- TD 5 -

Bekkali Hamza

12/02/2025

Exercice 1

a) Écrire une fonction itérative `factorielle(k)` qui retourne la valeur de $k!$.

b) Déterminer le nombre de multiplications qu'effectue `factorielle(k)`. Donner la complexité de cette fonction.
On considère la fonction suivante nommée `mystere` :

```
def mystere(n):
    s = 0
    for k in range(1, n+1):
        s = s + factorielle(k)
    return s
```

c) Déterminer le nombre de multiplications qu'effectue `mystere(n)`. Donner la complexité de cette fonction.

d) Proposer une amélioration de la fonction `mystere` afin d'obtenir une complexité linéaire `O(n)` .

Correction

a) La fonction itérative `factorielle(k)` qui retourne la valeur de $k!$ est la suivante :

```
def factorielle(k):
    result = 1
    for i in range(1, k + 1):
        result *= i
    return result
```

b) Le nombre de multiplications effectuées par `factorielle(k)` est k. La complexité de cette fonction est donc $O(k)$.

c) La fonction `mystere(n)` effectue la somme des factorielles de 1 à n. Le nombre total de multiplications effectuées par `mystere(n)` est la somme des multiplications effectuées par chaque appel à `factorielle(k)` pour k allant de 1 à n. Cela donne :

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

La complexité de `mystere(n)` est donc $O(n^2)$.

d) Pour améliorer la fonction `mystere` et obtenir une complexité linéaire $O(n)$, on peut utiliser une approche où l'on calcule les factorielles de manière cumulative :

```python
def mystere_ameliore(n):
    s = 0
    fact = 1
    for k in range(1, n + 1):
        fact *= k
        s += fact
    return s
```

Cette version de la fonction `mystere` a une complexité $O(n)$ car chaque itération de la boucle effectue un nombre constant d'opérations.

Exercice 2

On considère la fonction `mystere(n)` ci-dessous :

```python
def mystere(n):
  p=log((n),2)
  T=[]
  i , aux = 0,n
  while i<p:
    T.append(aux%2)
    aux=aux//2
    j=i+1
  return T
```

a) Décrire ce que fait cette fonction et proposer une autre solution.

b) Donner la complexité de cette fonction `mystere`.

Correction

a) La fonction `mystere(n)` convertit un nombre entier n en sa représentation binaire inversée sous forme de liste. Voici une autre solution pour obtenir le même résultat :

```python
def mystere_alternative(n):
    T = []
    while n > 0:
        T.append(n % 2)
        n = n // 2
    return T
```

b) La complexité de la fonction `mystere` est $O(\log n)$ car le nombre d'itérations de la boucle `while` est proportionnel au logarithme de n en base 2. Voici une analyse détaillée ligne par ligne :

```python
def mystere(n):
    p = log(n, 2)  # Cette ligne a une complexité O(1) car le calcul du
    ↪  logarithme est une opération constante.
    T = []         # Initialiser une liste vide a une complexité O(1).
    i, aux = 0, n  # L'assignation de variables a une complexité O(1).
    while i < p:   # La boucle while s'exécute environ log(n) fois, donc
    ↪  la complexité est O(log n).
        T.append(aux % 2)  # Ajouter un élément à la liste a une
        ↪  complexité amortie de O(1).
        aux = aux // 2     # La division entière a une complexité O(1).
        i = i + 1          # L'incrémentation de i a une complexité O(1).
    return T               # Retourner la liste a une complexité O(1).
```

En résumé, la boucle `while` est exécutée environ $\log_2(n)$ fois, et chaque opération à l'intérieur de la boucle a une complexité constante $O(1)$. Par conséquent, la complexité totale de la fonction `mystere` est $O(\log n)$.

Exercice 3

On rappelle la définition de la suite de Fibonacci :

$$F_n = \begin{cases} 0 & \text{Si } n = 0 \\ 1 & \text{Si } n = 1 \\ F_{n-1} + F_{n-2} & \text{Si } n > 1 \end{cases}$$

a) Évaluer la complexité de la fonction $fibo1(n)$ suivante :

```python
def fibo(n):
    f0 , f1 = 1, 1
    for i in range(n):
        f0, f1 = f1, f0 + f1
    return f0
```

b) Écrire une fonction récursive qui retourne la valeur de F_n. Donner sa complexité.

Correction

On rappelle la définition de la suite de Fibonacci :

$$F_n = \begin{cases} 1 & \text{Si } n = 0 \\ 1 & \text{Si } n = 1 \\ F_{n-1} + F_{n-2} & \text{Si } n > 1 \end{cases}$$

a) Évaluer la complexité de la fonction `fibo1(n)` suivante :

```python
def fibo(n):
    f0, f1 = 1, 1
    for i in range(n):
        f0, f1 = f1, f0 + f1
    return f0
```

La fonction `fibo(n)` est correcte dans sa déclaration. Elle calcule le n-ième terme de la suite de Fibonacci en utilisant une approche itérative. La complexité de cette fonction est $O(n)$ car la boucle `for` s'exécute n fois.

b) Écrire une fonction récursive qui retourne la valeur de F_n. Donner sa complexité.

```python
def fibo_recursive(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibo_recursive(n-1) + fibo_recursive(n-2)
```

La complexité de cette fonction récursive est $O(2^n)$ car chaque appel à `fibo_recursive` génère deux appels supplémentaires, ce qui entraîne une croissance exponentielle du nombre d'appels récursifs.

