Complexité : mesurer l'efficacité d'un algorithme

Activité – Première NSI

1 Pourquoi s'intéresser à l'efficacité ?

Exercice 1 *Deux programmes, un même problème*

Alice et Bob écrivent chacun un programme pour trouver le maximum d'une liste de n éléments.

Programme d'Alice :

```
def maximum_alice(liste):
    m = liste[0]
    for e in liste:
        if e > m:
            m = e
    return m
```

Programme de Bob :

```
def maximum_bob(liste):
    for e in liste:
        est_max = True
        for f in liste:
            if f > e:
                est_max = False
        if est_max:
            return e
```

1. Vérifiez que les deux programmes donnent le même résultat sur la liste [3 ; 8 ; 1 ; 5].
2. Combien de comparaisons effectue le programme d'Alice pour une liste de n éléments ?
3. Combien de comparaisons effectue le programme de Bob dans le pire des cas ? *Indication : pour chaque élément, Bob compare avec tous les autres.*
4. Complétez le tableau :

Taille n	**10**	**100**	**1 000**	**1 000 000**
Alice (comparaisons)				
Bob (comparaisons)				

5. En supposant qu'une comparaison prend 1 microseconde (10^{-6} s), quel est le temps d'exécution de chaque programme pour $n = 1\,000\,000$?
6. Qu'en concluez-vous ?

② Compter les opérations élémentaires

Définition 1 **(Complexité en temps)** La **complexité en temps** d'un algorithme est le nombre d'opérations élémentaires (comparaisons, affectations, opérations arithmétiques, accès mémoire) qu'il effectue, exprimé en fonction de la taille n des données d'entrée.

Remarque 1. On évalue la complexité **indépendamment de la machine** : on ne mesure pas un temps en secondes, mais un nombre d'opérations. Cela permet de comparer les algorithmes sans se soucier de la vitesse du processeur.

Exercice 2 *Compter les opérations*

Pour chaque fonction, déterminez le nombre d'opérations élémentaires effectuées en fonction de n.

Fonction 1 : somme des éléments d'une liste de taille n.

```
def somme(liste):
    s = 0                    # 1 affectation
    for i in range(len(liste)):  # n tours
        s = s + liste[i]    # 1 acces + 1 addition + 1 affectation
    return s
```

1. Combien d'opérations sont effectuées à chaque tour de boucle ?
2. Combien d'opérations au total (en comptant l'initialisation) ?

Fonction 2 : vérifier si une liste contient un doublon.

```
def a_doublon(liste):
    n = len(liste)
    for i in range(n):
        for j in range(i + 1, n):
            if liste[i] == liste[j]:
                return True
    return False
```

3. Dans le pire des cas (aucun doublon), combien de fois la comparaison est-elle effectuée ? *Indication : c'est la même somme que pour les tris naïfs.*

Fonction 3 : division successive par deux.

```
def nb_divisions(n):
    compteur = 0
    while n > 1:
        n = n // 2
        compteur = compteur + 1
    return compteur
```

4. Combien de tours de boucle effectue cette fonction pour $n = 8$? Pour $n = 16$? Pour $n = 1024$?
5. Pour un n quelconque, combien de tours de boucle ? *Indication : combien de fois peut-on diviser n par 2 ?*

Remarque 2 (Le logarithme en deux mots). Le logarithme en base 2, noté $\log_2(n)$, est simplement le nombre de fois que l'on peut diviser n par 2 avant d'atteindre 1. Par exemple :

— $\log_2(8) = 3$ car on divise trois fois par 2 (8, puis 4, puis 2, puis 1) ;
— $\log_2(1024) = 10$ car $1024 = 2^{10}$;
— $\log_2(10^6) \approx 20$;
— $\log_2(10^9) \approx 30$.

On peut aussi voir $\log_2(n)$ comme l'exposant tel que $2^{\log_2(n)} = n$.
C'est tout ce qu'il faut savoir pour ce chapitre. La notation $O(\log n)$ désigne les algorithmes dont le coût croît comme cette quantité, c'est-à-dire très lentement : $4\log_2(n) + 1 = O(\log n)$.

