# Report for SHA-2 256bits

To check the code of our implementation [Click Here](https://github.com/B1N4RY-P4R45173/pure-python-implementation-of-sha/tree/main/sha%202)

## Code explanation:

In this function we are taking the ascii message as a string and returning a binary string for each character as an array of binary strings for the whole message

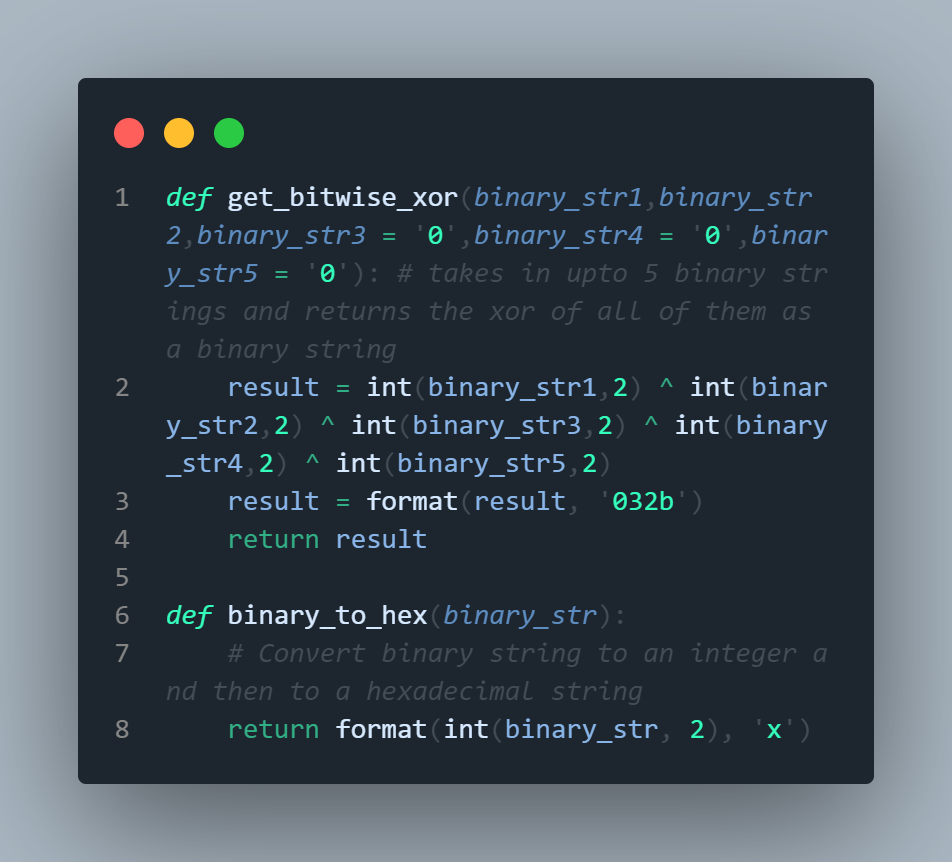
In this function we are taking the above binary strings array and concatenating all of them to get a binary string of the whole message

This function is used for printing the binary string with spaces after every user chosen number of characters. This was only used for debugging and isn’t used anywhere in code.

 This code takes the binary string of the whole message and pads it in the following way:

First it adds 1 at the end of the string and then adds 0s until the length of string is 448 modulo 512

It then concatenates it with number of characters in original message in the form of 64 bits (so no matter how small the input is it will also have padding 0s (accounting to 448 bits) and then 64-bit representation of the input which finally adds up to 512 bits). Therefore, after this step every message is a multiple of 512 bits.

These are just helper functions which were later used by other functions.

The first get a xor of up to 4 binary strings that are provided and the second function converts a binary string to hexadecimal representation

In this function we check if the argument is a string, if yes, we convert it into decimal number and add it to other arguments that are passes and find its total in modulo 2^32, and then we convert the result back to binary which is expressed in 32 bits

These two functions are used for internal calculations. Where sigma0(σ0) is given by right rotation of 7,18 and right shift by 3 and then finding the xor of these 3 values.

