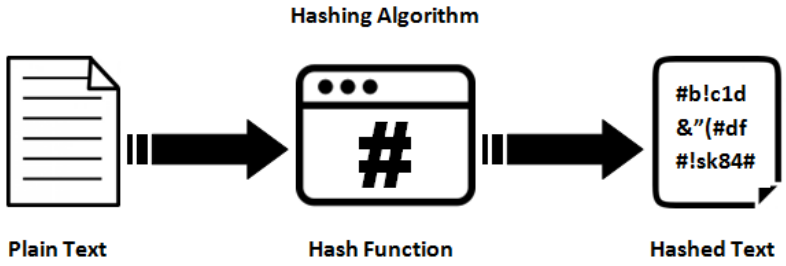
Vladimir Avdeev

SHA-1:

Introduction:

Secure Hashing Algorithm is what SHA stands for and the 1 stands for the first version. Now what is hashing and what is its significance. Hashing allows you to take in an input string and apply a mathematical formula splitting this string up into numerous pieces. Before it was used widely as a security method it was used to index through large amounts of data. A good way to think of this is thinking of a word you want to define in a dictionary book. Now thinking of hashing in a security standpoint we apply difficult formulas making it difficult to reverse the original string.

Example of Hashing:



Hashing takes in a string input then returns a fixed length output. Hashing is a highly effective method of securing data by converting it into a fixed-length, unique string (hash) that cannot be reversed to its original form. It ensures data integrity, as even minor changes to the input produce a completely different hash. Hashing is widely used for password storage, data verification, and digital signatures due to its irreversible and secure nature.

For example SHA-256 will always return a 64 character hash (or string). Hashing allows data to be secure by making it difficult to revert to the original string. In the real world we can commonly see this used as a storage method of passwords. Where only the hash of the password is being stored instead of the raw text of the password. An example on how this works with authentication is that environments will compare hashes instead of the raw text of the password.

History of SHA-1:

SHA-1 was developed by the US government back in 1993 and was a project that was developed by US government standard agencies NIST. Later the 1993 version was superseded and it's now called SHA-0 and was replaced in 1995 with a small change. The only change was a single bitwise rotation in the message schedule. The reason for this was that the original version contained a flaw which affected “cryptographic security”. [8]

SHA-1 was officially removed from major products as the method of hashing data was now deemed insecure. Its successor (SHA-256) is still commonly used, the main difference being that the hash sizes are larger. The biggest worry with smaller hash sizes is collisions which is when two different inputs create the same hash.

SHA-1 was widely used for many applications and protocols before it was superseded. It was used widely for TLS, SSL,PGP,SSH,S/MIME, and IPsec. This method of hashing was required by law for certain US government applications. This hashing algorithm played a key role in the security infrastructure and still is with its successors.

Intuition:

Hashing Functions:

I decided to conduct research on two different hashing algorithms, one allowing us to go over the concept in a little bit more simpler perspective. And the other being more sophisticated giving more of a real world example of how hashing really works.

DJB2:

The DJB2 hash function is a great example of hashing, where raw text is converted into a numeric representation that cannot be reversed. It initializes a hash value to 5381 and processes each character by shifting the hash 5 bits to the left (effectively multiplying by 32), adding the original hash (to make it a multiplication by 33), and then adding the ASCII value of the character. This simple formula efficiently produces a unique (but not cryptographic) representation of the input text.[6]

Bitwise Operations:

Before we begin explaining the steps of how SHA-1 works I think it would be useful to go over some bitwise operations. SHA-1 uses bitwise AND, OR, XOR, NOT, and circular shift operations. Which are used to mix bits, maintain fixed block sizes, and utilize non-linear transformations. [9]

AND : Returns true if both conditions are true.

OR : Returns true if atleast one condition is true.

NOT: Negates the value. True becomes false vice versa.

XOR : Returns true if exactly one condition is true.

Left shift : Shifts bits to the left by filling the vacant bits with 0’s.

AND operation	OR operation	NOT operation	XOR operation	Left shift

1 & 1 = 1 1 & 0 = 0 0 & 0 = 0	1 | 1 = 1 1 | 0 = 1 0 | 0 = 0	~ 1 = 0 ~ 0 = 1	1 ^ 1 = 0 1 ^ 0 = 1 0 ^ 0 = 0	1111 << 2 = 111100

SHA-1:

Sha-1 first works by taking in a string input then begins to pad the message. Padding the message is “to ensure that length is congruent to 448 modulo 512”[5]. What is padding? Padding is the processing step where the input message is suitable for an algorithm to process the input. Allowing the algorithm to divide the message into 512 bit blocks. Then to “divide into 16 words of 32 bits” [5]. Then these words will be “expanded into 80 32-bit words”[5]. After they are split into 80 words then SHA-1 uses five constant values which are applied to a specific range of rounds (0-79). After all rounds the values are all added into their original hash values in order to produce the final hash.

Steps of How SHA-1 Works:

1. First the message needs padding the message by adding 0’s until it's a multiple of 512 (or 64 bytes). Then append the original length of the message at the end.
2. Then we will divide the padded message into 16 32 bit “words”
3. Extend the 16 32 bit words to 80 32 bit words. First we are just initializing the 16 words from the message block. By extending these words we will run XOR operations on words before the word we are iterating on and then rotate the bits left by 1 bit.

   ```
   W[i] = RotateLeft( W[i - 3] XOR W[i - 8] XOR W[i - 14] XOR
   W[i - 16], 1)
   ```

4. We set up the five working variables with the given five constant values; A=H0 ,B=H1, C=H2, D=H3, and E=H4. We are cycling through 80 rounds applying different operations

to each word. For this we have a round function ‘F’ which determines how intermediate hash values B,C,D are combined each round. As well we have a different constant value of ‘K’ which is applied differently to each group of 20 rounds.

