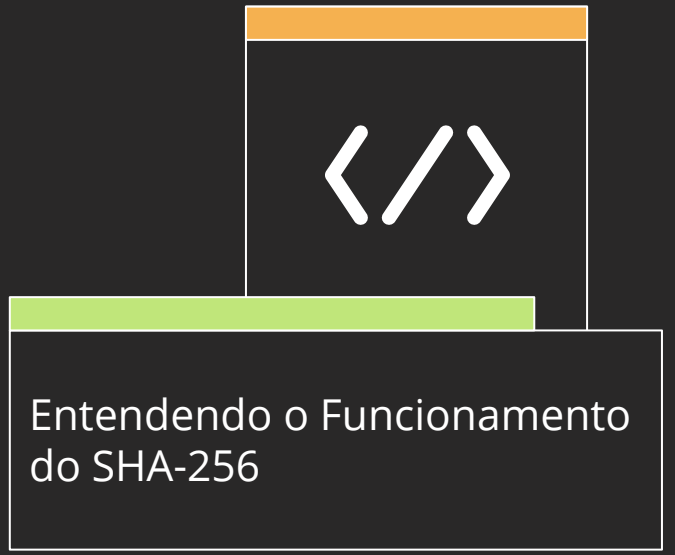
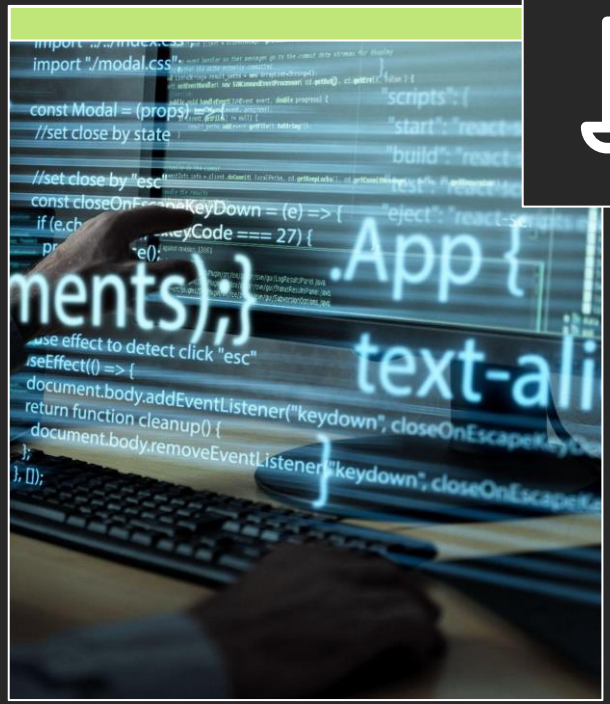
Entendendo o SHA-256





01

SHA-256

1 1 0 0 1 0 1 0 1 0 1 0 1 0 1 1 0

1 0 1 0 1 0 1 1 1 0 0 0 1 1 1 0

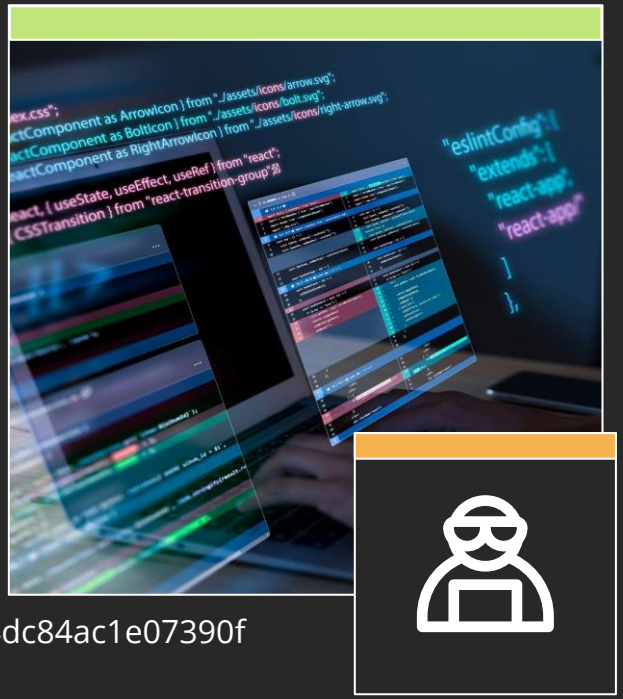
Introdução ao SHA-256

- **O SHA-256** é um algoritmo de hash criptográfica que produz hash de 256 bits, representado tipicamente por uma sequência hexadecimal de 64 caracteres.