Exercice 4

L'algorithme de l'exponentiation rapide, qui utilise le concept de diviser pour régner afin de calculer x^n. Cet algorithme repose sur les formules suivantes :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ a^2 & \text{si } n \text{ est pair, avec } a = x^{n/2} \\ x \cdot a^2 & \text{si } n \text{ est impair, avec } a = x^{(n-1)/2} \end{cases}$$

a) Écrire une fonction récursive `puissanceRec(x, n)` qui retourne la valeur x^n en utilisant le principe de l'exponentiation rapide.

b) Donner la complexité de cette fonction dans le pire des cas.

correction

a) Écrire une fonction récursive `puissanceRec(x, n)` qui retourne la valeur x^n en utilisant le principe de l'exponentiation rapide.

```python
def puissanceRec(x, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        a = puissanceRec(x, n // 2)
        return a * a
    else:
        a = puissanceRec(x, (n - 1) // 2)
        return x * a * a
```

b) La complexité de cette fonction dans le pire des cas est $O(\log n)$ car à chaque appel récursif, la valeur de n est divisée par 2.

- Cas de base : Si $n = 0$, la fonction retourne 1 immédiatement. Cette opération a une complexité $O(1)$.
- Cas pair : Si n est pair, la fonction appelle récursivement `puissanceRec(x, n // 2)`*. La valeur de n est divisée par 2, ce qui réduit le problème de moitié. La multiplication $a * a$ a une complexité $O(1)$.*
- Cas impair : Si n est impair, la fonction appelle récursivement `puissanceRec(x, (n - 1) // 2)`*. La valeur de n est d'abord réduite de 1, puis divisée par 2, ce qui réduit également le problème de moitié. La multiplication $x * a * a$ a une complexité $O(1)$.*

À chaque appel récursif, la taille du problème est divisée par 2. Cela signifie que le nombre total d'appels récursifs est proportionnel au logarithme de n en base 2, soit $\log_2(n)$. Par conséquent, la complexité temporelle de la fonction `puissanceRec` est $O(\log n)$.

Exercice 5

On appelle suite de Thue-Morse la suite définie récursivement par :

$$t_n = \begin{cases} 0 & \text{si } n = 0 \\ t_{n/2} & \text{si } n \text{ est pair et strictement positif} \\ 1 - t_{(n-1)/2} & \text{si } n \text{ est impair} \end{cases}$$

➙ On peut montrer que cette suite ne contient que les valeurs 0 et 1.
➙ Les premières valeurs de la suite sont : 0, 1, 1, 0, 1, 0, 0, 1, 1...

a) Compléter la fonction récursive d'entête `Morse_Recu(n)` qui retourne la valeur du terme t_n :

```python
def Morse_Recu(n):
    if n == 0:
        return 0
    elif n % 2 == 0:
        return Morse_Recu(n // 2)
    else:
        return 1 - Morse_Recu((n - 1) // 2)
```

b) Donner la liste des appels récursifs lors de l'exécution de `Morse_Recu(9)`.

c) Montrer que la complexité de la fonction `Morse_Recu(n)` est $O(\log_2(n))$.

d) On considère la fonction `ThueMorseListe_mauvais(n)` qui retourne les chiffres d'indice inférieur ou égal à n de la suite de Thue-Morse, une liste contenant :

```
def ThueMorseListe_mauvais(n):
    T = [0] * (n + 1)
    for i in range(n + 1):
        T[i] = Morse_Recu(i)
    return T
```

i. Quel est le résultat renvoyé lors de l'appel `ThueMorseListe_mauvais(5)` ?

ii. Montrer que la complexité de la fonction `ThueMorseListe_mauvais(n)` est $O(n\log_2(n))$.

e) Proposer une amélioration de la fonction précédente, on exige que la complexité de cette nouvelle fonction soit $O(n)$.