Values of K ;

Rounds 0-19: K=0x5A827999

Rounds 20-39: K=0x6ED9EBA1

Rounds 40-59: K=0x8F1BBCDC

Rounds 60-79: K=0xCA62C1D6

Function of F:

Rounds 0-19 : `F = (B & C) | ((~B) & D)`

Rounds 20-39: `F = B ^ C ^ D`

Rounds 40-59: `F = (B & C) | (B & D) | (C & D)`

Rounds 60-79: `F = B ^ C ^ D`

Key: & = and , | = or , ~ = not, ^ = XOR

For each round we store these values into a temp value. F adds the logic and complexity and K introduces the round specific constants. By combining this with words (word[iteration]) this ensures that hash computation is resistant to prediction.

```
TEMP = RotateLeft( A, 5) + F + E + W[iteration] + K

E = D , D = C , C =  RotateLeft( B, 30), B = A, A = TEMP
```

5. After 80 rounds we get the final values for A,B,C,D,E. We now update our constant the five constant values of H by adding the final values. H0 += A ,H1 += B, H2 += C, H3 += D, and H4 += E. After we add to all the blocks H0-H4 contain all the final hash values.

To create the final hash we concatenate them into one 160-bit single value. Concatenate means joining each binary value end to end.

Pseudocode:

DJB2:

```
FUNCTION djb2Hash(inputString):
    hash = 5381 // Initial seed value
    FOR each character c IN inputString:
        hash = (hash << 5) + hash + c // Equivalent to hash * 33 + c
 RETURN hash
```

Input

inputString: A string of characters to be hashed. Each character will be processed using its ASCII value.

Output

hash: An integer hash value, which is the result of applying the **djb2 hash algorithm** to the input string.

Algorithm Description

1. **Initialize the Hash**:
    - The hash is initialized to a fixed seed value 5381.
2. **Iterate Over the Characters**:
    - For each character c in the input string:
        - Compute the ASCII value of c.

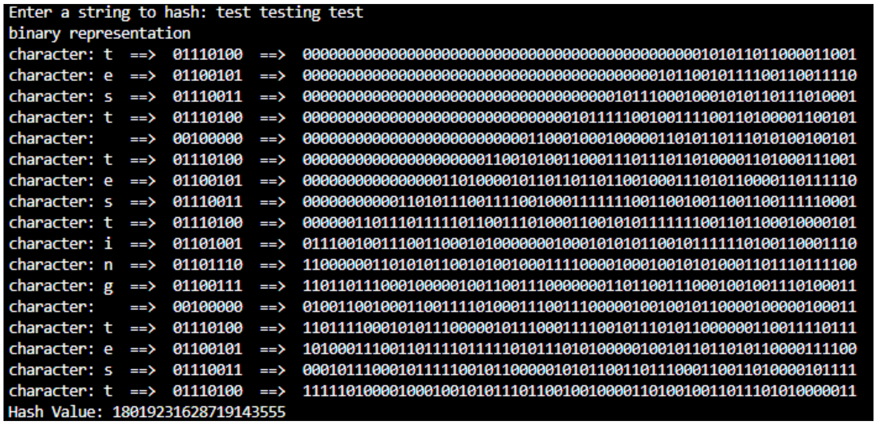
        - hash = (hash << 5) + hash + c

3. **Return the Hash**

Argument for Correctness

1. **Efficient Hashing:**
    - The algorithm uses a fixed multiplier (33) that has been found to provide good distribution for hash values with minimal collisions.

Example of execution:



```
Enter a string to hash: test testing test
binary representation
character: t  ==>  01110100  ==>  0000000000000000000000000000000000000000000000101011011000011001
character: e  ==>  01100101  ==>  0000000000000000000000000000000000000000010110010111100110011110
character: s  ==>  01110011  ==>  0000000000000000000000000000000000001011100010001010110111010001
character: t  ==>  01110100  ==>  0000000000000000000000000000000101111100100111100110100001100101
character:    ==>  00100000  ==>  0000000000000000000000000011000100010000011010110111010100100101
character: t  ==>  01110100  ==>  0000000000000000000001100101001100011101110110100001101000111001
character: e  ==>  01100101  ==>  0000000000000000110100001011011011011001000111010110000110111110
character: s  ==>  01110011  ==>  0000000000011010111001111001000111111100110010011001100111110001
character: t  ==>  01110100  ==>  0000001101110111110110011101000110010101111111001101100010000101
character: i  ==>  01101001  ==>  0111001001110011000101000000010001010101100101111110100110001110
character: n  ==>  01101110  ==>  1100000011010101100101001000111100001000100101010001101110111100
character: g  ==>  01100111  ==>  1101101110001000001001100111000000011011001110001001001110100011
character:    ==>  00100000  ==>  0100110010001100111101000111001110000010010010110000100000100011
character: t  ==>  01110100  ==>  1101111000101011100000101110001111001011101011000000110011110111
character: e  ==>  01100101  ==>  1010001110011011110111110101110101000001001011011010110000111100
character: s  ==>  01110011  ==>  0001011100010111110010110000010101100110111000110011010000101111
character: t  ==>  01110100  ==>  1111101000010001001010111011001001000011010010011011101010000011
Hash Value: 18019231628719143555
```

SHA-1:

padded_message:

```
func padded_message(message):
    padded_message = []  // Initialize an empty list for the padded message

    // Append the original message to the padded message
    padded_message += message

    // Append a single '1' bit (0x80 in hexadecimal is 10000000 in binary)
    padded_message += 0x80

    // Add zero bytes (0x00) until the message length is 64 bits shy of a multiple of 512
    while ((size(padded_message) * 8) % 512 != 448):
        padded_message += 0x00

    // Append the original message length as a 64-bit big-endian integer
    original_length = size(message) * 8  // Calculate the length of the original message in
bits
    for i from 7 to 0:  // Iterate from 7 to 0 for the 8 bytes of the 64-bit integer
        padded_message += (original_length >> (i * 8)) & 0xFF  // Extract each byte

    return padded_message
```

Input

- message: A list of bytes (or an equivalent iterable), representing the original message to be padded. Each element of the message should have a value in the range [0, 255].