③ La notation O (« grand O »)

Définition 2 **(Notation O)** La notation O exprime un **ordre de grandeur** du nombre d'opérations. On ne retient que le terme qui croît le plus vite, en ignorant les constantes multiplicatives et les termes de moindre degré.

Exemple 1 (Simplification).

a) $3n + 5 = O(n)$ b) $\dfrac{n(n-1)}{2} = O(n^2)$ c) $7n^2 + 3n + 12 = O(n^2)$

Exercice 3 *S'entraîner à simplifier*

Donnez la complexité en notation O pour chaque expression :

a) $5n + 2$
b) $3n^2 + 100n - 7$
c) $n^3 + 5n^2 + 2$
d) $2 + 3n\log(n)$
e) $8n + 2^n$
f) $\dfrac{n}{2} + 10$

④ Les grandes classes de complexité

Notation	Nom	Exemple
$O(1)$	Constante	Accès à `liste[i]`
$O(\log n)$	Logarithmique	Recherche dichotomique
$O(n)$	Linéaire	Recherche séquentielle
$O(n \log n)$	Quasi-linéaire	Tris efficaces (tri fusion, au programme de Terminale)
$O(n^2)$	Quadratique	Tri par sélection ou insertion

Exercice 4 *Conséquences pratiques*

On suppose qu'une opération prend 1 microseconde (10^{-6} s).

1. Complétez le tableau suivant :

	$n = 10$	$n = 1\,000$	$n = 10^6$	$n = 10^9$
$O(\log n)$				
$O(n)$				
$O(n^2)$				

Indication : convertissez en unités lisibles (microsecondes, secondes, minutes, heures, jours...). On rappelle que $\log_2(10) \approx 3{,}3$, $\log_2(1\,000) \approx 10$, $\log_2(10^6) \approx 20$ *et* $\log_2(10^9) \approx 30$.

2. Un algorithme en $O(n^2)$ met 0,5 s pour $n = 1\,000$. Combien de temps mettra-t-il pour $n = 10\,000$? Pour $n = 100\,000$? Justifiez.
3. Pourquoi dit-on que la complexité quadratique est « problématique » pour les grandes données ?

5 Identifier la complexité d'un programme

Propriété 1 **(Règles pratiques)**

— Une boucle simple de n tours donne une complexité en $O(n)$.

— Deux boucles imbriquées de n tours chacune donnent une complexité en $O(n^2)$.

— Une boucle qui divise n par 2 à chaque tour donne une complexité en $O(\log n)$.

— Si un algorithme contient deux parties successives de complexités $O(f)$ et $O(g)$, sa complexité globale est $O(\max(f, g))$.

Exercice 5 *Déterminer la complexité*

Donnez la complexité de chaque fonction :

a)

```
def f1(n):
    s = 0
    for i in range(n):
        for j in range(n):
            s = s + i * j
    return s
```

b)

```
def f2(n):
    s = 0
    for i in range(n):
        s = s + i
    for j in range(n):
        s = s * 2
    return s
```

c)

```
def f3(n):
    i = n
    while i > 1:
        i = i // 2
```

d)

```python
def f4(liste):
    n = len(liste)
    for i in range(n):
        for j in range(i):
            if liste[i] == liste[j]:
                return True
    return False
```

6 Bilan : les algorithmes vus en classe

Exercice 6 *Récapitulatif*

Complétez le tableau en vous appuyant sur les activités précédentes (recherches et tris) :

Algorithme	Complexité (pire cas)	Type
Recherche séquentielle		
Recherche dichotomique		
Tri par sélection		
Tri par insertion (pire cas)		
Tri par insertion (meilleur cas)		

7 Ce qu'il faut retenir

Propriété 2 **(L'essentiel sur la complexité)** La complexité d'un algorithme mesure le nombre d'opérations qu'il effectue en fonction de la taille n des données. La notation O donne un ordre de grandeur en ne retenant que le terme dominant.

Remarque 3. Un algorithme **correct mais lent** peut être inutilisable en pratique. Choisir un bon algorithme, c'est choisir la bonne complexité.