# SHA-256

L'algorithme SHA-256, conçu par la NSA est un algorithme de hashage qui permet de hasher n'importe qu'elle type de donnée, l'algorithme sha-256 fourniras toujours une donnée de 256 bits, peu importe la taille de la donnée fournis, qu'elle soit supérieur ou inférieur a 256, une valeur retourneras toujours le même Hash, néanmoins si elle change même d'un seul caractère elle ce retrouveras alors complétement différentes.

pour présenter l'algorithme nous allons le faire étape par étape par une démonstration guidé:

## Démonstration guidée (version simple)

Pour présenter l'algorithme, on suit un même message pas à pas :

1. Point de départ
    - On part de 8 nombres fixes (valeurs initiales) qui servent d'état de travail. On en fait une copie pour commencer le calcul.
2. Préparation du message
    - On transforme le message en bits.
    - On ajoute un bit "1", puis des "0", puis la taille du message sur 64 bits, pour que la longueur totale soit un multiple de 512.
3. Traitement par blocs de 512 bits
    - On découpe chaque bloc en 16 "mots" de 32 bits, puis on en fabrique 48 de plus pour obtenir 64 mots.
    - On fait 64 "tours" de mélange de l'état avec ces mots et des constantes. C'est là que l'information du message se diffuse et se brouille fortement.
4. Résultat final
    - Après tous les blocs, on obtient 8 nombres finaux. On les écrit en hexadécimal l'un à la suite de l'autre pour former une empreinte (64 caractères) : c'est le SHA-256 du message.

In [60]:
```
K = [
    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,
]

H_INIT = [
    0x6A09E667, 0xBB67AE85, 0x3C6EF372, 0xA54FF53A,
    0x510E527F, 0x9B05688C, 0x1F83D9AB, 0x5BE0CD19,
]
```

## Méthodes auxiliaires

- rotr_int(x, n) : rotation des bits de x vers la droite (rien n'est perdu).
- ch(x, y, z) : "sélecteur" bit à bit (x choisit entre y et z).
- maj(x, y, z) : "vote" bit à bit (valeur majoritaire).
- sigma0_int / sigma1_int : combinaisons de rotations pour bien brasser.

Versions chaînes de bits (pour fabriquer W[16..63]) :

- rotr, shiftr, xor, sigma0, sigma1.

But : calculer à chaque tour des valeurs intermédiaires qui diffusent et brouillent l'information du message.

In [61]:
```
def rotr_int(x, n):
    return ((x >> n) | (x << (32 - n))) & 0xFFFFFFFF


def ch(x, y, z):
    return (x & y) ^ (~x & z)


def maj(x, y, z):
    return (x & y) ^ (x & z) ^ (y & z)


def sigma0_int(x):
    return rotr_int(x, 2) ^ rotr_int(x, 13) ^ rotr_int(x, 22)


def sigma1_int(x):
    return rotr_int(x, 6) ^ rotr_int(x, 11) ^ rotr_int(x, 25)


def rotr(bits, nb):
    if not bits:
        return ""
    n = int(bits, 2)
    mask = (1 << len(bits)) - 1
    nb = nb % len(bits)
    rotated = (n >> nb) | ((n & ((1 << nb) - 1)) << (len(bits) - nb))
    return format(rotated & mask, f"0{len(bits)}b")
```

```
def shiftr(bits, nb):
    if not bits:
        return ""
    return "0" * nb + bits[:-nb] if nb < len(bits) else "0" * len(bits)


def xor(bits1, bits2):
    return "".join("1" if bit1 != bit2 else "0" for bit1, bit2 in zip(bits1,


def sigma0(bits):
    return xor(xor(rotr(bits, 7), rotr(bits, 18)), shiftr(bits, 3))


def sigma1(bits):
    return xor(xor(rotr(bits, 17), rotr(bits, 19)), shiftr(bits, 10))
```

In [62]:
```
def conversionBinaire(message):
    return "".join(["{0:08b}".format(x) for x in message.encode("utf-8")])


def remplissage(message):
    bits = conversionBinaire(message)
    message_size = format(len(message.encode("utf-8")) * 8, "064b")
    bits += "1"
    bits += "0" * ((512 - 64) - len(bits) % 512)
    bits += message_size
    return bits


def décomposition(bits):
    return [bits[i : i + 32] for i in range(0, len(bits), 32)]
```