Correction

a) Compléter la fonction récursive d'entête `Morse_Recu(n)` qui retourne la valeur du terme t_n :

```
def Morse_Recu(n):
    if n == 0:
        return 0
    elif n % 2 == 0:
        return Morse_Recu(n // 2)
    else:
        return 1 - Morse_Recu((n - 1) // 2)
```

b) La liste des appels récursifs lors de l'exécution de `Morse_Recu(9)` est : `Morse_Recu(9)`, `Morse_Recu(4)`, `Morse_Recu(2)`, `Morse_Recu(1)`, `Morse_Recu(0)`.

c) La complexité de la fonction `Morse_Recu(n)` est $O(\log_2(n))$ car à chaque appel récursif, la valeur de n est divisée par 2.

d) On considère la fonction `ThueMorseListe_mauvais(n)` qui retourne les chiffres d'indice inférieur ou égal à n de la suite de Thue-Morse :

```
def ThueMorseListe_mauvais(n):
    T = [0] * (n + 1)
    for i in range(n + 1):
        T[i] = Morse_Recu(i)
    return T
```

i. Le résultat renvoyé lors de l'appel `ThueMorseListe_mauvais(5)` est $[0, 1, 1, 0, 1, 0]$.

ii. La complexité de la fonction `ThueMorseListe_mauvais(n)` est $O(n\log_2(n))$ car pour chaque i de 0 à n, la fonction `Morse_Recu(i)` est appelée, et chaque appel a une complexité $O(\log_2(i))$.

e) Pour améliorer la fonction précédente et obtenir une complexité $O(n)$, on peut utiliser une approche itérative :

```python
def ThueMorseListe(n):
    T = [0] * (n + 1)
    for i in range(1, n + 1):
        if i % 2 == 0:
            T[i] = T[i // 2]
        else:
            T[i] = 1 - T[(i - 1) // 2]
    return T
```

Cette version a une complexité $O(n)$ car chaque élément de la liste est calculé en temps constant.

Exercice : (4 points)

Les nombres amis

Soit x un entier strictement positif. Un entier i est un diviseur strict de x si i divise x et i est différent de x.

Exemple : Les diviseurs stricts du nombre entier 8 sont 1, 2 et 4.

- **Q.1** - Écrire la fonction `divStrict(x)` qui reçoit en paramètre un entier strictement positif x, et qui affiche les diviseurs stricts de x.
  Exemple : `divStrict(8)` affiche les nombres 1, 2 et 4.

- **Q.2** - Déterminer la complexité de la fonction `divStrict(x)`, avec justification.

- **Q.3** - Écrire la fonction `somDivStrict(x)` qui reçoit en paramètre un entier strictement positif x, et qui retourne la somme des diviseurs stricts de x.
  Exemple : `somDivStrict(8)` retourne le nombre $7 = 1 + 2 + 4$.
  Deux entiers strictement positifs x et y sont amis si :
  - x est égal à la somme des diviseurs stricts de y ;
  - y est égal à la somme des diviseurs stricts de x.

- **Q.4** - Écrire la fonction `amis(x, y)` qui reçoit en paramètres deux entiers strictement positifs x et y et qui retourne `True` si x et y sont amis, sinon, la fonction retourne `False`.
  Exemple : `amis(284, 220)` retourne `True`.

- **Q.5** - Écrire la fonction `liste_amis(n)` qui reçoit en paramètre un entier n strictement positif. La fonction retourne une liste L qui contient les tuples de nombres amis (i, j) tels que $0 < i \leq j \leq n$.
  Exemple : `liste_amis(300)` retourne la liste $(6, 6), (28, 28), (220, 284)$.

- **Q.6** - Déterminer la complexité de la fonction `liste_amis(n)`, avec justification.

Partie III : Problème extrait du concours MP2022 *'extrait du concours MP*

Plus Longue Sous Séquence Commune

Une séquence est un ensemble d'éléments non vide, fini et ordonné. Pour représenter une séquence, on utilisera une liste. Si une séquence contient $\boldsymbol{n}$ éléments, alors ses éléments sont indicés de $\mathbf{0}$ à $\boldsymbol{n-1}$. Généralement, on représentera une séquence de n éléments par la liste $X = [x_0, x_1, x_2, \ldots, x_{n-2}, x_{n-1}]$. Pour un entier $\boldsymbol{k} \in \mathbf{1};\boldsymbol{n}$, on notera $\boldsymbol{X_k}$ pour désigner la séquence des $\boldsymbol{k}$ premiers éléments de la séquence $\boldsymbol{X}$. Ainsi, la séquence X_k sera représentée par la liste $[x_0, x_1, x_2, \ldots, x_{k-1}]$.

Dans la suite de cette partie, on suppose que les éléments des séquences sont tous des entiers positifs.

Exemple : La liste $X = [6, 23, 20, 18, 6, 54, 3, 13, 6, 18]$ représente une séquence de 10 éléments.

- ✓ La séquence $\boldsymbol{X}_1$ est représentée par la liste [6]
- ✓ La séquence X_4 est représentée par la liste [6,23,20,18]
- ✓ La séquence X_7 est représentée par la liste $[6, 23, 20, 18, 6, 54, 3]$
- ✓ La séquence X_{10} est représentée par la liste $[6, 23, 20, 18, 6, 54, 3, 13, 6, 18]$

1- Séquence triée

Q.1- Écrire la fonction `tri_sequence`(S) qui reçoit en paramètre une séquence S, et qui trie les éléments de S dans l'ordre croissant.

