**FNV-1 hash function or I prefer call it RapidXORHash**

**Initial Value (Offset Basis)**

hash value starts with a large constant called the "offset basis." For FNV-1a in the 64-bit version, this value is:

hash=14695981039346656037

This large prime number is carefully chosen to reduce initial collisions and distribute the keys more uniformly across the hash space.

**Hashing Each Byte**

For each byte b(i)​ in the input string key, the algorithm performs the following operations:

#### **XOR Operation**:

The current hash value is XORed with the byte value b(i)

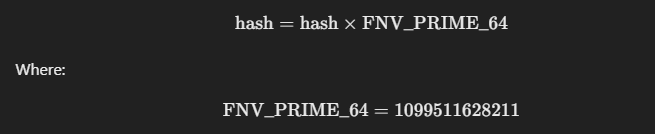


* **XOR (⊕)**: The XOR operation compares corresponding bits of the two operands. If the bits are the same, the result is 0; if they are different, the result is 1.

This operation ensures that even small changes in the input string result in significant changes to the hash value.

#### b. **Multiplication by a Prime Number**:

The resulting hash is then multiplied by a large prime number:



* **Why a Prime Number?**: Prime numbers are used because they help in distributing the hash values uniformly. Multiplication by a prime spreads the entropy (randomness) across the entire range of possible hash values, reducing the likelihood of collisions.

The combination of XOR and prime multiplication ensures that the hash value changes unpredictably with each byte processed. Mathematically, after processing nnn bytes, the hash can be represented as:



### 3. ****Final Hash Value****

After processing all the bytes in the input string, the final hash value is the FNV-1a hash for that string. This value is unique (or nearly unique) for each different input string.

### Optimization Insights

The process of XOR followed by multiplication can be optimized by processing multiple bytes at a time, which reduces the number of operations required.

### Key Mathematical Properties:

1. **Avalanche Effect**: Due to the XOR and prime multiplication, a small change in the input (like flipping a single bit in the string) will cause the final hash to change drastically. This is a desired property in cryptographic hash functions.
2. **Uniform Distribution**: The prime number multiplication helps in spreading out the hash values across the entire possible range, minimizing collisions (i.e., different inputs producing the same hash value).
3. **Efficiency**: The FNV-1a function is computationally efficient. XOR is a simple bitwise operation, and multiplication by a constant is fast on modern processors.

**RapidXORHash Algorithm by Mustafa haboul**

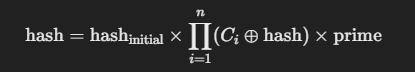
### The RapidXORHash Algorithm

**Mathematical Foundation:**

* **Process in 64-bit Chunks**: Instead of processing each byte individually, process 8 bytes at a time. This reduces the number of iterations by a factor of 8.
* **Final XOR:** After processing all chunks, apply a final XOR operation with a prime number to further mix the hash value.

### New Hash Function Formula:

Given an input string key , split it into 64-bit chunks C1,C2,…,Cn . The hash function becomes:



Hash initial = FNV\_OFFSET\_BASIS\_64

Prime = FNV\_PRIME\_64

Finally, XOR the entire hash value with a final constant to reduce collisions.

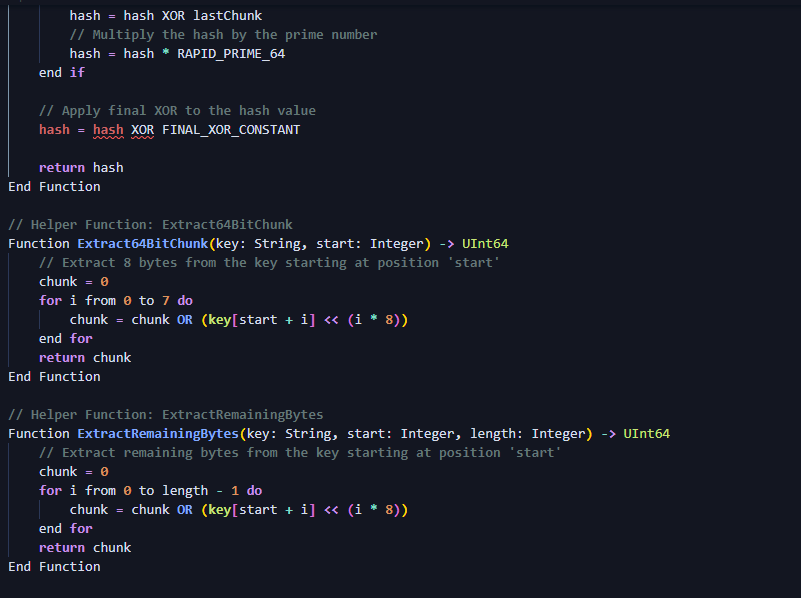
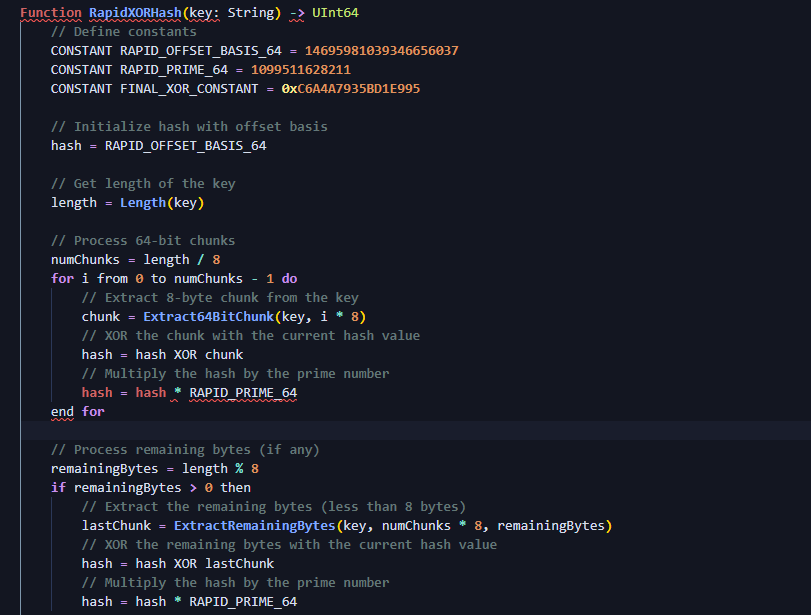
Optimized Code for RapidXORHash