### Fonction `newMot(bits, t)`

- **Paramètres d'entrée :**
  - `bits` : Une liste de mots de 32 bits (en binaire) représentant les données décomposées et étendues.
  - `t` : Un entier représentant l'indice du mot à générer.
- **Sortie :**
  - Retourne un mot de 32 bits (en binaire) calculé à partir des mots précédents et des fonctions auxiliaires `sigma0` et `sigma1` expliqués précédement.
- **Rôle dans SHA-256 :**
  - Génère un nouveau mot de 32 bits (W[t]) en combinant les résultats des fonctions `sigma0` et `sigma1` appliquées sur des mots spécifiques, ainsi que des additions modulo (2^{32}). Ces mots étendus sont nécessaires pour les calculs dans les 64 itérations de l'algorithme SHA-256.

### Fonction `genererListMot(word)`

- **Paramètres d'entrée :**
  - `word` : Une liste contenant les 16 premiers mots de 32 bits (en binaire) générés à partir du message.
- **Sortie :**
  - Retourne une liste étendue contenant 64 mots de 32 bits (en binaire).
- **Rôle dans SHA-256 :**
  - Étend la liste initiale de 16 mots à 64 mots en utilisant la fonction `newMot`. Ces mots étendus sont utilisés dans les 64 itérations de l'algorithme SHA-256 pour effectuer les calculs nécessaires.
- **Paramètres de la fonction :**
  - `word` : Une liste de 64 mots de 32 bits (en binaire) générés à partir du message. Ces mots sont utilisés pour effectuer les calculs dans chaque itération.
  - `H` : Une liste contenant les valeurs de hachage initiales (ou intermédiaires) utilisées pour démarrer les calculs.
- **Ce que fait la fonction :**
  - **Initialisation :** Une copie des valeurs de hachage `H` est créée dans `S` pour éviter de modifier directement `H` pendant les calculs.
  - **Boucle principale (64 itérations) :**
    - À chaque itération, un mot de 32 bits (`word[i]`) est converti en entier (`w_int`).
    - Deux valeurs temporaires, `T1` et `T2`, sont calculées en utilisant des fonctions auxiliaires (`sigma1_int`, `ch`, `sigma0_int`, `maj`) et des constantes (`K[i]`) définits précédement.
    - Les variables temporaires `S` sont mis à jour en effectuant des décalages et des additions basées sur `T1` et `T2`.
  - **Mise à jour des valeurs de hachage :** Après les 64 itérations, les valeurs de `S` sont ajoutées aux valeurs intermédiaires de `H`.
  - **Retourne :** La liste `H` mise à jour, qui contient les nouvelles valeurs de hachage après le traitement d'un bloc de 512 bits.

In [63]:
```python
def newMot(bits, t):
    s1 = int(sigma1(bits[t - 2]), 2)
    w7 = int(bits[t - 7], 2)
    s0 = int(sigma0(bits[t - 15]), 2)
    w16 = int(bits[t - 16], 2)
    result = (s1 + w7 + s0 + w16) % (2**32)
    return format(result, "032b")
```

```
def genererListMot(word):
    for i in range(16, 64):
        word.append(newMot(word, i))
    return word
```