Exemple : On considère la séquence $S = [6, 23, 20, 18, 6, 54, 3, 13, 6, 18]$. Après l'appel de la fonction `tri_sequence`(S), on obtient la séquence : $[3, 6, 6, 6, 13, 18, 18, 20, 23, 54]$

2- Sous-séquence d'une séquence

On considère deux séquences $\boldsymbol{X}$ et $\boldsymbol{Y}$. On dit que $\boldsymbol{X}$ est une sous-séquence de $\boldsymbol{Y}$, si $\boldsymbol{X}$ est composée des éléments qui apparaissent dans Y, dans le même ordre. Autrement dit, la sous-séquence $\boldsymbol{X}$ est obtenue en supprimant, dans Y, un certain nombre d'éléments (éventuellement aucun élément).

Formellement, étant données deux séquences $X = [x_0, x_1, x_2, \ldots, x_{n-1}]$ et $Y = [y_0, y_1, y_2, \ldots, y_{p-1}]$. On dit que $\boldsymbol{X}$ est une sous-séquence de $\boldsymbol{Y}$, s'il existe une suite croissante d'indices $(\mathbf{i_1}, \mathbf{i_2}, \ldots, \mathbf{i_k})$ de $\boldsymbol{Y}$, telle que pour tout j = 1, 2, ..., k, on a : $\boldsymbol{X}_j = Y_{i_j}$.

Exemple : On considère la séquence $X = [6, 23, 20, 18, 6, 54, 3, 13, 6, 18]$

✓$[6, 20, 18, 13]$ est une sous-séquence de X
✓$[23, 18, 6, 54, 6, 18]$ est une sous-séquence de X
✓$[15, 18, 6, 20]$ n'est pas une sous-séquence de X

Q.2- Écrire la fonction `sous_sequence`(X, Y) qui reçoit en paramètres deux séquences X et Y, et qui retourne True, si X est une sous-séquence de Y, sinon, la fonction retourne False.

Plus longue sous-séquence commune

Le problème de la plus longue sous-séquence commune (PLSSC) à deux séquences X et Y, consiste à trouver une sous-séquence de X et de Y, et qu'elle soit de longueur maximale.

3- Taille de la plus longue sous-séquence commune

Étant données deux séquences $X = [x_0, x_1, x_2, \ldots, x_{n-1}]$ et $Y = [y_0, y_1, y_2, \ldots, y_{p-1}]$. Pour calculer la taille de la PLSSC à X et Y, il est possible de ramener ce problème à un calcul entre deux séquences de taille inférieure grâce à la relation de récurrence suivante :

- si $\boldsymbol{x}_{n-1} = \boldsymbol{y}_{p-1}$ alors,
  La taille de la PLSSC à X et Y = 1+ taille de la PLSSC à X_{n-1} et Y_{p-1}
- si $\boldsymbol{x}_{n-1} \neq \boldsymbol{y}_{p-1}$ alors,
  La taille de la PLSSC à X et Y = max(taille de la PLSSC à X_{n-1} et Y_p, taille de la PLSSC à X_n et Y_{p-1})

Q.3- Écrire la fonction `taille_PLSSC`(X, Y) qui reçoit en paramètres deux séquences X et Y, et qui retourne la taille de la PLSSC à $\boldsymbol{X}$ et $\boldsymbol{Y}$, en utilisant la relation de récurrence citée ci-dessus.

Exemples : On considère la séquence $X = [6, 23, 20, 18, 6, 54, 3, 13, 6, 18]$

- ✓ La fonction `taille_PLSSC`$(\boldsymbol{X}, [20, 23, 10, 18, 6, 25, 3, 1, 6, 8, 2])$ retourne 5
- ✓ La fonction `taille_PLSSC`$(X, [23, 54, 6, 18])$ retourne 4
- ✓ La fonction `taille_PLSSC`$(\boldsymbol{X}, [28, 35, 12, 8, 17])$ retourne 0

Fonctions de hachage *(extrait du concours MP-2023)*

En 2002, la fonction de hachage SHA-256 devient un standard fédéral de traitement de l'information. À partir d'un texte de taille maximale 2^{64} bits (c.à.d. 2 Péta-octets), cette fonction produit un haché de 256 bits, représenté par 64 caractères hexadécimaux.