Sigma1(σ1) is given by right rotation of 17,19 and right shift of 10 and then finding the xor of these 3 values. Here we have used the above defined function get\_bitwise\_xor

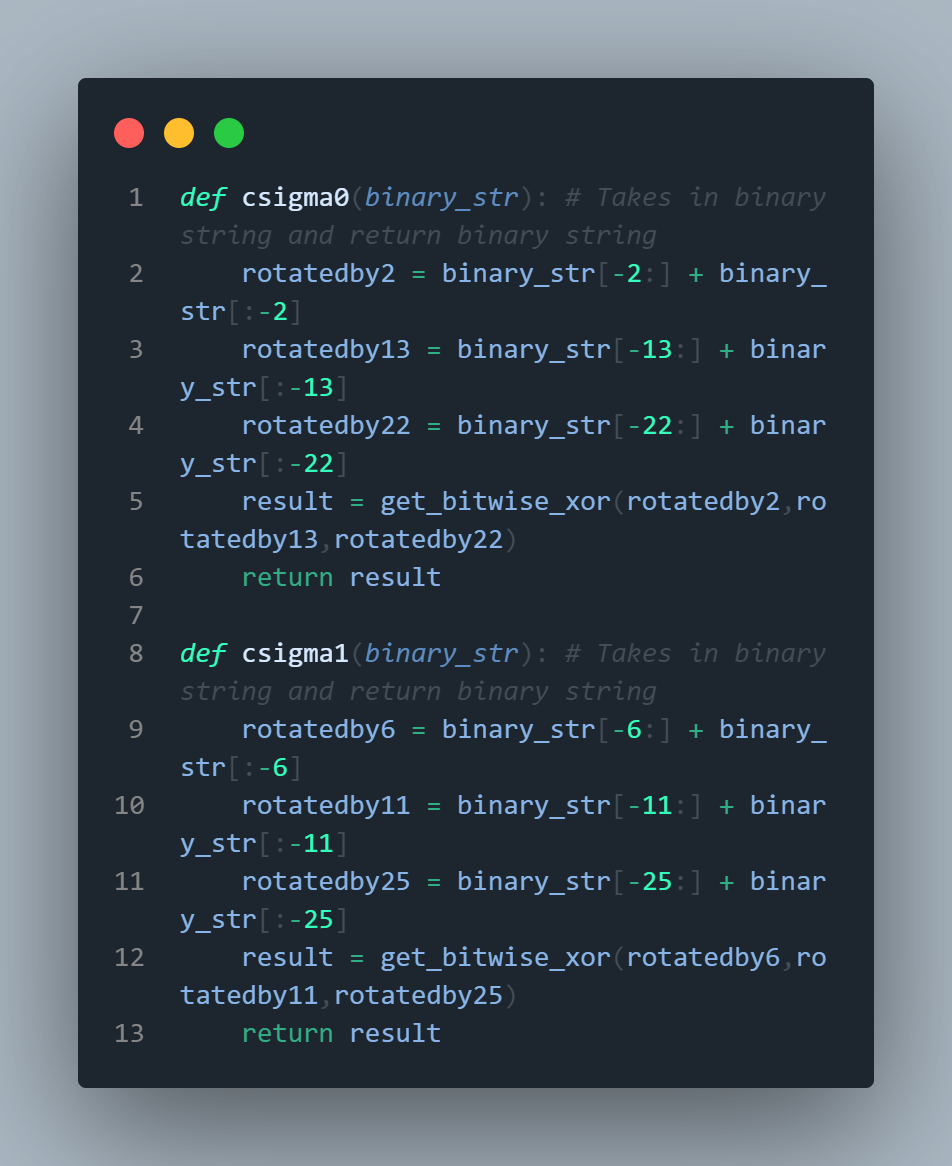
This function returns an array of all W values. Which are calculated as following:

For first 16 values (from w0 to w15) slice every 32 bits from the 512 padded input binary string.