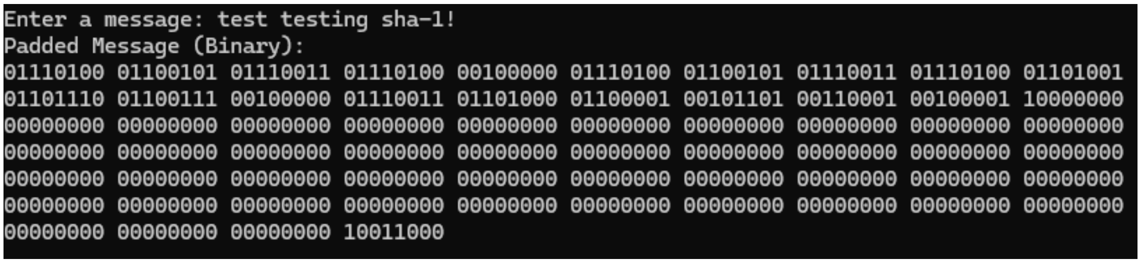
Output

- padded_message: A list of bytes that includes:
    1. The original message.
    2. A single 1 bit followed by zero bits to make the length 64 bits shy of a multiple of 512.
    3. The original message length encoded as a 64-bit big-endian integer.

Algorithm Description

1. **Initialize the Padded Message**:
   - Start with an empty list (padded_message) and copy the original message into it.
2. **Append the '1' Bit**:
   - Append the hexadecimal value 0x80 to signify the binary sequence 10000000 (a single '1' bit followed by seven '0' bits). This marks the end of the message.
3. **Add Padding Bytes**:
   - Use a while loop to append 0x00 bytes (binary 00000000) to the message until the total bit-length of padded_message is 64 bits shy of a multiple of 512.
4. **Encode the Original Message Length**:
   - Calculate the original message length in bits (size(message) * 8).
   - Split this 64-bit value into 8 bytes (big-endian order) by iterating from the most significant byte to the least significant byte and appending them to padded_message.
5. **Return the Padded Message**

Example of execution:



```
Enter a message: test testing sha-1!
Padded Message (Binary):
01110100 01100101 01110011 01110100 00100000 01110100 01100101 01110011 01110100 01101001
01101110 01100111 00100000 01110011 01101000 01100001 00101101 00110001 00100001 10000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 10011000
```

parse_message:

```
func parse_message(padded_message):
    blocks = []  // Initialize an empty list to store 32-bit words

    // Iterate over the padded_message in chunks of 4 bytes (32 bits)
    for i from 0 to size(padded_message) - 1 step 4:
        block = 0  // Initialize the block as 0

        // Combine 4 consecutive bytes into a single 32-bit word
        for j from 0 to 3:
            block = (block << 8) | padded_message[i + j]

        blocks += block  // Add the 32-bit word to the list of blocks

    return blocks
```

Detailed Description

Input

- padded_message: A list of bytes (or equivalent iterable) representing a padded message. Each element in the list should be an integer in the range [0, 255].

Output

- blocks: A list of 32-bit words. Each word is an integer formed by combining 4 consecutive bytes (32 bits) from the input padded_message.

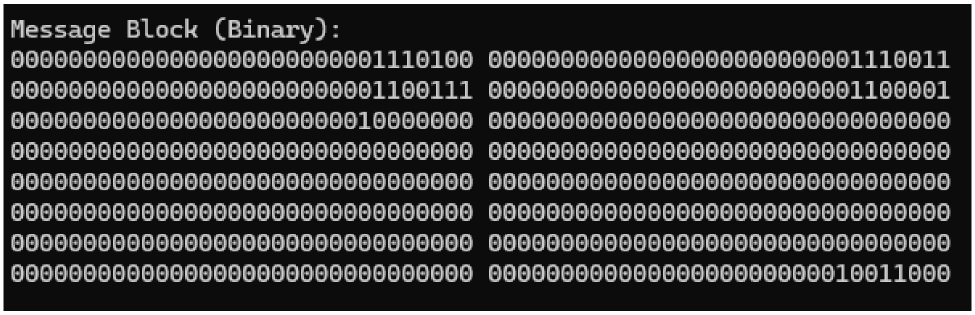
Algorithm Description

1. **Initialize the Blocks List**:
    - Create an empty list block to store the 32-bit words that will be extracted from

the padded_message.