## Explication simple de `iterateHash(word, H)`

- **Objectif**: traiter un bloc (64 mots) et mettre à jour l'état de hachage.
- **Entrées**
  - `word` : 64 mots de 32 bits (W[0]..W[63]) issus du bloc.
  - `H` : 8 nombres de 32 bits (état courant). On les copie dans `S` pour travailler.
- **Boucle de 64 tours (i = 0..63)**
  - On lit le mot du tour: `w_int = int(word[i], 2)` .
  - On calcule deux valeurs temporaires:
    - `T1 = h + σ1(e) + Ch(e,f,g) + K[i] + W[i]` (mod 2^32)
    - `T2 = Σ0(a) + Maj(a,b,c)` (mod 2^32)
    - Intuition: `T1` injecte le mot du message et la constante du tour; `T2` structure le mélange avec des rotations et une "majorité".
  - On met à jour les 8 registres (rotation des rôles):
    - h ← g, g ← f, f ← e
    - e ← d + T1 (mod 2^32)
    - d ← c, c ← b, b ← a
    - a ← T1 + T2 (mod 2^32)
    - Intuition: on "fait tourner" l'état et on y injecte `T1` / `T2` pour diffuser l'info.
- **Rétro-addition à la fin**
  - Après les 64 tours, on additionne `S` dans `H` mot à mot (mod 2^32).
  - Intuition: on "cumule" l'effet de ce bloc avec l'état global (feed-forward).
- **Détails pratiques**
  - Le `& 0xFFFFFFFF` force les résultats à rester sur 32 bits (arithmétique modulo 2^32).
  - Correspondance habituelle (pour se repérer): `a=S[0]` , `b=S[1]` , `c=S[2]` , `d=S[3]` , `e=S[4]` , `f=S[5]` , `g=S[6]` , `h=S[7]` .
- **À retenir**
  - 64 petits pas qui brouillent fortement les bits du bloc.
  - `W[i]` apporte le message au tour i, `K[i]` ajoute de la variété fixe, `σ/Ch/Maj` assurent le bon mélange.

In [64]:
```
def iterateHash(word, H):
    S = H.copy()
```

```
    for i in range(64):
        w_int = int(word[i], 2)
        T1 = (
            S[7] + sigma1_int(S[4]) + ch(S[4], S[5], S[6]) + K[i] + w_int
        ) & 0xFFFFFFFF
        T2 = (sigma0_int(S[0]) + maj(S[0], S[1], S[2])) & 0xFFFFFFFF
        S[7] = S[6]
        S[6] = S[5]
        S[5] = S[4]
        S[4] = (S[3] + T1) & 0xFFFFFFFF
        S[3] = S[2]
        S[2] = S[1]
        S[1] = S[0]
        S[0] = (T1 + T2) & 0xFFFFFFFF
    for i in range(8):
        H[i] = (H[i] + S[i]) & 0xFFFFFFFF
    return H
```

## Explication de la fonction `sha256(message)`

La fonction `sha256` implémente l'algorithme de hachage SHA-256. Voici une explication étape par étape avec des mots simples :

1. **Initialisation des valeurs de hachage :**
   - On commence par copier les valeurs initiales de hachage ( `H_INIT` ), qui sont des constantes définies au début de l'algorithme.
2. **Remplissage du message :**
   - Le message est converti en une chaîne binaire, puis "rempli" (padding) pour que sa longueur soit un multiple de 512 bits, comme l'exige SHA-256.
3. **Traitement par blocs de 512 bits :**
   - Le message est divisé en blocs de 512 bits.
   - Chaque bloc est décomposé en mots de 32 bits.
   - Des mots supplémentaires sont générés pour étendre la liste à 64 mots.
   - La fonction `iterateHash` est appelée pour mettre à jour les valeurs de hachage en fonction des mots et des constantes de l'algorithme.
4. **Construction du résultat final :**
   - Une fois tous les blocs traités, les valeurs finales de hachage (contenues dans `H` ) sont converties en une chaîne hexadécimale.
   - Cette chaîne représente le hash final du message.

## Résumé

La fonction prend un message en entrée, le transforme en blocs de 512 bits, applique des opérations mathématiques et logiques pour chaque bloc, et retourne un hash unique

de 256 bits (64 caractères hexadécimaux). Ce hash est une empreinte numérique du message, utilisée pour garantir son intégrité.

In [65]:
```python
def sha256(message):
    H = H_INIT.copy()
    bin_str = remplissage(message)

    for i in range(0, len(bin_str), 512):
        block = bin_str[i : i + 512]
        word = genererListMot(décomposition(block))
        H = iterateHash(word, H)

    result = ""
    for h in H:
        result += format(h, "08x")
    return result
```

In [66]:
```python
# Exemple d'utilisation
message = "abc"
result = sha256(message)
print(f"SHA-256 de '{message}': {result}")
```

```
SHA-256 de 'abc': ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f2
0015ad
```