Algorithmique

Complexité temporelle

Fiche d'exercices

Compter les opérations

Exercice 1 — Séquence sans boucle — 1

Compter le nombre d'opérations élémentaires effectuées par la fonction suivante. On détaillera chaque ligne.

Code à analyser

```
def perimetre_rectangle(l,
↪   h):
    p = 2 * l + 2 * h
    return p
```

Réponse

.....

.....

.....

.....

.....

.....

Exercice 2 — Séquence sans boucle — 2

Compter le nombre d'opérations élémentaires effectuées par la fonction suivante.

Code à analyser

```
def volume_boite(a, b, c):
    s1 = a * b
    s2 = s1 * c
    demi = s2 / 2
    return demi
```

Réponse

.....

.....

.....

.....

.....

.....

.....

Boucles simples

Exercice 3 — Calcul du produit

La fonction suivante calcule le produit de tous les éléments d'une liste `t` de longueur n.

Code à analyser

```
def produit(t):
    p = 1
    for i in range(len(t)):
        p = p * t[i]
    return p
```

1. Quel est le coût de l'initialisation `p = 1` ?

   .....

2. Combien d'opérations sont effectuées à chaque itération ? (On détaillera : accès, multiplication, affectation.)

   .....

   .....

3. Donner le coût total $T(n)$ en fonction de n.

   .....

4. Quelle est la complexité (notation grand O) ?

   .....

Exercice 4 — Recherche d'un élément

La fonction suivante cherche si la valeur `val` est présente dans la liste `t` de longueur n.

Code à analyser

```
def recherche(t, val):
    for i in range(len(t)):
        if t[i] == val:
            return True
    return False
```

1. Combien d'opérations coûte le test `if t[i] == val` dans le pire cas ?

   .....

2. Dans quel cas la boucle effectue-t-elle le **maximum** d'itérations ?

   .....

3. Donner $T(n)$ dans le pire cas, puis la complexité.

   .....

4. Que se passe-t-il dans le **meilleur cas** ? Combien d'itérations ?

   .....

Boucles imbriquées

Exercice 5 — Table de multiplication

La fonction suivante affiche la table de multiplication pour les entiers de 1 à n.

Code à analyser

```
def table(n):
    for i in range(1, n + 1):
        for j in range(1, n +
        ↪ 1):
            print(i * j)
```

1. Combien de fois le corps de la boucle interne est-il exécuté au total ?

   .....

2. Donner $T(n)$ puis la complexité.

   .....

3. Si $n = 100$, combien d'opérations sont effectuées ?

   .....

Exercice 6 — Tri par sélection

On donne ci-dessous l'algorithme du tri par sélection, où `t` est une liste de longueur n :

Code à analyser

```
def tri_selection(t):
    for i in range(len(t) -
    ↪ 1):
        imin = i
        for j in range(i + 1,
        ↪ len(t)):
            if t[j] <
            ↪ t[imin]:
                imin = j
        t[i], t[imin] =
        ↪ t[imin], t[i]
```

1. Pour $i = 0$, combien de fois la boucle interne s'exécute-t-elle ?

   .....

2. Pour $i = 1$? Pour $i = k$ (en général) ?

   .....

3. Le nombre total d'itérations de la boucle interne est $(n-1) + (n-2) + \cdots + 1 = \dfrac{n(n-1)}{2}$. Quelle est la complexité de cet algorithme ?

   .....

De $T(n)$ à O

Exercice 7 — Fonction de coût

Pour chacune des fonctions de coût suivantes, donner la complexité en notation O.

1. $T(n) = 5n + 12$

   .....

2. $T(n) = 3n^2 + 7n + 4$

   .....

3. $T(n) = 42$

   .....

4. $T(n) = 100n^2 + 10\,000n$

   .....

5. $T(n) = n^3 + n^2 + n + 1$

   .....

Exercice 8 — Complexité d'un code

Pour chacune des fonctions suivantes, identifier la complexité sans calculer $T(n)$ exactement. On justifiera brièvement.

Fonction A

```
def f_a(t):          # t de
↪  longueur n
    return t[0] + t[-1]
```

.....

Exercice 9 — Suite

Fonction B

```
def f_b(n):
    total = 0
    for i in range(n):
        for j in range(n):
            for k in
            ↪  range(n):
                total = total
                ↪  + 1
    return total
```

.....

Fonction C

```
def f_c(t):          # t de
↪  longueur n
    s = 0
    for i in range(len(t)):
        s = s + t[i]
    for i in range(len(t)):
        s = s * t[i]
    return s
```

.....

.....

Comparer deux algorithmes

Exercice 10 — Deux façons de vérifier qu'une liste est triée

On donne deux algorithmes qui testent si une liste `t` de longueur n est triée par ordre croissant.

Algorithme 1

```
def est_trie_v1(t):
    for i in range(len(t) - 1):
        if t[i] > t[i + 1]:
            return False
    return True
```

Algorithme 2

```
def est_trie_v2(t):
    for i in range(len(t)):
        for j in range(i + 1,
        ↪ len(t)):
            if t[i] > t[j]:
                return False
    return True
```

1. Quelle est la complexité dans le pire cas de l'algorithme 1 ?

   .....

2. Quelle est la complexité dans le pire cas de l'algorithme 2 ?
   Indice : même raisonnement que pour le tri par sélection.

   .....

3. Lequel préférer ? Justifier.

   .....

   .....

4. Dans quel cas les deux algorithmes s'arrêtent-ils très rapidement (bien avant d'avoir parcouru toute la liste) ?

   .....