CORRIGÉ : TP2 - BOUCLES IMBRIQUÉES

1 Premier exemple

1. (a) On trouve `24`.

   (b) Dans la boucle intérieure, on fait n additions et on fait n passages dans cette boucle. Cela donne $\boxed{n^2}$ additions.

   (c) On a $f(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} j = n \sum_{j=0}^{n-1} j$, on peut par exemple écrire :

```
def f(n):
    r = 0
    for j in range(n):
            r = r + j
    return n*r
```

   Cette fonction ne fait que n additions.
   Ou bien encore si on est savant :

```
def f(n):
    return n*n*(n-1)/2
```

2.

```
def somme_double(n,m):
    s = 0
    for i in range(n):
        for j in range(m):
            s += log(i+j**2+1)
    return s
```

3. Pour chaque valeur de i on fait m appels de la fonction `log`. Puisqu'il y a n valeurs de i différentes, on en déduit que à chaque appel de la `somme_double`, `log` est appelée $\boxed{n \times m \text{ fois}}$.

2 Maximums et minimums

1.

```
def maximum(L):
    max = L[0]
    for i in range(1,len(L)):
        if L[i] > max:
            max = L[i]
    return max
```

2.

```
def minmax(LL):
    min = maximum(LL[0])
    n = len(LL)
    for i in range(1,n):
        max = maximum(LL[i])
        if max < min :
            min = max
    return min
```

3. Pour chaque appel `maximum(LL[i]` il y a $n_i - 1$ comparaisons, où n_i est la longueur de la liste `LL[i]`. Pour chacune des listes on fait un appel à `maximum`, et pour toutes sauf la première, une comparaison à `min`. En sommant, on obtient $\sum n_i - 1$ comparaisons.

3 Recherche des deux valeurs d'un tableau les plus proches

3.

```
def valeurs_plus_proches(L):
    n = len(L)
    dref = abs(L[0]-L[1])
    v1, v2 = L[0], L[1]
    for i in range(n):
        for j in range(i+1,n):
            d = abs(L[i]-L[j])
            if d < dref:
                dref = d
                v1, v2 = L[i], L[j]
    return v1, v2
```

4. Pour le premier élément de la liste, la fonction `abs` est appelée $n-1$ fois (on calcule la distance entre le premier élément et chacun des suivants). Ensuite, on calcule la distance du deuxième élément de la liste à tous les suivants, la fonction `abs` est donc appelée $n-2$ fois, et ainsi de suite. Au total, on a donc : $\sum_{k=1}^{n-1}(n-k) = \sum_{i=1}^{n-1} i = \boxed{\frac{n(n-1)}{2} \text{ appels}}$ de la fonction `abs`.
   Le terme dominant est $\frac{n^2}{2}$, on a bien un coût quadratique.

4 Recherche d'un facteur dans un texte

5.

```
def facteur_position(texte,mot,i):
    m = len(mot)
    for j in range(m):
        if mot[j] != texte[i+j]:
            return False
    return True
```

6.

```
def facteur(texte, mot):
    n = len(texte)
    m=len(mot)
    for i in range(n-m+1):
        if facteur_position(texte, mot, i):
            return True
    return False
```

7.

```
def liste_occurrences(texte,mot):
    res = []
    n = len(texte)
    m=len(mot)
    for i in range(n-m+1):
        if facteur_position(texte, mot, i):
            res.append(i)
    return res
```

8. Pour chaque indice entre 0 et `n-m`, on fait entre 1 et m comparaisons. Au maximum, on a donc $\boxed{(n-m)\times m \text{ comparaisons}}$.

9.

```
def facteur2(texte, mot):
    n = len(texte)
    m=len(mot)
    for i in range(n-m+1):
        if texte[i:i+m] == mot:
            return True
    return False

def liste_occurrences2(texte,mot):
    res = []
    n = len(texte)
    m=len(mot)
    for i in range(n-m+1):
        if texte[i:i+m] == mot:
            res.append(i)
    return res
```