- É amplamente utilizado para garantir a integridade dos dados e a segurança das informações.

- Exemplo: considerando a frase “OpenAI é incrível!”. Seu hash SHA-256 seria:

b039f6283139e3e13ac5174982f21b7cf1aa95d3e89c8d30b4dc84ac1e07390f



Como o SHA-256 funciona

1 0 0 1 0 1 0 1 0 0 1 0 1 0 1 0

O SHA-256 processa a entrada em blocos de 512 bits.

Cada bloco passa por uma série de operações matemáticas para produzir um hash de 256 bits.

Mesmo uma pequena alteração na entrada resultará em um hash completamente diferente.

1 0 1 0 0 1 0 0 1 0 1 1 0 1 1 1

Preparando os Dados

1 0 1 0 1 0 1 0 1 1 1 0 0 0 1 1 1 0

1 1 0 0 1 0 1 0 1 0 1 0 1 0 1 1 1 0

Preparação

Antes de aplicar o algoritmo SHA-256, os dados de entrada precisam ser preparados. Isso geralmente envolve a conversão dos dados para uma sequência de bytes.

Exemplo

Vamos converter a frase "OpenAI é incrível!" em bytes usando a codificação UTF-8:

4F 70 65 6E 41 49 20 E9 20 69 6E 63 72 69 76 65 6C 21

Preenchimento de Dados

1 0 1 0 1 0 1 1 1 0 0 0 1 1 1 0

Preenchimento

Os dados de entrada são preenchidos para garantir que o tamanho final seja um múltiplo de 512 bits (64 bytes). Isso é necessário para atender aos requisitos do algoritmo SHA-256.

Exemplo

Se os dados não forem múltiplos de 512 bits, bits de preenchimento são adicionados até que o tamanho seja alcançado.

1 1 0 0 1 0 1 0 1 0 1 0 1 0 1 1 1 0

Divisão em Blocos de 512 Bits

1 0 1 0 1 0 1 0 1 1 1 0 0 0 1 1 1 0

1 1 0 0 1 0 1 0 1 0 1 0 1 0 1 1 1 0

Divisão

Os dados de entrada, após o preenchimento, são divididos em blocos de 512 bits (64 bytes). Cada bloco será processado individualmente pelo algoritmo SHA-256.

Exemplo

Se os dados tiverem 100 bytes, serão preenchidos com 412 bits de preenchimento para atingir 512 bits, formando um bloco.

Inicialização dos Valores Iniciais

Inicialização

O algoritmo SHA-256 utiliza um conjunto específico de valores iniciais, conhecidos como Valores Iniciais (IV). Esses valores são constantes definidas no algoritmo e são combinados com os blocos de entrada durante o processamento.

Valores Iniciais

Os valores iniciais (IV) do SHA-256 são constantes específicas definidas pelo algoritmo

Processamento dos Blocos de Dados

1 0 1 0 1 0 1 0 1 1 1 0 0 0 1 1 1 0

Processamento

Cada bloco de 512 bits de dados é processado pelo algoritmo SHA-256. Isso envolve várias etapas, incluindo permutações, combinações bit a bit, rotações e aplicações de funções não-lineares.

Exemplo

O algoritmo SHA-256 realiza uma série complexa de operações bitwise e aritméticas em cada bloco de dados.

1 1 0 0 1 0 1 0 1 0 1 0 1 1 1 0

Computação do Hash Final

1 0 1 0 1 0 1 1 1 0 0 0 1 1 1 0

Computar o Hash

Após o processamento de todos os blocos de dados, o hash final é computado. Este hash é a representação única e irreversível dos dados de entrada.

