Spécialité NSI Terminale
Correction du contrôle n° 3
Mardi 5 décembre 2023

A. Variables locales et globales, effets de bord

1. Il s'affiche :

```
x : 25 y : 250
R : 10 S : 15 T : 225
```

2. Il s'affiche :

```
(1) A : 70
(2) A : 35
```

3. Il s'affiche :

```
x : 40 y : 10
T : 50 V : 400 W : 350
```

4. a. On a les valeurs suivantes :

```
tab1 : [11, 13, 17, 23]
tab2 : [11, 13, 17, 23]
```

b. On a les valeurs suivantes :

```
tab3 : [9, 16, 25, 36]
tab4 : [9, 16, 25, 36, 49]
```

5. a. La fonction `decupler` a produit un effet de bord sur le tableau `T1` passé en argument. Il s'affiche :

```
T1 : [30, 50, 70, 90]
T2 : [30, 50, 70, 90]
```

b. Pour éviter cet effet de bord et obtenir l'affichage souhaité, on peut écrire le code suivant pour la fonction `decupler` :

```
def decupler(tab) :
    tab_res = tab.copy()
    lg = len(tab)
    for i in range(lg) :
        tab_res[i] = 10*tab[i]
    return tab_res
```

B. Récursivité

6. a. La fonction somme est récursive car son corps comporte l'instruction

```
return n + somme(n-1)
```

qui forme un appel récursif, c'est-à-dire une instruction où la fonction s'appelle elle-même.

b. Ce cas et appelé un cas de base.

c. On a les résultats suivants:

* `somme(1)` renvoie 1 ;
* `somme(2)` renvoie 3 ;
* `somme(5)` renvoie 15.

d. En théorie, l'instruction va générer une succession infinie d'appels récursif. En effet chaque appel récursif diminue l'argument de 1. Donc en partant de -1 on n'atteindra jamais le cas de base pour lequel l'argument est égale à 0.
Remarque : En pratique le programme s'interrompra avec un message d'erreur après avoir dépassé un maximum d'appels récursifs.

7. a. Pour obtenir une fonction récursive qui reconnaît les palindromes, on peut compléter le code de la façon suivante :

```
def palindrome(texte) :
    lg = len(texte)
    if lg < 2 :
        return True
    texte_central = texte[1 : lg-1]
    return texte[0] == texte[lg-1] and \
           palindrome(texte_central)
```

b. On peut tester la fonction avec le programme principal suivant :

```
res1 =palindrome("kayak")
res2 =palindrome("1589751")
print(res1) # on attend True
print(res2) # on attend False
```

8. a. On peut compléter la fonction de la façon suivante :

```
def syracuse(n, s0) :
    if n == 0 :
        return s0
    s = syracuse(n-1, s0)
    if s%2 == 0 :
        return s//2
    else :
        return 3* s + 1
```

b. On peut écrire le programme principal suivant :

```
for k in range(10) :
    print(syracuse(k, 7))
```

C. Algorithmes de tri

9. a. On peut écrire la fonction suivante :

```
def tri_selection(tab) :
    lg =len(tab)
    for i in range(lg) :
        k = minTab(tab[i:])
        echanger(tab, k, i)
```

Remarque : Si on suppose que la fonction `minTab` -comporte un deuxième argument `g` qui correspond à l'indice à partir duquel on cherche la plus petite valeur, on pouvait aussi écrire l'instruction `k = minTab(tab, i)`. Les deux réponses étaient acceptées.

On remarque d'autre part que cette fonction `tri_selection` ne comporte pas d'instruction `return ...` car elle effectue un tri en place du tableau passé en argument par effet de bord.

b. On peut écrire la deuxième version suivante :

```
def tri_selection2(tab) :
    tab_res = tab.copy()
    lg =len(tab_res)
    for i in range(lg) :
        k = minTab(tab_res[i:])
        echanger(tab_res, k, i)
    return tab_res
```

c. On peut tester les deux fonctions avec le programme principal suivant :

```
T1 = [8, 6, 2, -7, 21]
T2 = T1.copy()
tri_selection(T1)
T3 = tri_selection2(T2)
print("T1 (trié) :", T1)
print("T2 (non trié) :", T2)
print("T3 (trié) :", T3)
```

10. L'algorithme de tri par sélection est de complexité quadratique. Donc si on multiplie par 5 la taille du tableau à trier, on multipliera par 5 au carré donc par 25, le temps d'exécution du tri. Ainsi le temps d'exécution du tri d'un tableau de 5000 nombres sera d'environ 3*25 ms, soit 75 ms.

11. a. On peut compléter le code de la fonction de la façon suivante :

```
def tri_fusion(tab) :
lg = len(tab)
if lg < 2 :
    return tab
tab1, tab2 = decouper(tab)
tab1 = tri_fusion(tab1)
tab2 = tri_fusion(tab2)
return fusion(tab1, tab2)
```

b. Cette fonction met en œuvre une stratégie de "diviser pour régner" car elle suit les trois phases d'une telle stratégie :

* phase "diviser " avec l'instruction `tab1, tab2 = decouper(tab)`
* phase "régner" " avec les deux instructions
  `tab1 = tri_fusion(tab1)`
  `tab2 = tri_fusion(tab2)`
* phase "combiner" " l'instruction `return fusion(tab1, tab2)`