2. **Iterate Over the Padded Message**:
    - Use a loop to process the padded_message in chunks of 4 bytes (32 bits) at a time.
    - For each iteration, the loop starts at i and processes bytes at indices i, i+1, i+2, and i+3.
3. **Combine 4 Bytes Into a 32-Bit Word**:
    - For each chunk of 4 bytes:
        - Start with block = 0.
        - Use a nested loop to process each byte in the chunk:
            - Shift block left by 8 bits (block << 8) to make space for the next byte.
            - Combine the next byte into block using the bitwise OR (|) operation.
4. **Append the 32-Bit Word to Blocks**:
    - Once the 4 bytes have been combined into a 32-bit word, append the resulting block to the blocks list.
5. **Return the List of Blocks**

Example of Execution:



```
Message Block (Binary):
00000000000000000000000001110100 00000000000000000000000001110011
00000000000000000000000001100111 00000000000000000000000001100001
00000000000000000000000010000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000010011000
```

leftRotate:

```
func leftRotate(value , shift):
    return (value << shift) | (value >> (32 - shift))
```

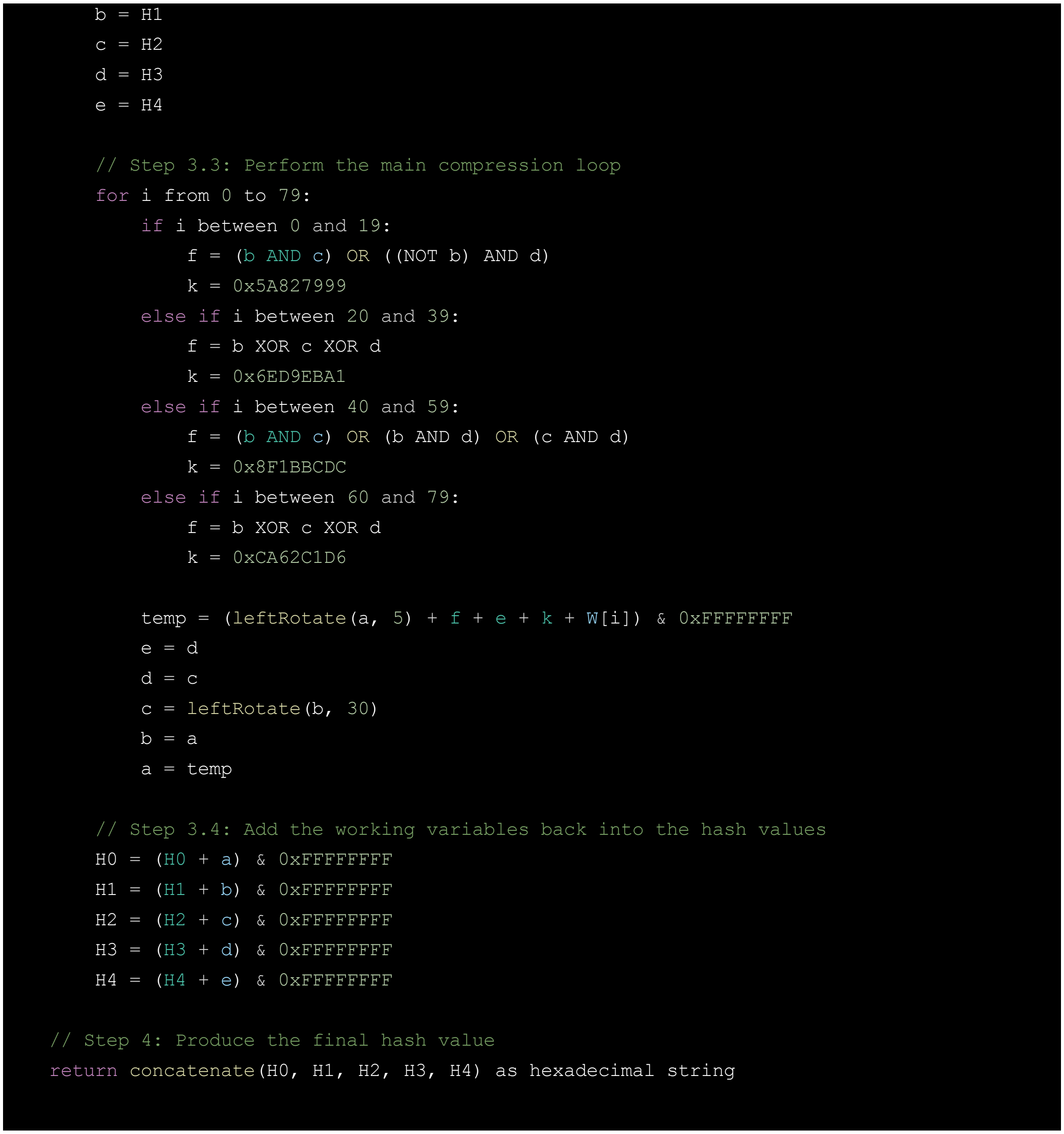
sha1:

```
func sha1(message):
    // Step 1: Preprocessing
    padded_message = pad_message(message)
    blocks = parse_message(padded_message)

    // Step 2: Initialize hash values (H0, H1, H2, H3, H4)
    H0 = 0x67452301
    H1 = 0xEFCDAB89
    H2 = 0x98BADCFE
    H3 = 0x10325476
    H4 = 0xC3D2E1F0

    // Step 3: Process each 512-bit block
    for block in blocks:
        // Step 3.1: Prepare the message schedule (W)
        W = array[80]
        for i from 0 to 15:
            W[i] = block[i]  // First 16 words come from the block
        for i from 16 to 79:
            W[i] = leftRotate(W[i-3] XOR W[i-8] XOR W[i-14] XOR W[i-16], 1)

        // Step 3.2: Initialize working variables (a, b, c, d, e)
        a = H0
```



```
  b = H1
  c = H2
  d = H3
  e = H4

  // Step 3.3: Perform the main compression loop
  for i from 0 to 79:
    if i between 0 and 19:
      f = (b AND c) OR ((NOT b) AND d)
      k = 0x5A827999
    else if i between 20 and 39:
      f = b XOR c XOR d
      k = 0x6ED9EBA1
    else if i between 40 and 59:
      f = (b AND c) OR (b AND d) OR (c AND d)
      k = 0x8F1BBCDC
    else if i between 60 and 79:
      f = b XOR c XOR d
      k = 0xCA62C1D6

    temp = (leftRotate(a, 5) + f + e + k + W[i]) & 0xFFFFFFFF
    e = d
    d = c
    c = leftRotate(b, 30)
    b = a
    a = temp

  // Step 3.4: Add the working variables back into the hash values
  H0 = (H0 + a) & 0xFFFFFFFF
  H1 = (H1 + b) & 0xFFFFFFFF
  H2 = (H2 + c) & 0xFFFFFFFF
  H3 = (H3 + d) & 0xFFFFFFFF
  H4 = (H4 + e) & 0xFFFFFFFF

// Step 4: Produce the final hash value
return concatenate(H0, H1, H2, H3, H4) as hexadecimal string
```