The next values are calculated by the formula:

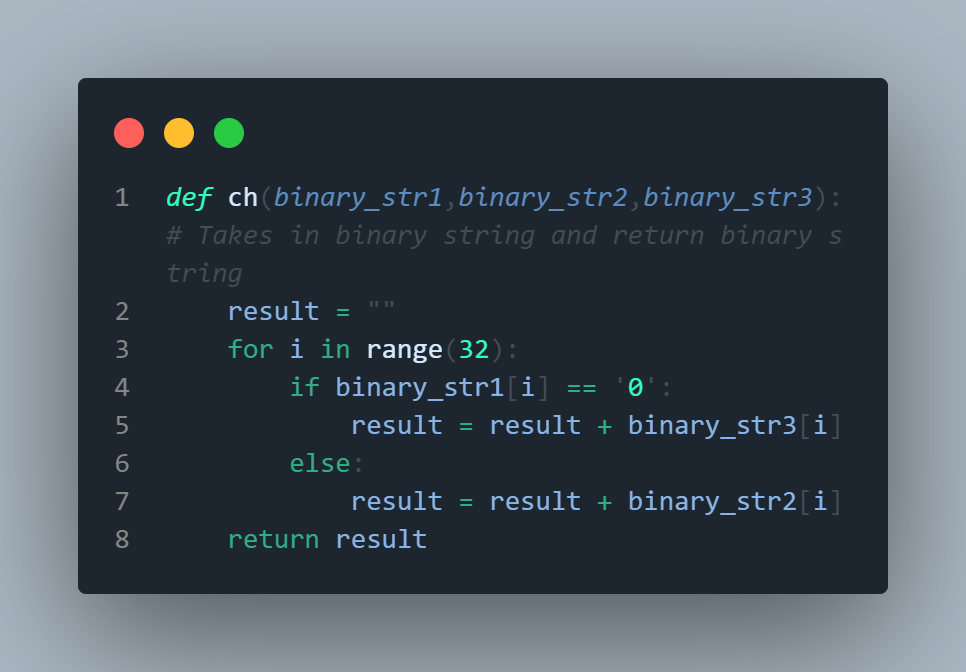
Where t corresponds to current value of W

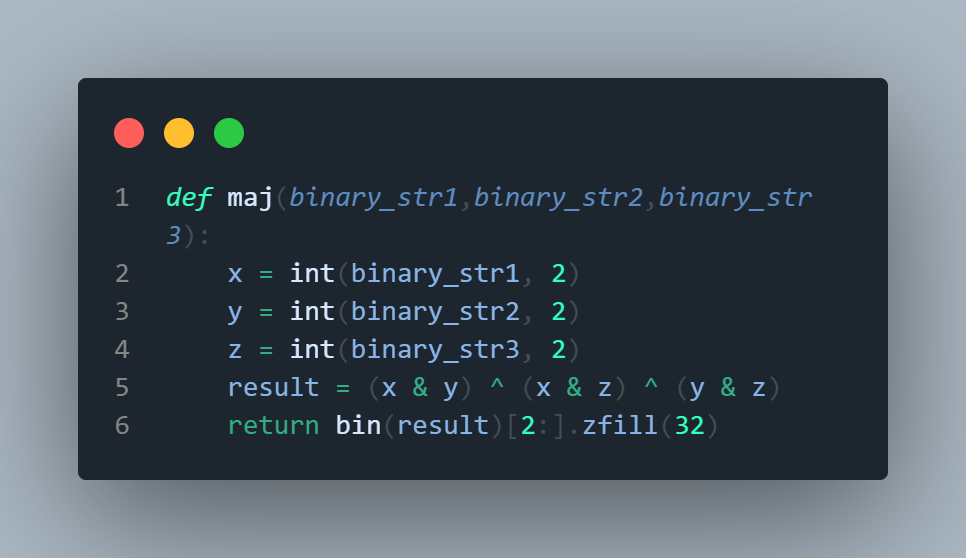


These functions help us in calculating Σ0 and Σ1 of a binary string. It is calculated as follows:

Σ0 = rotate binary string to right by 2,13,22 and xor all these values

Σ1 = rotate binary string to right by 6,11,25 and xor all these values

In this function we take 3 binary strings and iterate through the 1st binary string. If the value is 1 we take a character at that position from binary string 2 or if the value is 0 we take a character at that position from binary string 3

This function also takes three binary strings and compares the value at every position, and checks which value is most repeated from these 3 binary strings, then it writes the most repeated one to a new string called result and returns it.

