MD5

Step 1. Append Padding Bits

The input message is "padded" (extended) so that its length (in bits) equals to 448 mod 512 bits (or 56 mod 64 bytes). Padding is always performed, even if the length of the message is already 448 mod 512 (or 56 and 64 respectively if it's performed in bytes).
Padding is performed as follows: a single bit which is equal to 1 (or byte with hex value of 80) is appended to the message, followed by 0 so that length of the padded message becomes congruent to 448 mod 512 bits (or 56 mod 64 bytes).

Python code:
```python
# Convert message to ascii encoded bytes:
message = bytearray(text.encode('ascii'))
message.append(0x80)
# Add padding
while len(message) % 64 != 56:
    message.append(0)
```

Step 2. Append Length

A 64-bit representation of the length of the message is appended to the result of step1. If the length of the message is greater than 2^{64}, only the low-order 64 bits will be used (this can be done by performing AND operation with greatest 64-bit integer, which is equal to FFFFFFFFFFFFFFFF in hex). The resulting message (after padding with bits and with b) has a length that is an exact multiple of 512 bits. The input message will have a length that is an exact multiple of 16 words each 32 bits or 4 bytes in length.

Python code:
```python
# Calculate and append message length in bytes:
message_length = (8 * len(message)) & 0xFFFFFFFFFFFFFFFF
message += message_length.to_bytes(8, byteorder='little')
```

Step 3. Initialize MD Buffer

A four-word buffer (A, B, C, D) is used to compute the message digest. Each of A, B, C, D is a 32-bit register. These registers are initialized to the following values in hexadecimal, low-order bytes first (little-endian)

Normal (big endian) output:	Little-endian output:
word A: 01 23 45 67	word A: 67 45 23 01
word B: 89 ab cd ef	word B: ef cd ab 89
word C: fe dc ba 98	word C: 98 ba dc fe
word D: 76 54 32 10	word D: 10 32 54 76

Python code:
```python
__init_values = [0x67452301,  # A
                 0xefcdab89,  # B
                 0x98badcfe,  # C
                 0x10325476]  # D
```

Step 4. Process Message in 16-word Blocks

Four functions will be defined such that each function takes an input of three 32-bit words and produces a 32-bit word output. These functions will be used 16 times each.

The nonlinear functions (variable __*functions*) are:

