1 On fait appel aux fonctions suivantes avec $n = 4$, prévoir le résultat affiché.

1.

```
def T1(n):
    print(n * "a") # on affiche n fois le caractère a
    if n > 0:
        T1(n - 1)
```

2.

```
def T2(n):
    if n > 0:
        T2(n - 1)
    print(n * "a")
```

3.

```
def T3(n):
    print(n * "a")
    if n > 0:
        T3(n - 1)
    print(n * "a")
```

4.

```
def T4(n):
    if n > 0:
        T4(n - 1)
    print(n * "a")
    if n > 0:
        T4(n - 1)
```

2 La fonction suivante permet de savoir si un entier naturel est pair :

```
def pair(n):
    while n > 0:
        n = n - 2
    if n == 0:
        return("pair")
    else:
        return("impair")
```

Écrire une version récursive de cette fonction.

3 On considère la suite (u_n) définie par :

$$\begin{cases} u_0 = 1 \\ \forall n \in \mathbb{N}^*,\ u_n = \dfrac{1}{2}\left(u_{n-1} + \dfrac{3}{u_{n-1}}\right) \end{cases}$$

Écrire une fonction récursive qui calcule la valeur de (u_n) en fonction de l'entier naturel n. Votre fonction devra avoir une complexité linéaire.

4 On considère la fonction récursive suivante :

```
def f(n):
    print(n)
    if n > 100:
        return(n - 10)
    else:
        return(f(f(n + 11)))
```

Que vaut $f(50)$?

5 À quoi peut servir la fonction suivante ?

```
def limite(n):
    if n > 0:
        limite(n - 1)
    else:
        print("terminé")
```

6 Le but de cet exercice est de générer une liste contenant les 2^n sous-listes possibles d'une liste contenant les entiers de 1 à n. Par exemple, on doit obtenir :

```
>>> parties([1,2,3,4])
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3], [4], [1, 4], [2, 4], [1, 2, 4], [3, 4], [1, 3, 4],
 [2, 3, 4], [1, 2, 3, 4]]
```

On proposera une fonction récursive dont le principe est le suivant. On se donne une liste $E = [\![1, n]\!]$, les parties de E sont de deux types différents :

- celles ne contenant pas n, ce sont donc les parties de $[\![1, n-1]\!]$ (c'est cela qui nous pousse à faire un algorithme récursif).
- celles contenant n, ce sont donc les parties de $[\![1, n-1]\!]$ auxquelles on ajoute l'élément n.

7 Trouver ce que fait la fonction suivante et écrire une version itérative de la fonction en utilisant une boucle *for*.

```
def p(ch):
    # ch est une chaine de caractères
    n = len(ch)
    if n <= 1:
        return(True)
    else:
        return((ch[0] == ch[n - 1]) and (p(ch[1:n-1])))
        # ch[1:n-1] est la chaine de caractères obtenue en enlevant le premier et le dernier caractère
```