Detailed Description

Input

- message: A sequence of bytes representing the input message to be hashed.

Output

- A 160-bit SHA-1 hash value, represented as the concatenation of five 32-bit integers .

Algorithm Description

1. **Preprocessing**:
    - **Padding**: Pad the message to ensure its length (in bits) is a multiple of 512.
    - **Parsing**: Split the padded message into 16 32-bit blocks.
2. **Initialization**:
    - Set the initial hash values (H0, H1, H2, H3, H4) as constants defined by the SHA-1 standard.
3. **Process Each Block**:
    - **Prepare the Message Schedule**:
        - Create an array W of size 80.
            - Process was explained in depth in the intuition.
    - **Initialize Working Variables**:
        - Set a = H0, b = H1, c = H2, d = H3, e = H4.
    - **Main Loop**:

        - For each of the 80 rounds (i = 0 to 79):
            - Compute the function f and constant k based on the round index
                - The process of computing values on each round is explained in the intuition.

4. **Produce the Final Hash**:
    - Concatenate H0, H1, H2, H3, H4 to produce the final 160-bit hash.

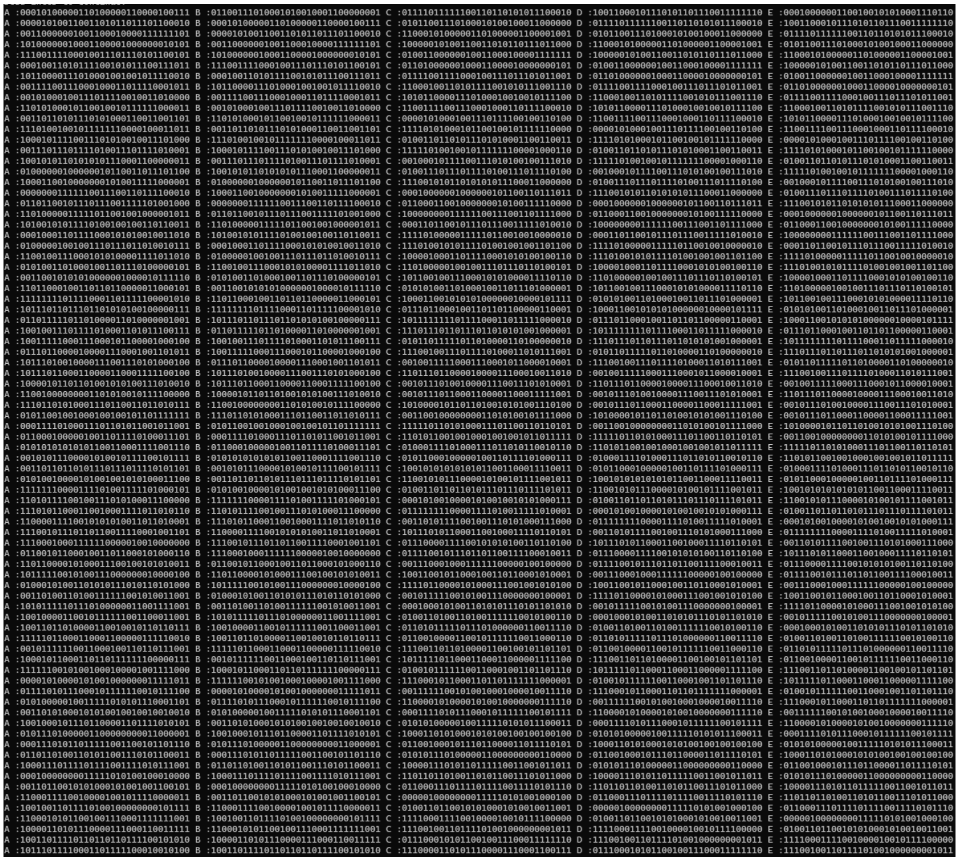
Argument for Correctness

1. **Adheres to the SHA-1 Standard**:
    - The algorithm follows the structure and steps defined by the SHA-1 cryptographic hash standard, including padding, message scheduling, and the main processing loop.
2. **Handles Any Input**:
    - The preprocessing step ensures the input is correctly padded, making it compatible with the SHA-1 block size requirements.
3. **Secure Message Scheduling**:
    - The schedule ensures that each word in the block contributes to the final hash, even for long inputs, by using XOR and left rotation.
4. **Iterative Hashing**:
    - Each block updates the hash values incrementally, preserving all information from previous blocks.
5. **Comparing Hashes:**
    - If we run SHA-1 algorithm on two different occasions with the same input

string. We should in theory get the same output hash as we are performing the

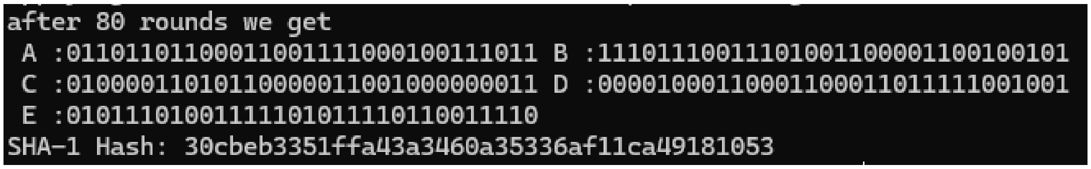
○ same operations on both strings.

