TP 3

TP 3 – Exercice 1

Enoncé :

Donner une définition, avec signature et hypothèse(s) éventuelle(s), de la fonction **compte_mots** qui, étant donné une chaîne de caractères s, renvoie le nombre de mots que contient cette chaîne.

On considère ici les mots “au sens large” : les mots sont séparés par un ou des espaces.

Exemple :

>>> compte_mots(’’) => 0

>>> compte_mots(’il ingurgite impunément un iguane.’) => 5

>>> compte_mots(’coursdeprogrammation’) => 1

>>> compte_mots(" Attention aux espaces consécutifs ou terminaux ") => 6

TP 3 – Exercice 2

Enoncé :

Donner une définition, avec signature et hypothèse(s) éventuelle(s), de la fonction **remplace_multiple** qui, étant donné étant donné deux chaînes de caractères s1 et s2, ainsi qu'un entier naturel n, renvoie la chaîne obtenue en remplaçant le caractère en position n dans s1 par le premier caractère de s2, puis le caractère en position 2n dans s1 par le deuxième caractère de s2, etc.. Le remplacement s'arrête quand il n'y a plus de caractères dans s2. Une fois la fin de s1 atteinte, s'il reste des caractères dans s2 non utilisés, on les ajoute au bout de la chaîne obtenue.

Exemple :

>>> remplace_multiple('','',2) => ''

>>> remplace_multiple('abacus','oiseau',2) => 'abocisseau'

>>> remplace_multiple('hirondelles','nid',3) => 'hirnndillds'

TP 3 – Exercice 3

Enoncé 1 :

Donner une définition de la fonction **termeU** qui, étant donné un entier naturel n, rend la valeur de *Un* correspondante.

Exemple :

>>> termeU(0) => 1

>>> termeU(1) => 3

>>> termeU(5) => 59013

>>> termeU(10) => 64885583712887818

$$S(p) = \sum_{n=0}^{p} u_n \qquad \text{avec} \begin{cases} u_0 = 1 \\ u_n = u_{n-1} \times 2^n + n \quad \text{si } n \geq 1 \end{cases}$$

TP 3 – Exercice 3

Enoncé 2 :

Pour calculer la valeur de S(p), un premier algorithme utilise la fonction termeU écrite à la question précédente. On réalise une itération pour calculer les valeurs de u0, u1, . . . , up à l'aide de termeU et les additionner au fur et à mesure.En utilisant cet algorithme, donner une définition de la fonction serie qui, étant donné un entier naturel p, rend la valeur correspondante de S(p).

Exemple :

>>> serie(0) => 1

>>> serie(1) => 4

>>> serie(5) => 60990

>>> serie(10) => 64949072787434918

$$S(p) = \sum_{n=0}^{p} u_n \qquad \text{avec} \begin{cases} u_0 = 1 \\ u_n = u_{n-1} \times 2^n + n & \text{si } n \geq 1 \end{cases}$$

TP 3 – Exercice 3

Enoncé 3 :

On peut reprocher à l’algorithme précédent de devoir recalculer un à partir de *U0* à chaque itération. Certaines valeurs de la suite sont donc recalculées plusieurs fois. Un autre algorithme calcule S(p) sans utiliser la fonction termeU : à chaque itération, *Un* est déterminé à partir de la valeur de *Un−1* calculée et mémorisée à l’itération précédente. En utilisant ce second algorithme, donner une définition de la fonction serie_v2 qui, étant donné un entier naturel prend la valeur correspondante de S(p).

Exemple :

>>> serie_v2(0) => 1

>>> serie_v2(1) => 4

>>> serie_v2(5) => 60990

>>> serie_v2(10) => 64949072787434918

TP 3 – Exercice 3

Enoncé 4 :

Remplir le tableau ci-dessous à partir de l’état des variables à chaque itération.

Itération	Variable ...	Variable ...	Variable ...	Variable ...
1				
2				

TP 3 – Exercice 3

Enoncé 5 :

Mesurer les temps d’exécution nécessaires au calcul des valeurs de S(100), S(300), S(400),..., S(1000) en utilisant chacune des deux fonctions serie et serie_v2. <u>Quel algorithme est le plus efficace ?</u>

<u>Remarque</u> : pour mesurer le temps d’exécution d’une application de fonction, il est possible d’utiliser la fonction clock() de la bibliothèque time. La fonction clock() rend l’heure au moment où elle est appelée. On mesure donc l’heure avant puis après l’exécution d’une application afin de connaître le temps qui lui est nécessaire.

Exemple :

```
import time

depart = time.clock()

serie(100)

arrivee = time.clock()

print('temps passe en secondes : ', arrivee-depart)
```

TP 3 – Exercice 4

Enoncé :

Donner une définition (avec une version fonctionnelle et une version non fonctionnel) de la fonction **factorielle** qui, étant donné un entier naturel n, rend sa factorielle.

$$n! = \prod_{1 \leqslant i \leqslant n} i = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n.$$

>>> factorielle(1) => 1

>>> factorielle(2) => 2

>>> factorielle(3) => 6

>>> factorielle(4) => 24