Exemples : Le haché du message 'Concours CNC 2023' est :
`'ff33dfa67e2c471aa85a83c7a4632955594a946f38ac4a5904873ad5c64f9894'`
Le haché du message 'Concours CNC MP-2023' est :
`'0f3503b5ac59d7160ed4c579d7548100c326c47a8673f36031e27d42e099e3a3'`

A- Manipulation des bits

Q.1- Écrire la fonction NON(X)`NON(`$\boldsymbol{X}$`)` qui reçoit en paramètre une chaîne de caractères X$\boldsymbol{X}$ de taille 32, contenant une représentation binaire. La fonction retourne la chaîne de caractères Z$\boldsymbol{Z}$ de taille 32. Les éléments zi$\boldsymbol{z_i}$ de la chaîne Z$\boldsymbol{Z}$ sont calculés en utilisant l'algorithme suivant :

- Si $x_i =$ '0' alors $z_i =$ '1'
- Si $x_i =$ '1' alors $z_i =$ '0'

Exemple : `NON('01000011010011100100001100100000')`
retourne la chaîne de caractères : `'10111100101100011011110011011111'`

Q.2- Écrire la fonction (X,Y)`ET(`$\boldsymbol{X}$, $\boldsymbol{Y}$`)` qui reçoit en paramètres deux chaînes de caractères X$\boldsymbol{X}$ et Y$\boldsymbol{Y}$ de même taille 32, contenant respectivement deux représentations binaires. La fonction retourne la chaîne de caractères Z$\boldsymbol{Z}$ de taille 32. Les éléments zi$\boldsymbol{z_i}$ de la chaîne Z$\boldsymbol{Z}$ sont calculés en utilisant l'algorithme suivant :

- Si $x_i =$ '1' et $y_i =$ '1' alors $z_i =$ '1'
- Si $x_i =$ '0' ou $y_i =$ '0' alors $z_i =$ '0'

Exemple : `ET('01000011010011100100001100100000', '00110010001100000011001000110011')`
retourne la chaîne : `'00000001000000000000000001000100000'`

Q.3- Écrire la fonction `XOR(`$\boldsymbol{X}, \boldsymbol{Y}$`)` qui reçoit en paramètres deux chaînes de caractères $\boldsymbol{X}$ et $\boldsymbol{Y}$ de même taille 32, contenant respectivement deux représentations binaires. La fonction retourne la chaîne de caractères $\boldsymbol{Z}$ de taille 32. Les éléments $\boldsymbol{z_i}$ de la chaîne $\boldsymbol{Z}$ sont calculés en utilisant l'algorithme suivant :

- Si $x_i = y_i$ alors $z_i =$ '0'
- Si $x_i \neq y_i$ alors $z_i =$ '1'

Exemple : `XOR('01000011010011100100001100100000', '00110010001100000011001000110011')`
retourne la chaîne : `'01110001011111100111000100010011'`

Q.4- Écrire la fonction `decal_droite(`$\boldsymbol{X}, \boldsymbol{p}$`)` qui reçoit en paramètres une chaîne de caractères $\boldsymbol{X}$ de taille 32, contenant une représentation binaire, et un entier $\boldsymbol{p}$ tel que $\mathbf{1} \leq \boldsymbol{p} < \mathbf{32}$. La fonction décale les bits de $\boldsymbol{X}$ de $\boldsymbol{p}$ positions vers la droite : Supprimer les $\boldsymbol{p}$ derniers bits de $\boldsymbol{X}$, et ajouter $\boldsymbol{p}$ fois le bit '0' au début de $\boldsymbol{X}$.

Exemple : `decal_droite('01000011010011100100001100100111', 8)` retourne la chaîne de caractères : `'00000000010000110100111001000011'`

Q.5- Écrire la fonction `rotation_droite`$(\boldsymbol{X}, \boldsymbol{p})$ qui reçoit en paramètres une chaîne de caractères $\boldsymbol{X}$ de taille 32, contenant une représentation binaire, et un entier $\boldsymbol{p}$, tel que $\mathbf{1} \leq \boldsymbol{p} < 32$. La fonction effectue une rotation des bits de $\boldsymbol{X}$ de $\boldsymbol{p}$ positions vers la droite : Déplacer les derniers $\boldsymbol{p}$ bits de $\boldsymbol{X}$ au début de $\boldsymbol{X}$.

Exemple : `rotation_droite('01000011010011100100001100100111', 12)` retourne la chaîne de caractères : `'00110010011101000011010011100100'`

Q.6- Écrire la fonction `addition`$(\boldsymbol{X}, \boldsymbol{Y})$ qui reçoit en paramètres deux chaînes de caractères $\boldsymbol{X}$ et $\boldsymbol{Y}$ de même taille 32, contenant respectivement deux représentations binaires. La fonction retourne la chaîne de caractères de taille 32, résultat du calcul en modulo 2^{32}, de l'addition binaire de $\boldsymbol{X}$ et $\boldsymbol{Y}$. L'addition binaire utilise le principe suivant :

- $0 + 0 = 0$
- $0 + 1 = 1$
- $1 + 0 = 1$
- $1 + 1 = 10$ (on pose 0 et on retient 1)
- $1 + 1 + 1 = 11$ (on pose 1 et on retient 1)

Exemple : -
`addition('11111111111111111111111111111010','00000000000000000000000000011100')` retourne la chaîne de caractères :
`'00000000000000000000000000010110'`

B- Initialisation des variables de hachage

L'algorithme SHA-256 utilise **8** variables globales : $\boldsymbol{h_0}$, $\boldsymbol{h_1}$, $\boldsymbol{h_2}$, $\boldsymbol{h_3}$, $\boldsymbol{h_4}$, $\boldsymbol{h_5}$, $\boldsymbol{h_6}$, $\boldsymbol{h_7}$. Ces variables sont initialisées respectivement par les chaînes de caractères contenant les premiers **32** bits des représentations binaires, des parties fractionnaires des racines carrées des **8** premiers nombres premiers : 2, 3, 5, 7, 11, 13, 17 et 19.