5 Tri à bulles

10. Appliquer « à la main » l'algorithme à la liste $[5,1,2,3,4,5]$.
Premier passage :
$[5,1,2,3,4,5]$, on compare 5 et 1 et on les échange car $5>1$;
$[1,5,2,3,4,5]$, on compare 5 et 2 et on les échange car $5>2$;
$[1,2,5,3,4,5]$, on compare 5 et 3 et on les échange ;
$[1,2,3,5,4,5]$, on compare 5 et 4 et on les échange ;
$[1,2,3,4,5,5]$, on compare 5 et 5 et on ne fait rien ;
$[1,2,3,4,5,5]$, fin du premier passage.
Deuxième passage :
$[1,2,3,4,5,5]$, on compare 1 et 2 et on ne fait rien ;
$[1,2,3,4,5,5]$, on compare 2 et 3 et on ne fait rien ;
$[1,2,3,4,5,5]$, on compare 3 et 4 et on ne fait rien ;
$[1,2,3,4,5,5]$, on compare 4 et 5 et on ne fait rien ;
$[1,2,3,4,5,5]$, fin du deuxième passage.
Troisième passage :
$[1,2,3,4,5,5]$, on compare 1 et 2 et on ne fait rien ;
$[1,2,3,4,5,5]$, on compare 2 et 3 et on ne fait rien ;
$[1,2,3,4,5,5]$, on compare 3 et 4 et on ne fait rien ;
$[1,2,3,4,5,5]$, fin du troisième passage.
Quatrième passage :
$[1,2,3,4,5,5]$, on compare 1 et 2 et on ne fait rien ;
$[1,2,3,4,5,5]$, on compare 2 et 3 et on ne fait rien ;
$[1,2,3,4,5,5]$, fin du quatrième passage.
Cinquième passage :
$[1,2,3,4,5,5]$, on compare 1 et 2 et on ne fait rien ;
$[1,2,3,4,5,5]$, fin du cinquième passage.

On remarque que les $2^e, 3^e, 4^e$ et 5^e passages n'ont pas été utiles...

11. On propose deux versions :

```
def echanger(L,i,j):
    # La fonction echange deux elements aux indices i et j d'une liste L
    temp = L[i]
    L[i] = L[j]
    L[j]=temp
  #  return L est inutile car L est passe en reference. Si on modifie un
  #argument d'entree mutable (liste, dictionnaire, deque, tableau numpy)
  #dans une fonction alors on retrouve pas l'etat
  # initial de l'objet lorsque que l'on quitte la fonction.
  #Voir le cours au second semestre sur les effets de bord.
  #La variable L dans notre fonction est un objet mutable,
  #il est donc inutile de retourner la variable L.
```

ou bien :

```
def echanger(L,i,j): #la meme en utilisant un tuple
    L[i] , L[j] = L[j], L[i]
```

12.

```
def tri_bulles(L):
    # la fonction trie par ordre croissant la liste L
    # attention : une liste python est un parametre passe par reference.
    n = len(L)
    for i in range(n-1,0,-1): # i varie entre n-1 et 1 en decroissant
        for j in range(1,i+1): # j varie entre 1 et i
            if L[j-1]>L[j]:
                echanger(L,j-1,j)
```

On le teste sur une liste aléatoire de nombres entiers :

```
list = random.sample(range(0, 100), 10)
print(list)
tri_bulles(list)
print(list)
```

13. Le raisonnement est le même que pour la recherche des deux valeurs d'un tableau les plus proches : on fait $\boxed{\frac{n(n-1)}{2} \text{ comparaisons}}$.
14. Il faudrait ne pas poursuivre l'algorithme si la liste est triée. On va donc mettre un drapeau `triee` qui sera à `True` si aucun échange n'a eu lieu et donc si la liste est triée.
15. On utilise une boucle `while` :

```
def tri_bulles2(L):
    n = len(L)
    triee = False
    l = n
    while l>=2 and not triee:
        triee = True
        for j in range(1,l):
            if L[j-1]>L[j]:
                echanger(L,j-1,j)
                triee = False
        l = l-1
```

16. Dans le meilleur des cas, si la liste est triée au départ, on fait un seul parcours de la liste et donc $n-1$ comparaisons, avec n la longueur de la liste.
Dans le pire des cas, si la liste est triée dans l'ordre inverse, on doit aller au bout de l'algorithme, on fait donc $\frac{n(n-1)}{2}$ comparaisons.
17. Pour comparer les temps d'exécution des deux fonctions, on utilise le programme suivant :

```
#comparaison des deux versions en temps de calcul
L = random.sample(range(0, 100000), 10000)
L2 = L.copy()
start = time.time()
tri_bulles(L)
end = time.time()
elapsed = end-start
print("temps d'execution premiere methode : ", elapsed, "s")

start = time.time()
tri_bulles2(L2)
end = time.time()
elapsed = end-start
print("temps d'execution deuxieme methode : ", elapsed, "s")
```

Quand on exécute, on obtient :

```
temps d'execution premiere methode :  8.024721145629883 s
temps d'execution deuxieme methode :  8.205002784729004 s
```

L'optimisation semble en fait contre-productive.

18. L'invariant de boucle est ici "la liste `L[i:n]` est triée".