

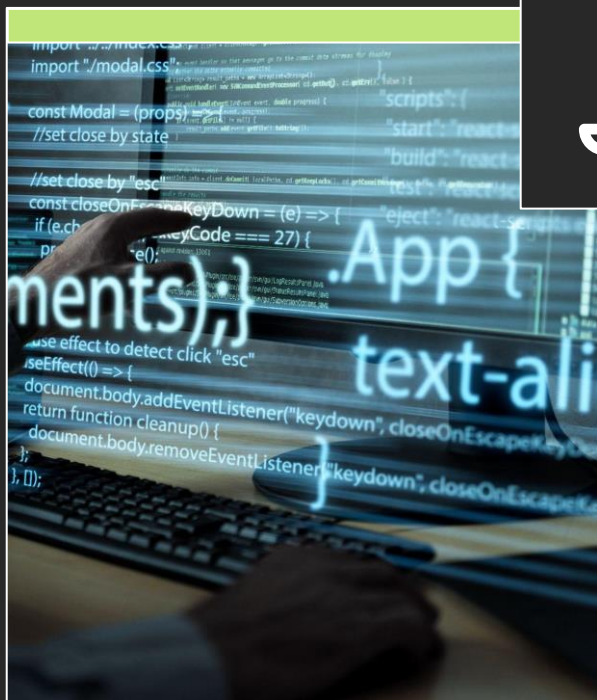
1  
0  
0  
1  
0  
1  
0  
1  
0  
1  
0  
1  
0  
1  
0  
0

# Entendendo o SHA-256



Entendendo o Funcionamento  
do SHA-256

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



01

SHA-256

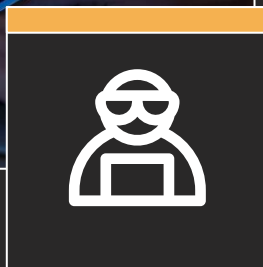
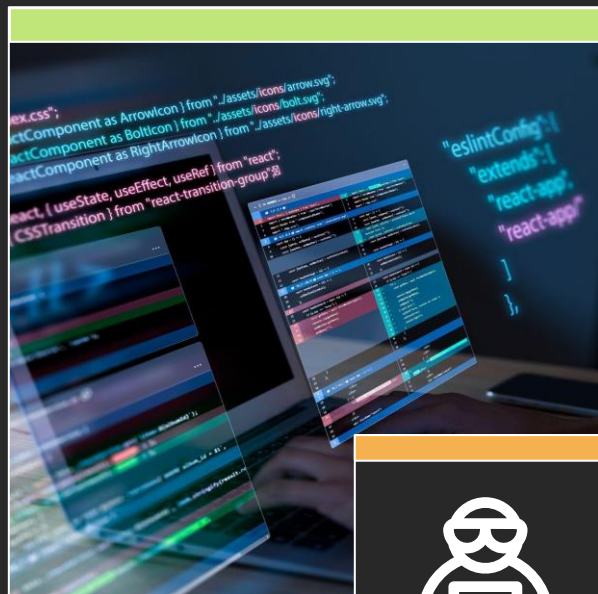
1  
0  
1  
0  
1  
0  
1  
1  
1  
0  
0  
0  
1  
1  
1  
0

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



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

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

# 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



## 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

1  
0  
1  
1  
0  
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

# Preenchimento de Dados



## 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  
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  
1  
1  
0

# Divisão em Blocos de 512 Bits



## 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.

1  
0  
1  
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  
1  
1  
1  
0

# 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

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  
1  
1  
0



# Processamento dos Blocos de Dados

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

# Computação do Hash Final

1  
0  
1  
1  
0  
1  
1  
1  
1  
0  
0  
0  
0  
1  
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.

10101  
01011

## 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  
1  
0

# Saída do Hash



## 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  
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  
1  
1  
0

# Passo 1 – Pré-processamento



Convertendo “hello world” para binário

```
01101000 01100101 01101100 01101100
01101111 00100000 01110111 01101111
01100010 01101100 01100100
```

Adicione um único 1

```
01101000 01100101 01101100 01101100
01101111 00100000 01110111 01101111
01100010 01101100 01100100 1
```

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  
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):

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

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

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

# 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”.

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

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

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

## 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  
0  
0  
0  
1  
1  
1  
0

$h_0 := 0x6a09e667$   
 $h_1 := 0xbb67ae85$   
 $h_2 := 0x3c6ef372$   
 $h_3 := 0xa54ff53a$   
 $h_4 := 0x510e527f$   
 $h_5 := 0x9b05688c$   
 $h_6 := 0x1f83d9ab$   
 $h_7 := 0x5be0cd19$