Exemple : -
La racine carrée de **2** est **1.4142135623730951**. Sa partie fractionnaire est **0.4142135623730951**. La représentation binaire de cette partie fractionnaire est :

$$\mathbf{0.01101010000010011110011001111111}\ldots$$

La variable $\boldsymbol{h_0}$ est initialisée par les premiers 32 bits :

$$\mathbf{01101010000010011110011001100111}$$

L'algorithme SHA-256 utilise aussi une variable globale $\boldsymbol{K}$ initialisée par une liste composée de **64** constantes. La liste $\boldsymbol{K}$ est initialisée respectivement par les chaînes de caractères contenant les premiers 32 bits des représentations binaires, des parties fractionnaires des racines cubiques des 64 premiers nombres premiers : 3, 5, 7, 11, 13, 17, 19, ..., 311.

On suppose que L est une variable globale initialisée par les **64** premiers nombres premiers :

$\mathbf{L} = [\mathbf{2}, \mathbf{3}, \mathbf{5}, \mathbf{7}, \mathbf{11}, \mathbf{13}, \mathbf{17}, \mathbf{19}, \mathbf{23}, \mathbf{29}, \mathbf{31}, \mathbf{37}, \mathbf{41}, \mathbf{43}, \mathbf{47}, \mathbf{53}, \mathbf{59}, \ldots, \mathbf{269}, \mathbf{271}, \mathbf{277}, \mathbf{281}, \mathbf{283}, \mathbf{293}, \mathbf{307}, \mathbf{311}]$

NB : Pour calculer nx $\sqrt[n]{\mathbf{x}}(laracinenièmed'unentierx\mathbf{x}), onutiliselasyntaxesuivante : x(1/n)\mathbf{x}^{**}(\mathbf{1}/\mathbf{n})$.

Q.7- Écrire la fonction `bin_fract`$(\boldsymbol{x}, \boldsymbol{n})$ qui reçoit en paramètres un nombre premier $\boldsymbol{x}$ et un entier $\boldsymbol{n} > \mathbf{1}$. La fonction retourne la chaîne de caractères contenant les premiers **32** bits de la représentation binaire, de la partie fractionnaire de $\sqrt[n]{\mathbf{x}}$.

Exemples :

- `bin_fract(7, 2)` retourne la chaîne de caractères : `'10100101010011111111101010011101'`
- `bin_fract(7, 3)` retourne la chaîne de caractères : `'11101001101101011101101110100101'`

Q.8- Écrire la fonction `liste_init`$(\boldsymbol{t}, \boldsymbol{n})$ qui reçoit en paramètres un entier strictement positif $\boldsymbol{t} \leq \mathbf{64}$ et un entier $\boldsymbol{n} > 1$. La fonction retourne une liste composée des chaînes de caractères contenant les premiers **32** bits des représentations binaires, des parties fractionnaires des racines nièmes des t premiers nombres premiers.

Exemple : `liste_init(4, 2)` retourne la liste :
['01101010000010011110011001100111','10111011011001111010111010000101', '00111100011011011110011011100010', '10100101010011111111010100111010']

Q.9- Écrire les instructions permettant d'initialiser les variables globales de hachage : h0$\boldsymbol{h_0}$, h1$\boldsymbol{h_1}$, h2$\boldsymbol{h_2}$, h3$\boldsymbol{h_3}$, h4$\boldsymbol{h_4}$, h5$\boldsymbol{h_5}$, h6$\boldsymbol{h_6}$, h7$\boldsymbol{h_7}$,h8$\boldsymbol{h_8}$ et k$\boldsymbol{k}$

```python
import math
#Q1
def NON(X):
    return ''.join('1' if x == '0' else '0' for x in X)
print("NON('01000011010011100100001100100000')
↪  \n",NON('01000011010011100100001100100000'))
#'10111100101100011011110011011111'

#Q2
def ET(X, Y):
    return ''.join('1' if x == '1' and y == '1' else '0' for x, y in zip(X, Y))
print("ET('01000011010011100100001100100000', '00110010001100000011001000110011')
↪  \n",ET('01000011010011100100001100100000',
↪  '00110010001100000011001000110011'))
#'00000010000000000000001000100000'

#Q3
def XOR(X, Y):
    return ''.join('0' if x == y else '1' for x, y in zip(X, Y))
print("XOR('01000011010011100100001100100000',
↪  '00110010001100000011001000110011')
↪  \n",XOR('01000011010011100100001100100000',
↪  '00110010001100000011001000110011'))
#'01110001011111100111000100010011'
#Q4
def decal_droite(X, p):
    return '0' * p + X[:-p]
print(decal_droite('01000011010011100100001100100111', 8))
#'00000000010000110100111001000011'

#Q5
def rotation_droite(X, p):
    return X[-p:] + X[:-p]
print(rotation_droite('01000011010011100100001100100111', 12))
#'00110010011101000011010011100100'

#Q6
def addition(X, Y):
    max_len = 32
    result = ''
    carry = 0

    for i in range(max_len - 1, -1, -1):
        r = carry
        r += 1 if X[i] == '1' else 0
        r += 1 if Y[i] == '1' else 0
        result = ('1' if r % 2 == 1 else '0') + result
        carry = 0 if r < 2 else 1
    return result
print(addition('11111111111111111111111111111110',
↪  '00000000000000000000000000011100'))
#'00000000000000000000000000010110'
```

