The cryptographic hash function SHA-256

General description

SHA-256 (secure hash algorithm, FIPS 182-2) is a cryptographic hash function with digest length of 256 bits. It is a keyless hash function; that is, an MDC (Manipulation Detection Code).

A message is processed by blocks of $512 = 16 \times 32$ bits, each block requiring 64 rounds.

Basic operations

- Boolean operations AND, XOR and OR, denoted by $\land$, $\oplus$ and $\lor$, respectively.
- Bitwise complement, denoted by $\bar{}$.
- Integer addition modulo $2^{32}$, denoted by $A + B$.

Each of them operates on 32-bit words. For the last operation, binary words are interpreted as integers written in base 2.

- $RotR(A, n)$ denotes the circular right shift of $n$ bits of the binary word $A$.
- $ShR(A, n)$ denotes the right shift of $n$ bits of the binary word $A$.
- $A\|B$ denotes the concatenation of the binary words $A$ and $B$.

Functions and constants

The algorithm uses the functions:

\[ Ch(X, Y, Z) = (X \land Y) \oplus (X \land Z), \]
\[ Maj(X, Y, Z) = (X \land Y) \oplus (X \land Z) \oplus (Y \land Z), \]
\[ \Sigma_0(X) = RotR(X, 2) \oplus RotR(X, 13) \oplus RotR(X, 22), \]
\[ \Sigma_1(X) = RotR(X, 6) \oplus RotR(X, 11) \oplus RotR(X, 25), \]
\[ \sigma_0(X) = RotR(X, 7) \oplus RotR(X, 18) \oplus ShR(X, 3), \]
\[ \sigma_1(X) = RotR(X, 17) \oplus RotR(X, 19) \oplus ShR(X, 10), \]

and the 64 binary words $K_i$ given by the 32 first bits of the fractional parts of the cube roots of the first 64 prime numbers:

```
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 0xe4c85791 0xefbe4786 0x0fc19dc6
0x02b3e88b 0x0a300982 0x116f8801 0x28c7c4a8 0x325c7452 0x40e0d1ab 0x5184faba 0x66f02c54
0x74a002d4 0x82e4dd67 0x92f4f061 0xa3e0f0f4 0xb1f4e2b2 0xc5e0f431 0xd8b7e6dd 0xe2bfe3c8
0xf4292244 0x123b2f29 0x271c4a7c 0x3851f7e2 0x418fe5ee 0x5be3535d 0x6e095cd3 0x8771f2dd
0x936f97c0 0xa517a014 0xb4c07684 0xcde0b0b1 0xd2b4e898 0xe0b5b766 0xec13e4be 0xf98f6245
``
Padding
To ensure that the message\(^1\) has length multiple of 512 bits:

- first, a bit 1 is appended,
- next, \(k\) bits 0 are appended, with \(k\) being the smallest positive integer such that \(l + 1 + k \equiv 448 \mod 512\), where \(l\) is the length in bits of the initial message,
- finally, the length \(l < 2^{64}\) of the initial message is represented with exactly 64 bits, and these bits are added at the end of the message.

The message shall always be padded, even if the initial length is already a multiple of 512.

Block decomposition
For each block \(M \in \{0, 1\}^{512}\), 64 words of 32 bits each are constructed as follows:

- the first 16 are obtained by splitting \(M\) in 32-bit blocks \(M = W_1||W_2||\cdots||W_{16}\)
- the remaining 48 are obtained with the formula:
  \[ W_i = \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16}, \quad 17 \leq i \leq 64. \]

Hash computation

- First, eight variables are set to their initial values, given by the first 32 bits of the fractional part of the square roots of the first 8 prime numbers:
  \[
  H_1^{(0)} = \text{0x6a09e667} \quad H_2^{(0)} = \text{0xbb67ae85} \quad H_3^{(0)} = \text{0x3c6ef372} \quad H_4^{(0)} = \text{0xa54ff53a} \\
  H_5^{(0)} = \text{0x510e527f} \quad H_6^{(0)} = \text{0x9b05688c} \quad H_7^{(0)} = \text{0x1f83d9ab} \quad H_8^{(0)} = \text{0x5be0cd19}
  \]
- Next, the blocks \(M^{(1)}, M^{(2)}, \ldots, M^{(N)}\) are processed one at a time:
  For \(t = 1\) to \(N\)
  - construct the 64 blocks \(W_i\) from \(M^{(t)}\), as explained above
  - set
    \[
    (a, b, c, d, e, f, g, h) = (H_1^{(t-1)}, H_2^{(t-1)}, H_3^{(t-1)}, H_4^{(t-1)}, H_5^{(t-1)}, H_6^{(t-1)}, H_7^{(t-1)}, H_8^{(t-1)})
    \]
  - do 64 rounds consisting of:
    \[
    T_1 = h + \Sigma_1(e) + Ch(e, f, g) + K_i + W_i \\
    T_2 = \Sigma_0(a) + Maj(a, b, c) \\
    h = g \\
    g = f \\
    f = e \\
    e = d + T_1 \\
    d = c \\
    c = b \\
    b = a \\
    a = T_1 + T_2
    \]

\(^1\)We assume that the length of the message can be represented by a 64-bit integer.
compute the new value of $H_j(t)$

\[
\begin{align*}
H_1^{(t)} &= H_1^{(t-1)} + a \\
H_2^{(t)} &= H_2^{(t-1)} + b \\
H_3^{(t)} &= H_3^{(t-1)} + c \\
H_4^{(t)} &= H_4^{(t-1)} + d \\
H_5^{(t)} &= H_5^{(t-1)} + e \\
H_6^{(t)} &= H_6^{(t-1)} + f \\
H_7^{(t)} &= H_7^{(t-1)} + g \\
H_8^{(t)} &= H_8^{(t-1)} + h 
\end{align*}
\]

End for

- The hash of the message is the concatenation of the variables $H_j^{(N)}$ after the last block has been processed

$$H = H_1^{(N)} || H_2^{(N)} || H_3^{(N)} || H_4^{(N)} || H_5^{(N)} || H_6^{(N)} || H_7^{(N)} || H_8^{(N)}.$$ 

### Implementation: signatures

Implement the cryptographic hash function just described. Define the class `sha256` with the method:

```java
public static BigInteger hash(byte[] M)
```

input: $M$ is a chain of bytes of arbitrary length;
output: a positive integer in the interval $[0, 2^{256})$, the value of the hash of $M$.

### Test values

To check the implementation, you can use the following values, given in hexadecimal notation.

<table>
<thead>
<tr>
<th>input</th>
<th>hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>61 62 63</td>
<td>ba7816bf 8f01cfa 4141f0dde 5dae2223 b00361a9c b410f61 f20015ad</td>
</tr>
<tr>
<td>61 62 63 64 62 63 64 65 63 64 65 66 64 65 66 67 65 66 67 68 68 69 69 67 68 69 6a 6b 6c 6d 6e 6f 6e 6f 70 6e 6f 70 71</td>
<td>248d6a61 d20638b8 e5e02693 0c3e6b309 a33ec459 64ff2167 f6ecc4d4 19db06c1</td>
</tr>
<tr>
<td>One million of 61</td>
<td>cdc76e5c 9914fbb92 81a1c7e62 84d73e67 f1809a48 a497200e 046d39cc c7112cd0</td>
</tr>
</tbody>
</table>