$F(B,C,D) = (B \wedge C) \vee (\neg B \wedge D)$

$G(B,C,D) = (B \wedge D) \vee (C \wedge \neg D)$

$H(B,C,D) = B \oplus C \oplus D$

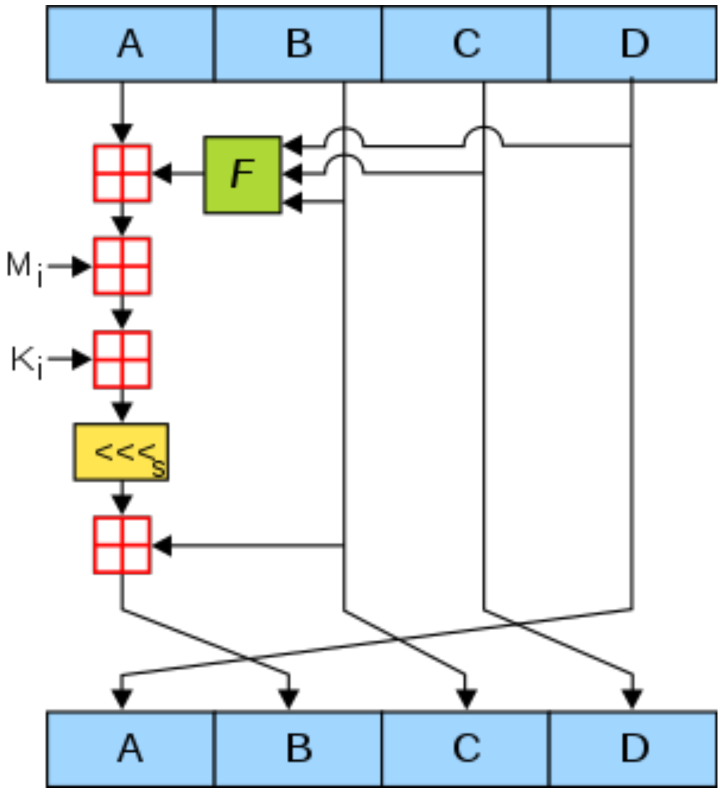
$I(B,C,D) = C \oplus (B \vee \neg D)$

<u>Python code:</u>

```
__functions = 16 * [lambda b, c, d: (b & c) | (~b & d)] + \
              16 * [lambda b, c, d: (d & b) | (~d & c)] + \
              16 * [lambda b, c, d: b ^ c ^ d] + \
              16 * [lambda b, c, d: c ^ (b | ~d)]
```

Step 5. Perform MD5 Rounds

Picture below demonstrates single MD5 round performed on padded message block. First of all, message is split into chunks with size of 64 bits (or 8 bytes), then each chunk is processed 64 times by the aforementioned function. Each new round gets values from previous round according to the diagram, initial values for A, B, C and D are described in Step 3, if message consists of more than one chunk, the initial values for following chunks are taken from last values from the earlier chunk (after 64 rounds).



MD5 consists of 64 of these operations, grouped in four rounds of 16 operations.

- F (variable __*functions*) is a nonlinear function; one function is used in each round.
- M_i (variable __*constants*) denotes a 32-bit block of the message input. Constant consists of 64 elements.

  <u>Constants can be calculated by using this formula in Python:</u>

  ```
  __constants = [int(abs(math.sin(i + 1)) * 2 ** 32) & 0xFFFFFFFF
                 for i in range(64)]
  ```

Or by writing them out in an array:

```
__constants = [0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee,
               0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501,
               0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be,
               0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821,
               0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa,
               0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8,
               0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed,
               0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a,
               0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c,
               0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70,
               0x289b7ec6, 0xeaa127fa, 0xd4ef3085, 0x04881d05,
               0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665,
               0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039,
               0x655b59c3, 0x8f0ccc92, 0xffeff47d, 0x85845dd1,
               0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1,
               0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391]
```

- K_i (variable __*index_functions*) denotes a 32-bit constant, different for each operation. There are 4 different operations that are performed 16 each (just like F function).

  Python implementation:

  ```
  __index_functions = 16 * [lambda i: i] + \
                      16 * [lambda i: (5 * i + 1) % 16] + \
                      16 * [lambda i: (3 * i + 5) % 16] + \
                      16 * [lambda i: (7 * i) % 16]
  ```

- left shift ($\lll_s$) denotes a left bit rotation by s places; s (variable __*rotate_amounts*) varies for each operation.

  Left shift implementation in Python:

  ```
  @staticmethod
  def __left_rotate(x, amount):
      x &= 0xFFFFFFFF
      return ((x << amount) | (x >> (32 - amount))) & 0xFFFFFFFF
  ```

  s implementation in Python:

  ```
  __rotate_amounts = [
  7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22,
  5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20
  4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23
  6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21]
  ```

- Addition ($\boxplus$) denotes addition modulo 2^{32} in Python it is performed by performing AND operation with highest value 32-bit integer which is equal to 2^{32} or FFFFFFFF in hex in Python it is implemented as `& 0xFFFFFFFF`

- Finally, after all rounds are completed, the hash is formed by converting raw bytes in little endian order to 32 symbols length hex number with big endian byte arrangement.

  Python implementation:

  ```
  raw = sum(x << (32 * i) for i, x in
        enumerate(hash_pieces)).to_bytes(16,byteorder='little')
  return '{:032x}'.format(int.from_bytes(raw, byteorder='big'))
  ```