Algorithm 1: Tri par comptage

1 Votre Titre ICI

Données: L : une liste à trier, $v_{\min}$, $v_{\max}$: les valeurs extrêmes
Sortie: une nouvelle liste triée T

begin
  $n \leftarrow v_{\max} - v_{\min}$;
  $C \leftarrow [0, \ldots, 0]$ `// (n + 1 cases)`
  chaque terme x de L incrémenter la valeur $C[x - v_{\min}]$;
  i de 0 **à** n ajouter $C[i]$ termes $v_{\min} + i$ dans T;
  Retourner T;

Exercice

Soit le programme suivant :

```
def f(c):
    p = 1
    if c > 0:
        while c != 0:
            p = p * 2
            c = c - 2
    return p
```

- **Q.1** - Que renvoie la fonction f lorsque l'on affecte au paramètre c les valeurs suivantes :
  - $c = 6$?
  - $c = -4.5$?
  - $c = 0$?
  - $c = 4.5$?

- **Q.2** - Déterminer l'ensemble des valeurs à affecter au paramètre c pour que le programme se termine.

- **Q.3** - Documenter le programme.

- **Q.4** - Modifier le programme de sorte qu'il se termine quelle que soit la valeur affectée au paramètre c et qu'il renvoie les mêmes valeurs que précédemment dans les cas où il se terminait.

Exercice "problème de Syracuse"

Définissons une fonction f sur $\mathbb{N}^*$ par, pour n appartenant à $\mathbb{N}^*$,

$$f(n) = \begin{cases} \frac{n}{2} & \text{si } n \text{ est pair} \\ 3n+1 & \text{si } n \text{ est impair} \end{cases}$$

On appelle trajectoire de n la suite obtenue en partant de n et en lui appliquant de façon répétée cette fonction. Cette suite $(t_k)_{k\in\mathbb{N}^*}$ est définie ainsi :

$$\begin{cases} t_1 = n \\ \forall k \in \mathbb{N}^* \quad t_{k+1} = f(t_k) \end{cases}$$

- **Q.1** - Quelle est la trajectoire de 6 ? de 17 ?

- **Q.2** - On constate sur ces deux exemples qu'une fois que le nombre 1 a été atteint, la suite des valeurs $1, 4, 2, 1, 4, 2, \ldots$ se répète indéfiniment en un cycle de longueur 3, appelé cycle trivial. La conjecture de Syracuse est l'hypothèse mathématique selon laquelle la trajectoire de n'importe quel entier strictement positif atteint 1.

- **Q.3** - Écrire une fonction `syracuse` qui prend en paramètre un entier non nul n et qui renvoie $f(n)$.

- **Q.4** - Écrire une fonction `trajectoire` qui prend en paramètres un entier non nul n et un entier non nul N et qui affiche sur une même ligne, séparés par des espaces, les N premiers termes de la trajectoire de n.

- **Q.5** - Écrire une fonction `terme` qui prend en paramètres un entier non nul n et un entier non nul N et qui renvoie le N^e terme de la suite.

- **Q.6** - On admet la conjecture de Syracuse. Écrire une fonction `dureedevol` qui prend en paramètre un entier non nul n et qui renvoie le plus petit entier $k_0 \in \mathbb{N}^*$ tel que $t_{k_0} = 1$. On pourra utiliser la fonction `terme` ou la fonction `syracuse`.

Recursivité

Exercice Puissance

Écrire une fonction `puissance` qui calcule la puissance entière d'un nombre. On pourra écrire une fonction récursive non terminale puis la transformer en une fonction récursive terminale.

— La récursivité «multiple» : la fonction comporte plusieurs appels récursifs. Exemple : calcul du n^e terme d'une suite récurrente linéaire multiple.

Exercice Fibonacci

Écrire une fonction `fibo` qui renvoie le n^e terme de la suite de Fibonacci. Là encore, on pourra écrire une version non terminale et une version terminale. Noter que pour la version terminale, il ne s'agit plus de récursivité multiple.