Example of execution of 80 rounds of operations performed on values A,B,C,D E:



```
A :00010100000110100000110000100111 B :01100111010001010010001100000001 C :01111011111100110110101011100010 D :10011000101110101101110011111110 E :00010000001100100101010001110110
A :00001010011001101011011101100010 B :00010100000110100000110000100111 C :01011001110100010100100011000000 D :01111011111100110110101011100010 E :10011000101110101101110011111110
A :00110000000100110001000011111101 B :00001010011001101011011101100010 C :11000101000001101000001100001001 D :01011001110100010100100011000000 E :01111011111100110110101011100010
A :10100000001000110000100000010101 B :00110000000100110001000011111101 C :10000010100110011010110111011000 D :11000101000001101000001100001001 E :01011001110100010100100011000000
A :11100111100010011101110101100101 B :10100000001000110000100000010101 C :01001100000001001100010000111111 D :10000010100110011010110111011000 E :11000101000001101000001100001001
A :00010011010111100101011100111011 B :11100111100010011101110101100101 C :01101000000010001100001000000101 D :01001100000001001100010000111111 E :10000010100110011010110111011000
A :10110000111010001001001011110010 B :00010011010111100101011100111011 C :01111001111000100111011101011001 D :01101000000010001100001000000101 E :01001100000001001100010000111111
A :00111100111000100011011110001011 B :10110000111010001001001011110010 C :11000100110101111001010111001110 D :01111001111000100111011101011001 E :01101000000010001100001000000101
A :00101000100111011110010011010000 B :00111100111000100011011110001011 C :10101100001110100010010010111100 D :11000100110101111001010111001110 E :01111001111000100111011101011001
A :11010100010110010010111111000011 B :00101000100111011110010011010000 C :11001111001110001000110111100010 D :10101100001110100010010010111100 E :11000100110101111001010111001110
A :00110110101110101000110011001101 B :11010100010110010010111111000011 C :00001010001001110111100100110100 D :11001111001110001000110111100010 E :10101100001110100010010010111100
A :11101001001011111110000100011011 B :00110110101110101000110011001101 C :11110101000101100100101111110000 D :00001010001001110111100100110100 E :11001111001110001000110111100010
A :10001011110011101010010011101000 B :11101001001011111110000100011011 C :01001101101011101010001100110011 D :11110101000101100100101111110000 E :00001010001001110111100100110100
A :00111011101111010011101111010001 B :10001011110011101010010011101000 C :11111010010010111111100001000110 D :01001101101011101010001100110011 E :11110101000101100100101111110000
A :10010101101010101110001100000011 B :00111011101111010011101111010001 C :00100010111100111010100100111010 D :11111010010010111111100001000110 E :01001101101011101010001100110011
A :01000000100000010110011011101100 B :10010101101010101110001100000011 C :01001110111011110100111011110100 D :00100010111100111010100100111010 E :11111010010010111111100001000110
A :10001100100000001010011111000001 B :01000000100000010110011011101100 C :11100101011010101011100011000000 D :01001110111011110100111011110100 E :00100010111100111010100100111010
A :00000001111110011100110111100010 B :10001100100000001010011111000001 C :00010000001000000101100110111011 D :11100101011010101011100011000000 E :01001110111011110100111011110100
A :01101100101110111001111101001000 B :00000001111110011100110111100010 C :01100011001000000010100111110000 D :00010000001000000101100110111011 E :11100101011010101011100011000000
A :11010000011111011001001000001011 B :01101100101110111001111101001000 C :10000000011111100111001101111000 D :01100011001000000010100111110000 E :00010000001000000101100110111011
A :10100101011110100100100110110011 B :11010000011111011001001000001011 C :00011011001011101110011111010010 D :10000000011111100111001101111000 E :01100011001000000010100111110000
A :00010001101111000101010010011010 B :10100101011110100100100110110011 C :11110100000111110110010010000010 D :00011011001011101110011111010010 E :10000000011111100111001101111000
A :01000001001001110111011010010111 B :00010001101111000101010010011010 C :11101001010111101001001001101100 D :11110100000111110110010010000010 E :00011011001011101110011111010010
A :11001001110001010100001111011010 B :01000001001001110111011010010111 C :10000100011011110001010100100110 D :11101001010111101001001001101100 E :11110100000111110110010010000010
A :01010011010001001101110100000101 B :11001001110001010100001111011010 C :11010000010010011101110110100101 D :10000100011011110001010100100110 E :11101001010111101001001001101100
A :00110010101010000001000010111110 B :01010011010001001101110100000101 C :10110010011100010101000011110110 D :11010000010010011101110110100101 E :10000100011011110001010100100110
A :11011000100110110110000011000101 B :00110010101010000001000010111110 C :01010100110100010011011101000001 D :10110010011100010101000011110110 E :11010000010010011101110110100101
A :11111111011110001101111100001010 B :11011000100110110110000011000101 C :10001100101010100000010000101111 D :01010100110100010011011101000001 E :10110010011100010101000011110110
A :10111011011101101010100100000111 B :11111111011110001101111100001010 C :01110110001001101101100000110001 D :10001100101010100000010000101111 E :01010100110100010011011101000001
A :01101111101101000011010000001001 B :10111011011101101010100100000111 C :10111111110111100011011111000010 D :01110110001001101101100000110001 E :10001100101010100000010000101111
A :10010011101111010001101011100111 B :01101111101101000011010000001001 C :11101110110111011010101001000001 D :10111111110111100011011111000010 E :01110110001001101101100000110001
A :10011111000111000101100001000100 B :10010011101111010001101011100111 C :01011011111011010000110100000010 D :11101110110111011010101001000001 E :10111111110111100011011111000010
A :01110110000100001110001001101011 B :10011111000111000101100001000100 C :11100100111011110100011010111001 D :01011011111011010000110100000010 E :11101110110111011010101001000001
A :10111010010000111001110101000100 B :01110110000100001110001001101011 C :00100111110001110001011000010001 D :11100100111011110100011010111001 E :01011011111011010000110100000010
A :10111011000110000110001111100100 B :10111010010000111001110101000100 C :11011101100001000011100010011010 D :00100111110001110001011000010001 E :11100100111011110100011010111001
A :10000101101101001010100111010010 B :10111011000110000110001111100100 C :00101110100100001110011101010001 D :11011101100001000011100010011010 E :00100111110001110001011000010001
A :11001000000001101010010111100000 B :10000101101101001010100111010010 C :00101110110001100001100011111001 D :00101110100100001110011101010001 E :11011101100001000011100010011010
A :11101101010001110110011011010111 B :11001000000001101010010111100000 C :10100001011011010010101001110100 D :00101110110001100001100011111001 E :00101110100100001110011101010001
A :01011001001000100100101101111111 B :11101101010001110110011011010111 C :00110010000000011010100101111000 D :10100001011011010010101001110100 E :00101110110001100001100011111001
A :00011110100011101101011001011001 B :01011001001000100100101101111111 C :11111011010100011101100110110101 D :00110010000000011010100101111000 E :10100001011011010010101001110100