Hash Final

O hash final é uma sequência de 256 bits (32 bytes) representada em hexadecimal.

1 1 0 0 1 0 1 0 1 0 1 0 1 1 1 0

Saída do Hash

1 0 1 0 1 0 1 0 1 1 1 0 0 0 1 1 1 0

Saída desejada

O hash final, gerado pelo algoritmo SHA-256, é a saída desejada. Ele pode ser utilizado para verificar a integridade dos dados originais ou para autenticar a identidade dos dados.

Utilização

O hash SHA-256 é frequentemente utilizado em sistemas de autenticação, criptografia de senhas, verificação de arquivos e muito mais.

1 1 0 0 1 0 1 0 1 0 1 0 1 1 1 1 0

Passo 1 — Pré-processamento

1 0 1 0 1 0 1 1 1 0 0 0 1 1 1 0

Convertendo “hello world” para binário

01101000 01100101 01101100 01101100
01101111 00100000 01110111 01101111
01110010 01101100 01100100

Adicione um único 1

01101000 01100101 01101100 01101100
01101111 00100000 01110111 01101111
01110010 01101100 01100100 1

1 1 0 0 1 0 1 0 1 0 1 0 1 1 1 0

Passo 1 — Pré-processamento

Preencha com 0 até que os dados sejam múltiplos de 512, menos 64 bits (448 bits no nosso caso):

1010101101100011110

01101000 01100101 01101100 01101100
01101111 00100000 01110111 01101111
01110010 01101100 01100100 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 00000000

1100101010101101110

Passo 1 — Pré-processamento

Anexe 64 bits ao final, onde os 64 bits são um número inteiro big-endian que representa o comprimento da entrada original em binário. No nosso caso, 88, ou em binário, “1011000”.

01101000 01100101 01101100 01101100
01101111 00100000 01110111 01101111
01110010 01101100 01100100 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 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 01011000

1011010110001110

1100101011010110

Passo 2 — Inicializando valores de Hash (H)•••

Agora criamos 8 valores de hash. Estas são constantes codificadas que representam os primeiros 32 bits das partes fracionárias das raízes quadradas dos primeiros 8 primos: 2, 3, 5, 7, 11, 13, 17, 19

1 0 1 0 1 0 1 1 1 1 0 0 0 1 1 1 0

1 1 0 0 1 0 1 0 1 0 1 0 1 0 1 1 1 0

h0 := 0x6a09e667
h1 := 0xbb67ae85
h2 := 0x3c6ef372
h3 := 0xa54ff53a
h4 := 0x510e527f
h5 := 0x9b05688c
h6 := 0x1f83d9ab
h7 := 0x5be0cd19

Passo 3 — Inicializando constantes (K)

Semelhante ao passo 2, estamos criando algumas constantes (saiba mais sobre constantes e quando usá-las aqui). Desta vez, são 64. Cada valor (0-63) são os primeiros 32 bits das partes fracionárias das raízes cúbicas dos primeiros 64 números primos (2 - 311).

0x428a2f98	0x71374491	0xb5c0fbcf	0xe9b5dba5
0x3956c25b	0x59f111f1	0x923f82a4	0xab1c5ed5
0xd807aa98	0x12835b01	0x243185be	0x550c7dc3
0x72be5d74	0x80deb1fe	0x9bdc06a7	0xc19bf174
0xe49b69c1	0xefbe4786	0x0fc19dc6	0x240ca1cc
0x2de92c6f	0x4a7484aa	0x5cb0a9dc	0x76f988da
0x983e5152	0xa831c66d	0xb00327c8	0xbf597fc7
0xc6e00bf3	0xd5a79147	0x06ca6351	0x14292967
0x27b70a85	0x2e1b2138	0x4d2c6dfc	0x53380d13
0x650a7354	0x766a0abb	0x81c2c92e	0x92722c85
0xa2bfe8a1	0xa81a664b	0xc24b8b70	0xc76c51a3
0xd192e819	0xd6990624	0xf40e3585	0x106aa070
0x19a4c116	0x1e376c08	0x2748774c	0x34b0bcb5
0x391c0cb3	0x4ed8aa4a	0x5b9cca4f	0x682e6ff3
0x748f82ee	0x78a5636f	0x84c87814	0x8cc70208
0x90befffa	0xa4506ceb	0xbef9a3f7	0xc67178f2