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

## 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	1
1	0x3956c25b	0x59f111f1	0x923f82a4	0xab1c5ed5	0
0	0xd807aa98	0x12835b01	0x243185be	0x550c7dc3	0
1	0x72be5d74	0x80deb1fe	0x9bdc06a7	0xc19bf174	1
0	0xe49b69c1	0xefbe4786	0x0fc19dc6	0x240ca1cc	0
1	0x2de92c6f	0x4a7484aa	0x5cb0a9dc	0x76f988da	1
1	0x983e5152	0xa83lc66d	0xb00327c8	0xbf597fc7	0
1	0xc6e00bf3	0xd5a79147	0x06ca6351	0x14292967	1
0	0x27b70a85	0x2e1b2138	0x4d2c6dfc	0x53380d13	1
0	0x650a7354	0x766a0abb	0x81c2c92e	0x92722c85	0
1	0xa2bfe8a1	0xa81a664b	0xc24b8b70	0xc76c51a3	
1	0xd192e819	0xd6990624	0xf40e3585	0x106aa070	
1	0x19a4c116	0x1e376c08	0x2748774c	0x34b0bcb5	
0	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	
1	01110010011011000110010010000000	00000000000000000000000000000000	1
0	00000000000000000000000000000000	00000000000000000000000000000000	0
1	00000000000000000000000000000000	00000000000000000000000000000000	1
0	00000000000000000000000000000000	00000000000000000000000000000000	0
1	00000000000000000000000000000000	00000000000000000000000000000000	1
0	00000000000000000000000000000000	00000000000000000000000000000000	0
1	00000000000000000000000000000000	00000000000000000000000000000000	1
1	00000000000000000000000000000000	00000000000000000000000000000000	0
1	00000000000000000000000000000000	00000000000000000000000000000000	1
0			1
0			1
0			0
1			
1			
1			
0			

● ● ●

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

[illegible]

## Passo 5 – Criar (W)



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

```
1
0
1
0
1
0
1
0
1
1
1
0
0
0
1
1
1
0
```

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)



Vamos fazer  $w[16]$  para vermos como funciona:

```
1
0
1
0
1
0
1
1
0
0
0
1
1
1
0
```

For i from  $w[16] \dots 63$ :

```
    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
0
```

# Passo 5 – Criar (W)



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

Message schedule - 1st chunk

```
w0 01101000011001010110001101100 ▶
w1 0110111100100000011101110110111 ▶
w2 0110010011011000110010010000000
w3 0000000000000000000000000000000
w4 0000000000000000000000000000000
w5 0000000000000000000000000000000
w6 0000000000000000000000000000000
w7 0000000000000000000000000000000
w8 0000000000000000000000000000000
w9 0000000000000000000000000000000 ▶
w10 0000000000000000000000000000000
w11 0000000000000000000000000000000
w12 0000000000000000000000000000000
w13 0000000000000000000000000000000
w14 0000000000000000000000000000000 ▶
w15 0000000000000000000000000101100
w16 00110111010001110000001000110111 ▶
w17 0000000000000000000000000000000
w18 0000000000000000000000000000000
w19 0000000000000000000000000000000
w20 0000000000000000000000000000000
w21 0000000000000000000000000000000
w22 0000000000000000000000000000000
w23 0000000000000000000000000000000
w24 0000000000000000000000000000000
w25 0000000000000000000000000000000
w26 0000000000000000000000000000000
w27 0000000000000000000000000000000
```

$w_1$   
right rotate 7  
right rotate 18  
right shift 3

$\sigma_0$ :

$w_{14}$   
right rotate 17  
right rotate 19  
right shift 10

$\sigma_1$ :

$w_6$

$\sigma_0$

$w_9$

$\sigma_1$

$w_{16}$

01101111001000000111011101101111

11011110110111100100000011101110

000111011101101111011111001000

XOR

00001101111001000000111011101101

XOR

11001110111000011001010111001011

0000000000000000000000000000000

0000000000000000000000000000000

0000000000000000000000000000000

XOR

0000000000000000000000000000000

XOR

0000000000000000000000000000000

01101000011001010110001101100

11001110111000011001010111001011

+

0000000000000000000000000000000

+

0000000000000000000000000000000

+

00110111010001110000001000110111

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

# Passo 5 – Criar (W)



## Message schedule - 1st chunk

```
w0 01101000011001010110110001101100
w1 01101111001000000111011101101111
w2 01110010011011000110010010000000
w3 00000000000000000000000000000000
w4 00000000000000000000000000000000
w5 00000000000000000000000000000000
w6 00000000000000000000000000000000
w7 00000000000000000000000000000000
w8 00000000000000000000000000000000
w9 00000000000000000000000000000000
w10 00000000000000000000000000000000
w11 00000000000000000000000000000000
w12 00000000000000000000000000000000
w13 00000000000000000000000000000000
w14 00000000000000000000000000000000
w15 00000000000000000000000000110000
w16 00110111010001110000001000110111
w17 10000110110100001100000000110001
w18 11010011101111010001000100001011
w19 0111100000111111010001110000010
w20 00101010100100000111110011101101
w21 01001011001011110111110011001001
w22 00110001111000011001010001011101
w23 10001001001101100100100101100100
w24 0111111011110100000011011011010
w25 11000001011110011010100110011010
w26 1011011111010001111011001010101
w27 00001100000110101110001111100110
w28 10110000111111100000110101111101
w29 010111110110111001010110010011
w30 00000000100010011001101101010010
w31 00000111111100011100101010010100
w32 001110110101111111001011010110
```