A :01100010000010011011110100011101 B :00011110100011101101011001011001 C :11010110010010001001001011011111 D :11111011010100011101100110110101 E :00110010000000011010100101111000
A :01010101010101100110001111001110 B :01100010000010011011110100011101 C :01000111101000111011010110010110 D :11010110010010001001001011011111 E :11111011010100011101100110110101
A :00101011100001010010111100101111 B :01010101010101100110001111001110 C :01011000100000100110111101000111 D :01000111101000111011010110010110 E :11010110010010001001001011011111
A :00110110110101110111011110101101 B :00101011100001010010111100101111 C :10010101010101011001100011110011 D :01011000100000100110111101000111 E :01000111101000111011010110010110
A :01010010000101001001010100011100 B :00110110110101110111011110101101 C :11001010111000010100101111001011 D :10010101010101011001100011110011 E :01011000100000100110111101000111
A :11111100001111010011111101000101 B :01010010000101001001010100011100 C :01001101101101011101110111101011 D :11001010111000010100101111001011 E :10010101010101011001100011110011
A :11010111100100111010100011100000 B :11111100001111010011111101000101 C :00010100100001010010010101000111 D :01001101101101011101110111101011 E :11001010111000010100101111001011
A :11101011000110010001111011010110 B :11010111100100111010100011100000 C :01111111000011110100111111010001 D :00010100100001010010010101000111 E :01001101101101011101110111101011
A :11000011110010101010011011010001 B :11101011000110010001111011010110 C :00110101111001001110101000111000 D :01111111000011110100111111010001 E :00010100100001010010010101000111
A :11100101110110110011110001001101 B :11000011110010101010011011010001 C :10111010110001100100011110110101 D :00110101111001001110101000111000 E :01111111000011110100111111010001
A :11100010001111110000010010000000 B :11100101110110110011110001001101 C :01110000111100101010100110110100 D :10111010110001100100011110110101 E :00110101111001001110101000111000
A :01100101100010011011000101000110 B :11100010001111110000010010000000 C :01111001011101101100111100010011 D :01110000111100101010100110110100 E :10111010110001100100011110110101
A :11011000010100011100100101010011 B :01100101100010011011000101000110 C :00111000100011111100000100100000 D :01111001011101101100111100010011 E :01110000111100101010100110110100
A :10111110010100111000000010000100 B :11011000010100011100100101010011 C :10011001011000100110110001010001 D :00111000100011111100000100100000 E :01111001011101101100111100010011
A :01000101001101010111010110101000 B :10111110010100111000000010000100 C :11110110000101000111001001010100 D :10011001011000100110110001010001 E :00111000100011111100000100100000
A :00110100110100111111001010011001 B :01000101001101010111010110101000 C :00101111100101001110000000100001 D :11110110000101000111001001010100 E :10011001011000100110110001010001
A :10101111101110100000011001111001 B :00110100110100111111001010011001 C :00010001010011010101110101101010 D :00101111100101001110000000100001 E :11110110000101000111001001010100
A :10010000110010111111001100011001 B :10101111101110100000011001111001 C :01001101001101001111110010100110 D :00010001010011010101110101101010 E :00101111100101001110000000100001
A :10011011010000110010010110110111 B :10010000110010111111001100011001 C :01101011111011101000000110011110 D :01001101001101001111110010100110 E :00010001010011010101110101101010
A :11111011000110001100000111110010 B :10011011010000110010010110110111 C :01100100001100101111110011000110 D :01101011111011101000000110011110 E :01001101001101001111110010100110
A :00101111110011000100110110111001 B :11111011000110001100000111110010 C :11100110110100001100100101101101 D :01100100001100101111110011000110 E :01101011111011101000000110011110
A :10001011000110110111111100000111 B :00101111110011000100110110111001 C :10111110110001100011000001111100 D :11100110110100001100100101101101 E :01100100001100101111110011000110
A :11111100101001000100001001111000 B :10001011000110110111111100000111 C :01001011111100110001001101101110 D :10111110110001100011000001111100 E :11100110110100001100100101101101
A :00001010000101001000000011111011 B :11111100101001000100001001111000 C :11100010110001101101111111000001 D :01001011111100110001001101101110 E :10111110110001100011000001111100
A :01111010111000101111110010111100 B :00001010000101001000000011111011 C :00111111001010010001000010011110 D :11100010110001101101111111000001 E :01001011111100110001001101101110
A :01010000010011111010101110001101 B :01111010111000101111110010111100 C :11000010100001010010000000111110 D :00111111001010010001000010011110 E :11100010110001101101111111000001
A :00110101000101010010010010010010 B :01010000010011111010101110001101 C :00011110101110001011111100101111 D :11000010100001010010000000111110 E :00111111001010010001000010011110
A :10010001011101100001101111010101 B :00110101000101010010010010010010 C :01010100000100111110101011100011 D :00011110101110001011111100101111 E :11000010100001010010000000111110
A :01011101000001100000000011000001 B :10010001011101100001101111010101 C :10001101010001010100100100100100 D :01010100000100111110101011100011 E :00011110101110001011111100101111
A :00011101011011111001100101101110 B :01011101000001100000000011000001 C :01100100010111011000011011110101 D :10001101010001010100100100100100 E :01010100000100111110101011100011
A :01101101001101011001110101100011 B :00011101011011111001100101101110 C :01010111010000011000000000110000 D :01100100010111011000011011110101 E :10001101010001010100100100100100
A :10001110111101111001111010111001 B :01101101001101011001110101100011 C :10000111010110111110011001011011 D :01010111010000011000000000110000 E :01100100010111011000011011110101
A :00010000000011111010100100010000 B :10001110111101111001111010111001 C :11011011010011010110011101011000 D :10000111010110111110011001011011 E :01010111010000011000000000110000
A :00110110010101000101001001100101 B :00010000000011111010100100010000 C :01100011101111011110011110101110 D :11011011010011010110011101011000 E :10000111010110111110011001011011
A :11000111100100001001011110000011 B :00110110010101000101001001100101 C :00000100000000111110101001000100 D :01100011101111011110011110101110 E :11011011010011010110011101011000
A :10010011011110100100000000101111 B :11000111100100001001011110000011 C :01001101100101010001010010011001 D :00000100000000111110101001000100 E :01100011101111011110011110101110
A :11000101011001001110001111111001 B :10010011011110100100000000101111 C :11110001111001000010010111100000 D :01001101100101010001010010011001 E :00000100000000111110101001000100
A :10000110101110000111000110011111 B :11000101011001001110001111111001 C :11100100110111101001000000001011 D :11110001111001000010010111100000 E :01001101100101010001010010011001
A :10011011110110110110111100101010 B :10000110101110000111000110011111 C :01110001010110010011100011111110 D :11100100110111101001000000001011 E :11110001111001000010010111100000
A :10111011110001101111100010010100 B :10011011110110110110111100101010 C :11100001101011100001110001100111 D :01110001010110010011100011111110 E :11100100110111101001000000001011
```