Passo 4 — LOOP DE PEDAÇO

1 0 1 0 1 0 1 1 1 0 0 0 1 1 1 0

As etapas a seguir acontecerão para cada “pedaço” de dados de 512 bits de nossa entrada. No nosso caso, como “olá mundo” é muito curto, temos apenas um pedaço. A cada iteração do loop, estaremos alterando os valores de hash h0-h7, que será a saída final.

1 1 0 0 1 0 1 0 1 0 1 0 1 1 1 0

Passo 5 — Criar (W)

Copie os dados de entrada da etapa 1 em uma nova matriz onde cada entrada é uma palavra de 32 bits:

01101000011001010110110001101100	01101111001000000111011101101111
01110010011011000110010010000000	00000000000000000000000000000000
00000000000000000000000000000000	00000000000000000000000000000000
00000000000000000000000000000000	00000000000000000000000000000000
00000000000000000000000000000000	00000000000000000000000000000000
00000000000000000000000000000000	00000000000000000000000000000000
00000000000000000000000000000000	00000000000000000000000001011000

1 0 1 0 1 0 1 1 1 1 0 0 0 1 1 1 0

1 1 0 0 1 0 1 0 1 0 1 0 1 0 1 1 1 0

Passo 5 – Criar (W)

Adicione mais 48 palavras inicializadas em zero, de modo que tenhamos um array w

```
01101000011001010110110001101100 01101111001000000111011101101111
01110010011011000110010010000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000001011000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
...
...
00000000000000000000000000000000 00000000000000000000000000000000
```

Passo 5 — Criar (W)

1 0 1 0 1 0 1 1 1 0 0 0 1 1 1 0

Modifique os índices zerados no final da matriz usando o seguinte algoritmo:

For i from w[16...63]:
s0 = (w[i-15] rightrotate 7) xor (w[i-15] rightrotate 18) xor (w[i-15] rightshift 3)
s1 = (w[i- 2] rightrotate 17) xor (w[i- 2] rightrotate 19) xor (w[i- 2] rightshift 10)
w[i] = w[i-16] + s0 + w[i-7] + s1

1 1 0 0 1 0 1 0 1 0 1 0 1 1 1 0

Passo 5 — Criar (W)

1 0 1 0 1 0 1 1 1 0 0 0 1 1 1 0

Vamos fazer w[16] para vermos como funciona:

For i from w[16...63]:

