2.2 Runtime Analysis:

The function performs a few arithmetic operations, which are O(1)
Time Complexity: O(1)

2.3 Add_url():

This method adds a URL to the Bloom filter by computing multiple hash values and setting corresponding bits in the bit array.. It writes a confirmation to the output file once the url is added.

2.3.1 Steps:

- Computes three hash values for the given URL using _hash1, _hash2, and _hash3.
- Sets the bits at the computed indices to 1 in the Bloom filter.
- Writes a message to the output file indicating the URL has been added

2.3.2 Runtime Analysis:

Time Complexity: O(k) ≈ O(1), where k = 3 hash functions.

2.4 Check_url() :

This method checks whether a given URL is possibly present in the Bloom filter. The function passes the URL to the different hash functions and checks the bit value in the bitarray of the Bloom Filter at the calculated hash value for the URL.

2.4.1 Steps:

- Computes three hash values for the URL using _hash1, _hash2, and _hash3.
- Checks if all corresponding bits in the bit array are set to 1.
- Returns True if all bits are set, otherwise returns False (ensuring the URL is definitely not in the filter).
- Writes a message to the output file regarding the check result.

2.4.2 Runtime Analysis:

Time Complexity: O(k) ≈ O(1), due to constant-time bit lookups.

2.4 _hash1():

This method implements a polynomial rolling hash function for hashing a given URL string. It helps distribute URL values uniformly across the bit array.

2.4.1 Steps:

- Iterates over each character in the URL.
- Computes a hash using a prime number (31) and modular arithmetic to avoid overflow.

2.4.2 Runtime Analysis:

Time Complexity: O(m), where m is the length of the URL.

2.5 _hash2():

This method generates a hash value using the **DJB2 algorithm**, which is a simple and efficient hashing function used for fast computations. It offers a good balance between speed and distribution.

2.5.1 Steps:

- Initializes a hash value with 5381.
- Iterates through each character in the URL.
- Uses bitwise shifting and addition (hash * 33 + char) to compute the hash.
- Maps the final hash value within the Bloom filter size using modulus operation

2.5.2 Runtime Analysis:

Time Complexity: O(m), where m is the length of the URL.

2.6 _hash3():

This method generates a hash value using an XOR-based hash function. This function efficiently distributes hash values using bitwise operations for better randomness.

2.6.1 Steps:

- Initializes hash_value as 17 (Prime number).
- Iterates through each character in the URL.
- Uses XOR bitwise operation combined with left and right shifts to mix the bits and distribute hash values.
- Maps the final hash value within the Bloom filter size using modulus operation.

2.6.2 Runtime Analysis:

Time Complexity: O(m), where m is the length of the URL.

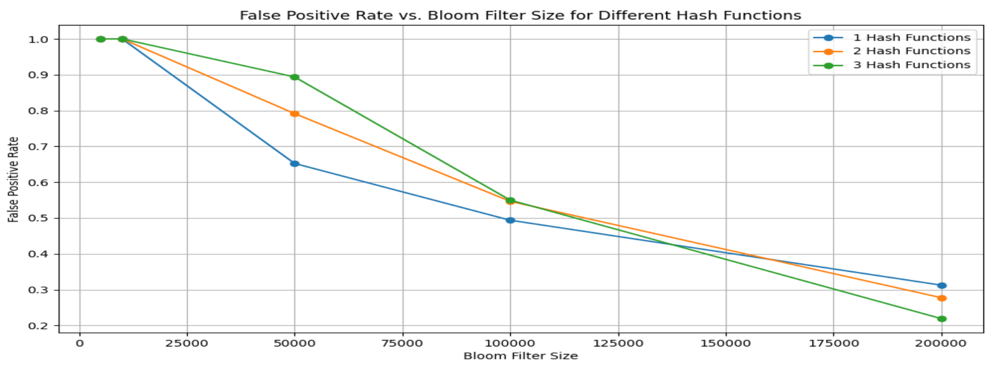
3. Alternative Approach: HashSet Implementation

An alternative approach is using a **HashSet** for caching URLs.
A HashSet is a built-in data structure in Python that stores unique elements in an unordered manner. It provides O(1) time complexity for insertions, deletions, and lookups due to its underlying hash table implementation.
Unlike a Bloom filter, a HashSet does not have false positives but requires **significantly more memory** as it stores entire URLs instead of bit representations.
This makes HashSet a suitable alternative when memory is not a constraint and 100% accuracy is required for membership tests.

Feature	**Bloom Filter**	**HashSet**
Memory Usage	O(m) (bit array)	O(n) (direct storage)
Insertion Time	O(1)	O(1)
Lookup Time	O(1) (with false positives)	O(1) (accurate)
False Positives	Possible	None
Scalability	Excellent (fixed memory)	Higher memory overhead
Deletion Support	Not supported	Fully supported

4. Correlation of Size Bloom Filter and number of Hash Functions:

Larger Bloom Filter Size (m): A larger bit array reduces the false positive rate since there are more available positions to set bits, reducing accidental overlaps.
Number of Hash Functions (k): Increasing the number of hash functions initially reduces false positives. However, beyond an optimal point, too many hash functions start setting too many bits, leading to increased false positives.
Optimal Trade-off: The ideal number of hash functions (k) is approximately (m/n) * ln(2), where m is the Bloom filter size and n is the number of inserted elements.


False Positive Rate vs. Bloom Filter Size for Different Hash Functions
1 Hash Functions
2 Hash Functions
3 Hash Functions
False Positive Rate
Bloom Filter Size