— La récursivité mutuelle : deux fonctions f et g de paramètre n comportent des appels croisés, c'est-à-dire que la définition de f au rang n va appeler les fonctions f et g à des rangs d'ordres inférieurs, et de même pour la définition de g au rang n.

Exercice Récursivité mutuelle

Calculer le n^e terme des suites (u_n) et (v_n) définies par $u_0 = 1$ et $v_0 = -1$ et, pour tout $n \in \mathbb{N}$,

$$\begin{cases} u_{n+1} = 2u_n - v_n + 3 \\ v_{n+1} = -u_n + 2v_n \end{cases}$$

Exercice Fibonacci : calcul Matricielle

Version matricielle du calcul des termes de la suite de Fibonacci :

Notons $(F_n)_{n\in\mathbb{N}}$ la suite de Fibonacci et posons $U_n = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}$ pour tout $n \in \mathbb{N}$.

- **Q.1** - Calculer U_0 et démontrer qu'il existe une matrice $A \in \mathscr{M}_2(\mathbb{R})$, indépendante de n, telle que

$$\forall n \in \mathbb{N} \quad U_{n+1} = AU_n$$

- **Q.2** - En déduire un nouvel algorithme de calcul des termes de la suite de Fibonacci, utilisant des tableaux du module numpy et ne faisant pas de récursivité multiple.

- **Q.3** - Estimer sa complexité en temps.

Correction

- **Q.1** - Calcul de U_0 :

$$U_0 = \begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Démonstration de l'existence de la matrice A :

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

En effet, nous avons :

$$U_{n+1} = \begin{pmatrix} F_{n+1} \\ F_{n+2} \end{pmatrix} = \begin{pmatrix} F_{n+1} \\ F_n + F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = AU_n$$

- **Q.2** - Algorithme de calcul des termes de la suite de Fibonacci :

```python
import numpy as np

def fibonacci_matrix(n):
    A = np.array([[0, 1], [1, 1]])
    U = np.array([[0], [1]])

    if n == 0:
        return 0
    elif n == 1:
        return 1

    result = np.linalg.matrix_power(A, n-1).dot(U)
    return result[1, 0]

# Exemple d'utilisation
n = 10
print(f"Le terme F_{n} de la suite de Fibonacci est :
↪  {fibonacci_matrix(n)}")
```

- **Q.3** - Estimation de la complexité en temps : La complexité temporelle de cet algorithme est $O(\log n)$ en raison de l'utilisation de l'exponentiation rapide des matrices.

Exercice inversion d'une liste

Écrire une version itérative pour l'inversion de l'ordre des éléments d'une liste et estimer la complexité temporelle ainsi que la complexité mémoire. Comparer.

Exercice Minimum d'une liste

- **Q.1** - En remarquant que $\text{Min}\,[a_0, \ldots, a_n] = \text{Min}\,(\text{Min}\,[a_0, \ldots, a_{n-1}], a_n)$, écrire une fonction récursive de paramètre une liste L et qui renvoie le minimum de cette liste L.

- **Q.2** - Effectuer une preuve de terminaison de cet algorithme.

- **Q.3** - Déterminer la complexité temporelle associée.

Exercice Puissance et puissance rapide

Puissance et puissance rapide :

- **Q.1** - Déterminer la complexité de l'algorithme de "l'exercice Puissance".

- **Q.2** - L'algorithme de puissance rapide repose sur la propriété suivante : pour a réel et n entier naturel non nul,

$$a^n = \begin{cases} a^{\frac{n}{2}} \times a^{\frac{n}{2}} & \text{si } n \text{ est pair} \\ a^{\frac{n-1}{2}} \times a^{\frac{n-1}{2}} \times a & \text{si } n \text{ est impair} \end{cases}$$

  - **Q.3** -.1 Écrire une nouvelle fonction récursive `puissance_rapide` de paramètres a et n qui renvoie la valeur de a^n en utilisant la remarque précédente.
  - **Q.3** -.2 Écrire la preuve de terminaison de `puissance_rapide`.
  - **Q.3** -.3 Pourquoi cet algorithme est-il appelé «puissance rapide» ?

Exercice Fonction de McCarthy

Soit la fonction

$$f : \begin{cases} \mathbb{N} \longrightarrow \mathbb{N} \\ n \longmapsto \begin{cases} n - 10 & \text{si } n > 100 \\ f(f(n+11)) & \text{sinon} \end{cases} \end{cases}$$

- **Q.1** - Programmer une fonction en Python permettant le calcul des valeurs de f et calculer $f(n)$ pour tout $n \in 0; 120$.

- **Q.2** - Conjecturer la valeur de $f(n)$ pour tout $n \in \mathbb{N}$.

- **Q.3** - Démontrer cette conjecture. On pourra commencer par calculer $f(n)$ pour $n \in 90; 100$, puis pour $n \in 80; 90$, etc.