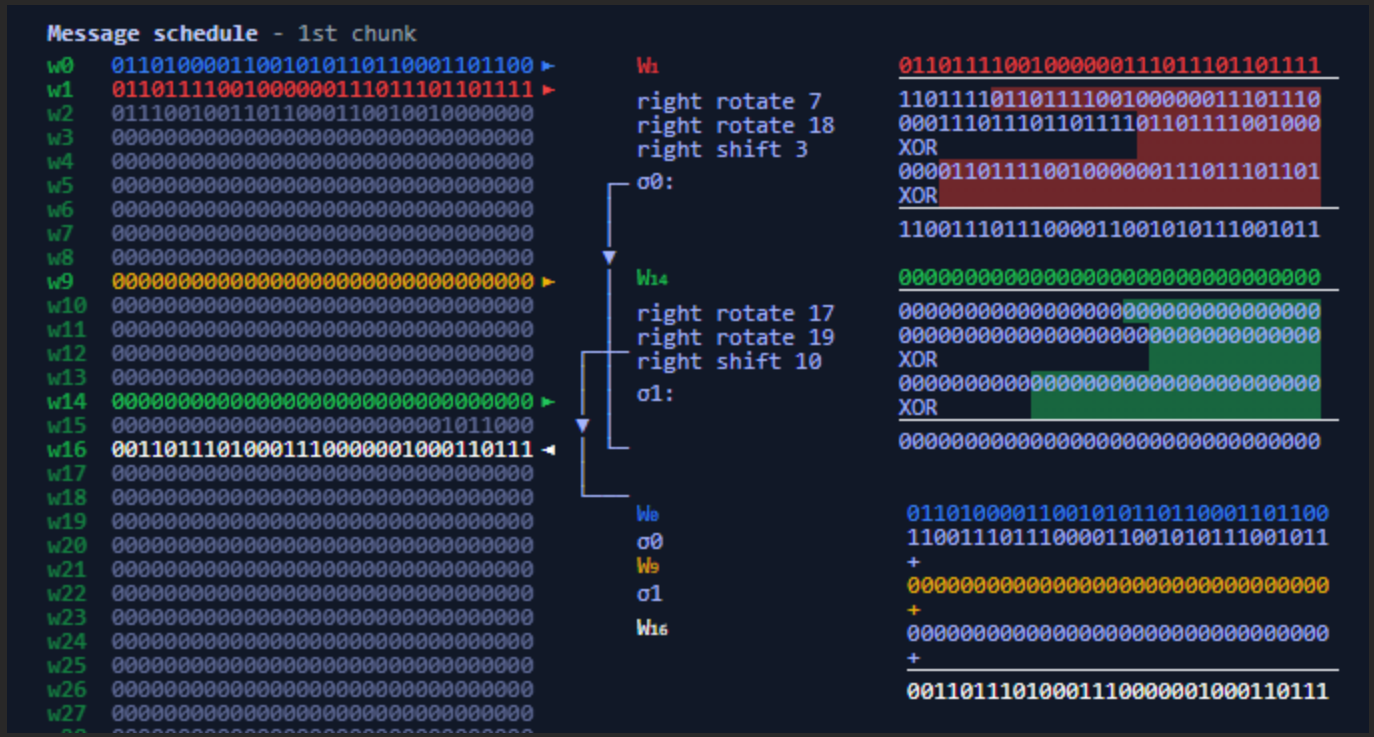
s0 = (w[i-15] rightrotate 7) xor (w[i-15] rightrotate 18) xor (w[i-15] rightshift 3)

s1 = (w[i- 2] rightrotate 17) xor (w[i- 2] rightrotate 19) xor (w[i- 2] rightshift 10)