W48

right rotate 7  
right rotate 18  
right shift 3

σ0:

W61

right rotate 17  
right rotate 19  
right shift 10

σ1:

W47

σ0

W56

σ1

W53

00111001001111110000010110101101

01011010011100100111111000001011

11000001011010110100111001001111

XOR

00000111001001111110000010110101

XOR

10011100001111101101000011110001

000000010000111111001100101111011

11001100101111011000000010000111

1111001100101111011000000100001

XOR

00000000000000000100001111100110

XOR

00111111100100101010001101000000

11010101101011100111100100111000

10011100001111101101000011110001

+

00010001010000101111110110101101

+

00111111100100101010001101000000

+

11000010110000101110101100010110

```
w33 01101000011001010110001011100110
w34 110010000100111000000101010011110
w35 000001101010111111001101100100101
w36 10010010111011110110010011010111
w37 01100011111110010101111001011010
w38 11100011000101100110011111010111
w39 10000100001110111101111000010110
w40 11101110111011001010100001011011
w41 10100000010011111111001000100001
w42 111110010001100010101101111000
w43 000101001010100010010000110001
w44 000100001000010001010011100011101
w45 01100000100100111110000011001101
w46 10000011000000110101111111101001
w47 11010101101011100111100100111000 ▶
w48 00111001001111110000010110101101 ▶
w49 11111011010010110001101111101111
w50 11101011011101011111111100101001
w51 01101010001101101001010100110100
w52 0010001011111100100111001110011000
w53 10101001011101000000110100101011
w54 01100000110011110011100010000101
w55 11000100101011001001100000111010
w56 00010001010000101111110110101101 ▶
w57 10110000101100000001110111011001
w58 100110001111100001100001101101111
w59 01110010000101111011100000011110
w60 101000101101010001100111110011010
w61 00000001000011111100110010111011 ▶
w62 11111100000101110100111100001010
w63 11000010110000101110101100010110 ◀
```

● ● ●

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

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

```

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

```



● ● ●

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 = 1010010101001111111010100111010  
h4 = 510E527F = 01010001000001110010100100111111  
h5 = 9B05688C = 10011011000001010110100010001100  
h6 = 1F83D9AB = 00011111100000011110110011010101  
h7 = 5BE0CD19 = 0101101111000001100110100011001

a = 4F434152 = 01001111010000110100000101010010  
b = D7E58F83 = 11010111111001011000111110000011  
c = 68BF5F65 = 0110100010111111010111101100101  
d = 352DB6C0 = 00110101001011011011011011000000  
e = 73769D64 = 0111001101101101001110101100100  
f = DF4E1862 = 11011111010011100001100001100010  
g = 71051E01 = 01110001000001010001111000000001  
h = 870F00D0 = 100001110000111100000000011010000

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

1  
1  
0  
0  
1  
0  
1  
0  
1  
0  
1  
0  
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  $2^{32}$ .

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

```
h0 = h0 + a = 1011100101001101001001011011001
h1 = h1 + b = 10010011010011010011010000001000
h2 = h2 + c = 10100101001011100101001010101011
h3 = h3 + d = 1101101001110101010101110101010
h4 = h4 + e = 1100010010000100110101111100011
h5 = h5 + f = 0111010010100111000000011101101
h6 = h6 + g = 10010000100010001101011101010100
h7 = h7 + h = 1110001011011111001011110101011
```

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

## Passo 8 – Concatenação do Hash Final ...

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

```
digest = h0 append h1 append h2 append h3 append h4 append h5 append h6 append h7
=
B94D27B9934D3E08A52E52D7DA7DABFAC484EFE37A5380EE9088F7ACE2EFCDE9
```

# Exemplo do passo 7 e 8



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

## Update hash values

```
a = 01001111010000110100000101010010 = Temp1 + Temp2
h0  01101010000010011110011001100111 +
h0 = 10111001010011010010011110111001 → b9 4d 27 b9

b = 1101011111001011000111110000011
h1  10111011011001111010111010000101 +
h1 = 10010011010011010011111000001000 → 93 4d 3e 08

c = 01101000101111110101111101100101
h2  00111100011011101111001101110010 +
h2 = 10100101001011100101001011010111 → a5 2e 52 d7

d = 00110101001011011011011011000000
h3  10100101010011111111010100111010 +
h3 = 11011010011111011010101111111010 → da 7d ab fa

e = 01110011011101101001110101100100 = d + Temp1
h4  01010001000011100101001001111111 +
h4 = 11000100100001001110111111000111 → c4 84 ef e3

f = 11011111010011100001100001100010
h5  10011011000001010110100010001100 +
h5 = 01111010010100111000000011101110 → 7a 53 80 ee

g = 01110001000001010001111000000001
h6  00011111100000111101100110101011 +
h6 = 10010000100010001111011110101100 → 90 88 f7 ac

h = 10000111000011110000000011010000
h7  01011011111000001100110100011001 +
h7 = 11100010111011111100110111101001 → e2 ef cd e9
```

Sha256

b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9

1  
1  
0  
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

# Alunos



Felipe Franco Pinheiro  
Yann Lucas Saito da Luz

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