**The next function is broken down into 4 parts for ease of explanation.**

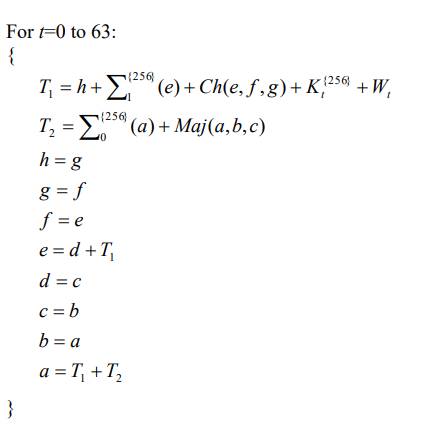


In this function we initialize 8 working variables by taking a hex value from array H and converting it to binary in form of 32 bits.

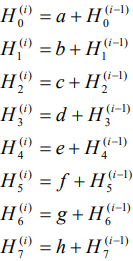
The formula to calculate values in array H is given later.

We initialize and array called K which is the hex representation of fractional part of the first 64 prime numbers. This is to make sure there are no backdoors in the algorithm

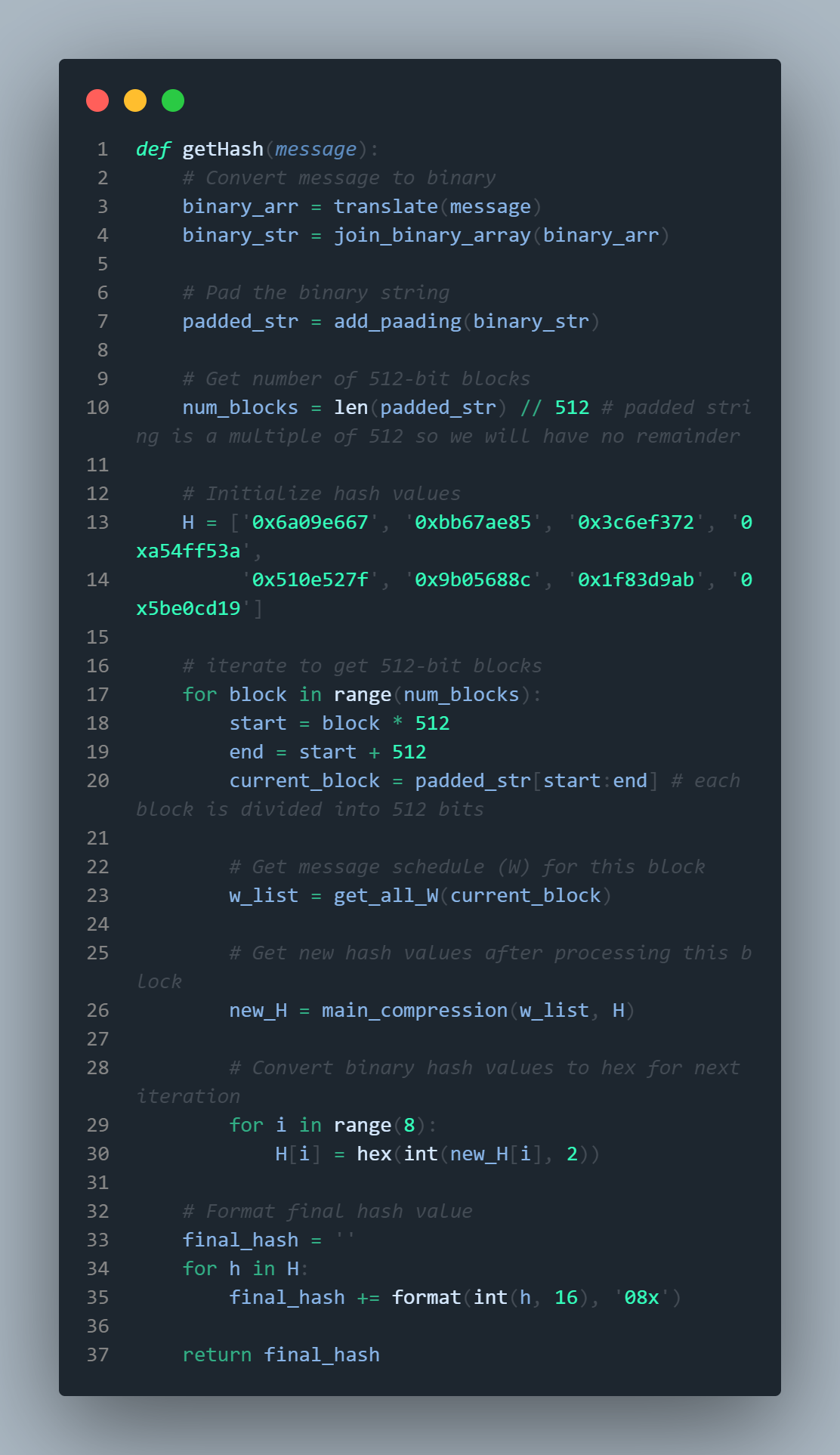
Convert the values of K at every index from hex to binary in form of 32 bits.