w[i] = w[i-16] + s0 + w[i-7] + s1

1 1 0 0 1 0 1 0 1 0 1 0 1 1 1 0

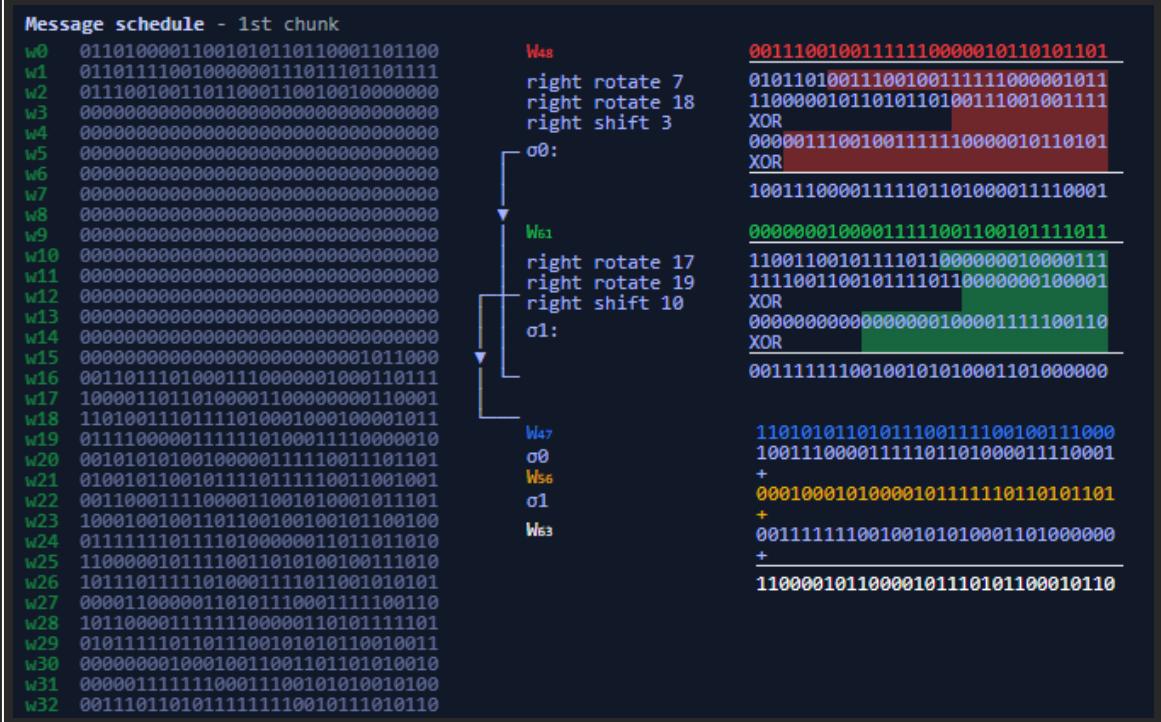
Passo 5 — Criar (W)

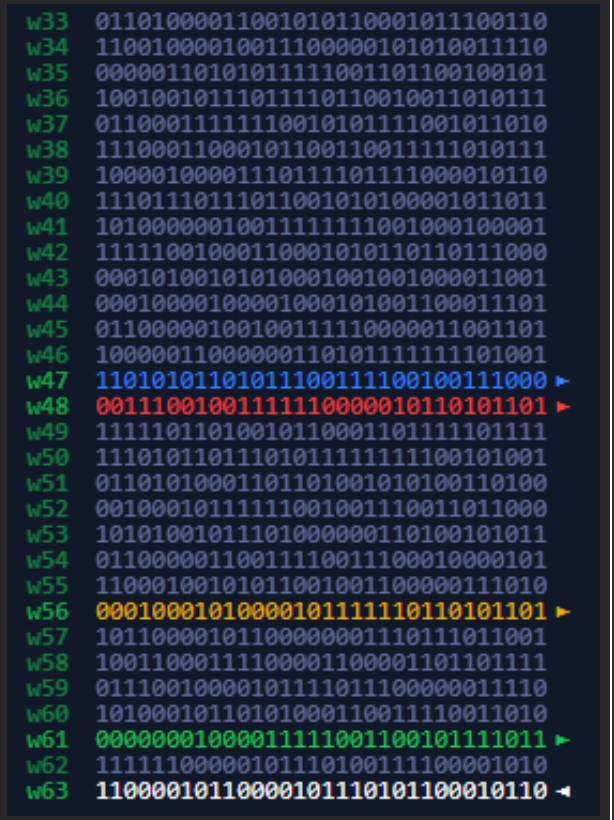


1 0 1 0 1 0 1 1 1 1 0 0 0 1 1 1 0

1 1 0 0 1 0 1 0 1 0 1 0 1 1 1 1 0

Passo 5 — Criar (W)





Passo 6 – Compressão

Inicialize as variáveis a, b, c, d, e, f, g, h e defina-as iguais aos valores de hash atuais, respectivamente. h0, h1, h2, h3, h4, h5, h6, h7

Execute o loop de compressão. O loop de compressão alterará os valores de a...h. O loop de compressão é o seguinte:

```
for i from 0 to 63
    S1 = (e rightrotate 6) xor (e rightrotate 11) xor (e rightrotate 25)
    ch = (e and f) xor ((not e) and g)
    temp1 = h + S1 + ch + k[i] + w[i]
    S0 = (a rightrotate 2) xor (a rightrotate 13) xor (a rightrotate 22)
    maj = (a and b) xor (a and c) xor (b and c)
    temp2 := S0 + maj
    h = g
    g = f
    f = e
    e = d + temp1
    d = c
    c = b
    b = a
    a = temp1 + temp2
```