### Explanation:

1. **Chunk Processing:** The main optimization is processing the input string in 64-bit chunks (8 bytes at a time). This approach leverages the CPU's ability to handle 64-bit operations efficiently, significantly reducing the number of iterations required for longer strings.
2. **Final XOR:** The final XOR operation with a constant further scrambles the resulting hash, making it less likely for two similar inputs to produce the same hash value (collisions).
3. **Avoiding Conditionals:** By processing the remainder of the string in one go (outside the main loop), the function avoids unnecessary conditional checks, which improves efficiency.



**Pseudocode for** **[RapidXORHash]**



### Explanation of the Pseudocode:

1. **Initialization:**
   * Start with the predefined offset basis (RAPID\_OFFSET\_BASIS\_64).
2. **64-bit Chunk Processing:**
   * Process the key in chunks of 8 bytes (64 bits). For each chunk, XOR it with the hash and multiply by the prime number (RAPID\_PRIME\_64).
3. **Remaining Bytes:**
   * After processing all full 64-bit chunks, handle any remaining bytes that are less than 8 bytes. XOR these remaining bytes with the hash and multiply by the prime number.
4. **Final XOR:**
   * Apply a final XOR operation (FINAL\_XOR\_CONSTANT) to the hash value to further reduce collisions and ensure a more evenly distributed hash.
5. **Helper Functions:**
   * Extract64BitChunk extracts an 8-byte chunk from the key.
   * ExtractRemainingBytes extracts bytes less than 8 bytes at the end of the key.

This pseudocode illustrates the core idea of the RapidXORHash algorithm, ensuring efficient processing and maintaining the properties of the original hash function while optimizing performance.

### Time Complexity

The time complexity of the RapidXORHash algorithm depends on the length of the input string and how the data is processed. Here's a breakdown:

1. **Processing 64-bit Chunks:**
   * The algorithm processes the input string in chunks of 8 bytes (64 bits). Each chunk requires a fixed amount of work (XOR operation and multiplication by a prime number).
   * If the input string length is N, then there are (n/8) chunks (integer division).
   * Each chunk requires constant time operations, so processing all chunks takes O(n8)=O(n)O(\frac{n}{8}) = O(n/8) = O(n) time.
2. **Handling Remaining Bytes:**
   * After processing full 64-bit chunks, any remaining bytes (less than 8 bytes) are processed in constant time.
   * This step also takes O(1) time.
3. **Final XOR Operation:**
   * The final XOR operation is a constant-time operation, O(1).

**Overall Time Complexity:**

* The total time complexity is O(n), where N is the length of the input string.

### Space Complexity

The space complexity of the RapidXORHash algorithm is determined by the amount of extra memory used beyond the input data.

1. **Hash Variable:**
   * A single uint64\_t variable is used for the hash value.
   * Space required: O(1).
2. **Temporary Variables:**
   * Temporary variables like chunk and lastChunk are used for processing data.
   * Space required: O(1).
3. **Input String:**
   * The input string is not modified, so its space is not considered as additional space used by the algorithm.

**Overall Space Complexity:**

* The total space complexity is O(1), which means the algorithm uses a constant amount of extra space regardless of the input size.

### Summary

* **Time Complexity:** O(n) — The time taken scales linearly with the length of the input string.
* **Space Complexity:** O(1) — The space used is constant and does not depend on the size of the input.

The RapidXORHash algorithm is efficient in both time and space, making it suitable for applications requiring fast and memory-efficient hashing.

My githup repo:

https://github.com/MustcodeQ/RapidXORHash-Algorithm