Example of execution after all 80 rounds and we are concatenating the final values of



```
after 80 rounds we get
 A :01101101100011001111000100111011 B :11101110011101001100001100100101
 C :01000011010110000011001000000011 D :00001000110001100011011111001001
 E :01011101001111101011110110011110
SHA-1 Hash: 30cbeb3351ffa43a3460a35336af11ca49181053
```

Asymptotic Runtime Analysis:

DJB2:

Runtime: **O(n)**

This is because we only have one loop that is iterating through the size of the string.

SHA-1:

Runtime: **O(n)**

For the three main functions that are used in this algorithm we are iterating through a given size and the main function iterates to the size of the input string. The *padMessage* function has a while loop that iterates until the size of the array is a multiple of 512. Then *parseMessage* iterates to the given size of the padded message made by the function *padMessage.* The main function *SHA-1* will iterate to the size of the parsed message.

Sources:

1. https://tresorit.com/blog/the-history-of-encryption-the-roots-of-modern-day-cyber-security/#:~:text=The%20first%20encryption%20was%20the,with%20the%20sender%20and%20receiver.
2. https://spectrum.ieee.org/hans-peter-luhn-and-the-birth-of-the-hashing-algorithm
3. https://www.clickssl.net/blog/difference-between-hashing-vs-encryption#:~:text=Hashing%20vs%20Encryption%20%E2%80%93%20Hashing%20refers,the%20reach%20of%20third%20parties.
4. https://en.wikipedia.org/wiki/SHA-1
5. https://www.geeksforgeeks.org/sha-1-hash-in-java/
6. http://www.cse.yorku.ca/~oz/hash.html
7. https://chatgpt.com/
8. https://en.wikipedia.org/wiki/SHA-1#:~:text=SHA%2D1%20was%20developed%20as,Institute%20of%20Standards%20and%20Technology).
9. https://www.geeksforgeeks.org/bitwise-operators-in-c-cpp/