Passo 6 — Compressão

1 0 1 0 1 0 1 1 1 1 0 0 0 0 1 1 1 0

Working Variables

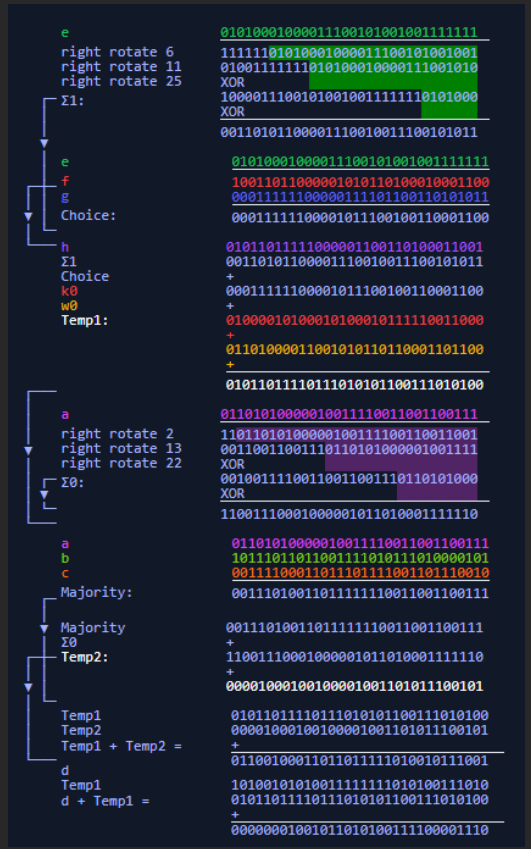
```
a  01101010000010011110011001100111
b  10111011011001111010111010000101
c  00111100011011101111001101110010
d  10100101010011111111101010111010
e  01010001000011100101001001111111
f  10011011000001010110100010001100
g  00011111110000011110110011010111
h  01011011111100000110011010011001
```

Initial hash value

```
h0  01101010000010011110011001100111 = a
h1  10111011011001111010111010000101 = b
h2  00111100011011101111001101110010 = c
h3  10100101010011111111101010111010 = d   Init working
h4  01010001000011100101001001111111 = e   variables
h5  10011011000001010110100010001100 = f
h6  00011111110000011110110011010111 = g
h7  01011011111100000110011010011001 = h
```

Update working variables

```
a =   01100100011011011111010010111001  = Temp1 +
b =   01101010000010011110011001100111  Temp2
c =   10111011011001111010111010000101
d =   00111100011011101111001101110010
e =   00000001001011010100111100001110
f =   01010001000011100101001001111111  = d + Temp1
g =   10011011000001010110100010001100
h =   00011111110000011110110011010111
```



1 1 0 0 1 0 1 0 1 0 1 0 1 1 1 0

Passo 6 — Compressão

Todo esse cálculo é feito mais 63 vezes, modificando as variáveis ah do começo ao fim. Não faremos isso manualmente, mas teríamos finalizado com:

h0 = 6A09E667 = 01101010000010011110011001100111
h1 = BB67AE85 = 10111011011001111010111010000101
h2 = 3C6EF372 = 00111100011011101111001101110010
h3 = A54FF53A = 10100101010011111111010100111010
h4 = 510E527F = 01010001000011100101001001111111
h5 = 9B05688C = 10011011000001010110100010001100
h6 = 1F83D9AB = 00011111100000111101100110101011
h7 = 5BE0CD19 = 01011011111000000110011010001100 1

a = 4F434152 = 01001111010000110100000101010010
b = D7E58F83 = 11010111111001011000111111000011
c = 68BF5F65 = 01101000101111110101111101100101
d = 352DB6C0 = 00110101001011011011011011000000
e = 73769D64 = 01110011011101101001110101100100
f = DF4E1862 = 11011111010011110000110000011000 10
g = 71051E01 = 01110001000001010001111000000001
h = 870F00D0 = 10000111000001111000000000110100 00