For the Main compression loop, we loop for 64 setting the following values each time.

This creates a new array called new\_H which gets the final values of a,b,c,d,e,f,g,h as:

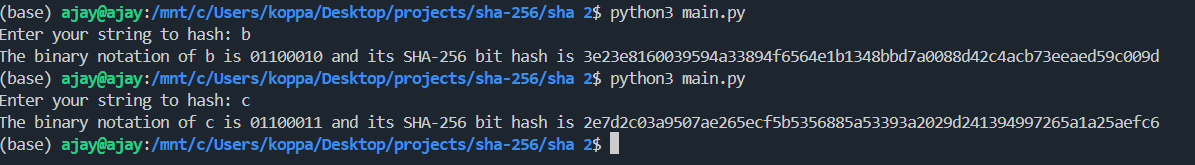


**All of this works perfectly when the input is less the 512 bits, but what if we have more?? Then we need to divide the input into blocks of size 512 and iterate through all of the blocks**

This is the final function which is used to get the sha 256-bit hash. Here we take the message as input and convert it into binary array then to binary string. Then we pad it to make an exact multiple of 512 in bits. Then we divide the total length by 512 to get the no. of blocks. Then we initialize array H by taking square root of first 8 primes and converting their fractional part to hex.

Then we iterate over as many blocks as there are slicing each block into 512 bits and getting all values for current block and pass it to main\_compression function. Then we convert the binary values from new\_H back to hex. And then we concatenate all these values where each hex is exactly 8 characters and return it as final output.

## Avalanche effect:

The avalanche effect is when a small change in the input, like flipping a single bit, causes a significant change in the output. This is a desirable property because it makes it harder for cryptanalysts to predict the input based on the output.

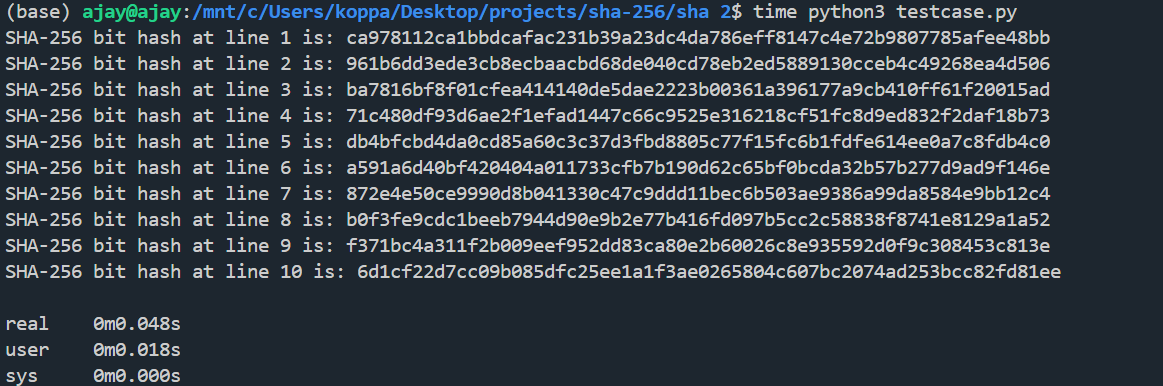
As you can see the binary representation of b is 01100010 and that of c is 01100011. Only the last bit has changed. But there is a massive change in the whole hash that is produced which shows an avalanche effect