1 0 1 0 1 0 1 1 1 1 0 0 0 1 1 1 0

1 1 0 0 1 0 1 0 1 0 1 0 1 1 1 0

Passo 7 — Modificando os valores finais

Após o loop de compressão, mas ainda dentro do loop de pedaços, modificamos os valores de hash adicionando suas respectivas variáveis a eles, a-h. Como sempre, toda adição é módulo 2^32.

h0 = h0 + a = 10111001010011010010011110111001
h1 = h1 + b = 10010011010011010011111000001000
h2 = h2 + c = 10100101001011100101001011010111
h3 = h3 + d = 11011010011111011010101111111010
h4 = h4 + e = 11000100100001001110111111100011
h5 = h5 + f = 01111010010100111000000011101110
h6 = h6 + g = 10010000100010001111011110101100
h7 = h7 + h = 11100010111011111100110111101001

Passo 8 — Concatenação do Hash Final

Por último, mas não menos importante, junte todos eles, uma simples concatenação de strings bastará.

1 0 1 0 1 0 1 1 1 1 0 0 0 0 1 1 1 1 0

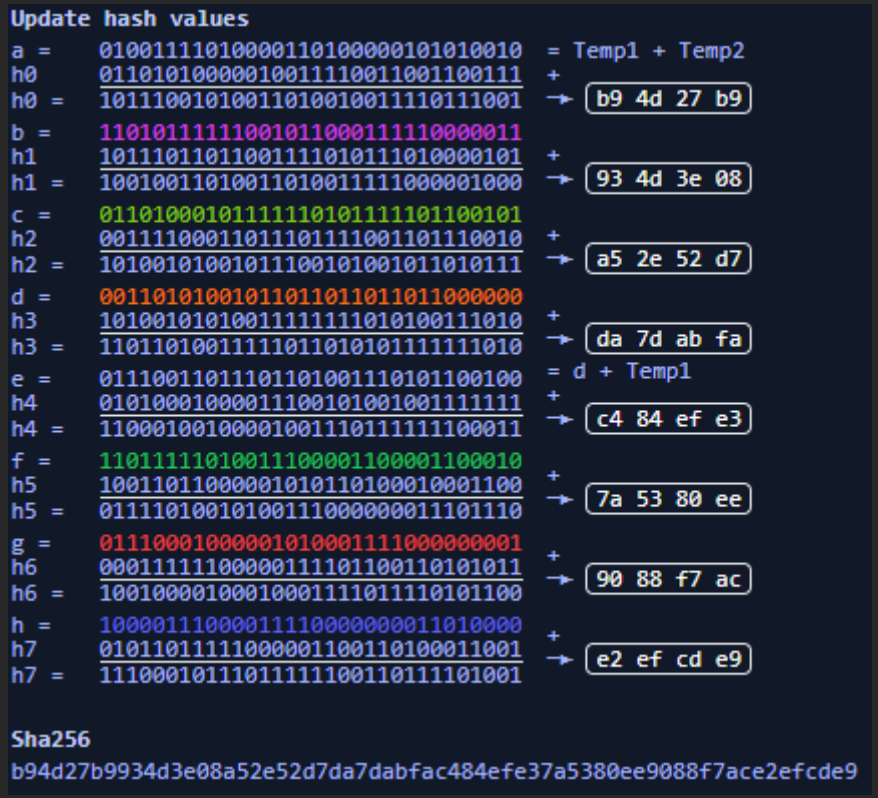
1 1 0 0 1 0 1 0 1 0 1 0 1 0 1 1 1 1 0

digest = h0 append h1 append h2 append h3 append h4 append h5 append h6 append h7

=

B94D27B9934D3E08A52E52D7DA7DABFAC484EFE37A5380EE9088F7ACE2EFCDE9

Exemplo do passo 7 e 8



Alunos

Felipe Franco Pinheiro
Yann Lucas Saito da Luz