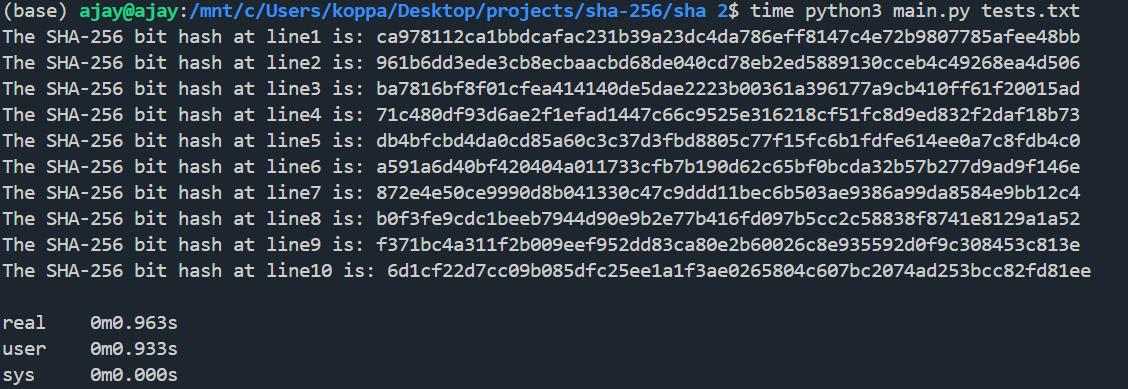
## Efficiency Comparison:

We have compared the efficiency of our implementation with the standard hashlib library in python. For testing we have included the following:

1. Passing a filename, the hash is calculated for every line in the file
2. Trying multiple outputs taken from user input with varying spaces, lengths, characters and order of characters.
3. Trying extremely long messages (as long as a 100,000 characters)

Results:

Our implementation proved to be very accurate achieving 100% accuracy for every input. As you can see below  
  
Time taken by hashlib to hash 10 lines from the tests.txt file is 0.048 seconds  


Whereas time taken by our implementation for hashing the same tests.txt is 0.963 seconds which is slightly less than the double of time taken by hashlib  
  
The main reasons for this are:

1. Hashlib is an industry standard library which directly uses low level instructions and operates on bytes for faster computation, where as our implementation converts it into binary strings and has to convert back forth to integers and strings for computation which wastes a lot of time.
2. Our implementation depends on sys library to get the filename as argument, which could make it slow but for hashlib we have hardcoded the file parameters.

**Note: The tests.txt file is available in our GitHub repo for you to test our results**

## How to Use the SHA-256 Implementation:

**Option 1: Direct File Copy**

1. Download the sha256.py file from the repository.
2. Place the file in the same folder as your Python script.
3. Add the following line to your Python script:

‘’’python

import sha256

‘’’

**Option 2: Clone the Repository**

1. Open a terminal or command prompt.
2. Clone the repository by running:

‘’’bash

git clone <https://github.com/B1N4RY-P4R45173/pure-python-implementation-of-sha.git> && cd pure-python-implementation-of-sha/sha2

‘’’

**Using the SHA-256 Implementation**

1. Import the module in your Python code:

‘’’python

import sha256

‘’’

1. To compute the SHA-256 hash for a string, call the getHash function with the string as an argument:

‘’’python

hashed\_value = sha256.getHash("Your string here")

print(hashed\_value)

‘’’

Replace "Your string here" with the input you want to hash.

