ITC : **Complexité des algorithmes**

I Introduction

Les algorithmes sont des "recettes" informatiques proposées pour résoudre un problème donné. Pour chaque problème, il existe souvent plusieurs algorithmes permettant de le résoudre, certains étant plus "efficaces" que d'autres. Dans ce chapitre, on s'intéresse à la notion de **complexité algorithmique**, qui peut recouvrir deux problématiques :

— la **compléxité temporelle** s'intéresse à étudier la rapidité d'exécution des algorithmes en dénombrant les opérations effectuées en fonction de la taille n des données d'entrée.

— la **compléxité spatiale** s'intéresse à étudier l'espace mémoire occupé par les algorithmes lors de leur exécution.

Exemple ou exercice d'application – *Deux algorithmes pour déterminer les diviseurs d'un nombre*

On propose ci-dessous deux algorithmes afin de déterminer le nombre d de diviseurs de n.

Le second repose sur le fait que si un entier k est une diviseur de n, alors $k' = \dfrac{n}{k}$ l'est également. Comme $kk' = n$, il suffit de chercher les diviseurs $k \leqslant \sqrt{n}$.

```
def diviseurs1(n):
    d = 0
    k = 1
    while k <= n:
        if n % k == 0:
            d += 1
        k += 1
    return d
```

```
def diviseurs2(n):
    d = 0
    k = 1
    while k * k < n:
        if n % k == 0:
            d += 2
        k += 1
    if k * k == n:
        d += 1
    return d
```

Justifier que les algorithmes proposés effectuent la même tâche en commentant. Évaluer le nombre d'opérations élémentaires effectuées par chaque algorithme. Lequel vous semble le plus "efficace" ?

II Complexité temporelle

Étudier la **complexité temporelle** d'un algorithme permet de mesurer son "efficacité" en terme de temps de calcul. Bien sûr, la durée τ d'exécution d'un programme dépend de la "taille" n des données sur lesquelles il est appelé. Par exemple, on s'attend à ce qu'un programme calculant la moyenne d'une liste de nombres ait un temps de calcul d'autant plus long que la liste sera longue.

Définition – Instructions élémentaires

Les instructions élémentaire sont considérées comme ayant un coût constant, indépendant de leurs paramètres. Parmi celles-ci figurent en général :

— les opérations arithmétiques (addition, soustraction, multiplication, division, modulo, partie entière, ...)

— la comparaisons de données (test d'égalité, d'infériorité, ...)

— le transferts de données (lecture et écriture dans un emplacement mémoire)

— les instructions de contrôle (test conditionnel et inconditionnel, appel à une fonction auxiliaire, ...)

Définition – Compléxité temporelle et notation $O(\cdot)$

• La **complexité temporelle** exprime la relation entre la taille n des données et la durée τ d'exécution.

• Le temps d'exécution est évalué de manière **asymptotique**, pour des taille n grandes, en utilisant la notation mathématique $O(\cdot)$, qui explicite une fonction majorante en "ordre de grandeur".

Pour deux fonctions $f(n)$ et $g(n)$, on dit que $f = O(g)$ si f est dominée par g, c'est-à-dire que $f(n) \leqslant K \times g(n)$ où K est une constante numérique.

Remarque : Les instructions **élémentaire** on une complexité **constante** : $O(1)$.

Ex : Un algorithme effectuant :

— $2n+8$ instructions a une complexité $O(n)$.

— $4n^2+3n$ instructions a une complexité $O(n^2)$.

— $5\log(n)+8$ instructions a une complexité $O(\log n)$.

— $2\log(n)+5n$ instructions a une complexité $O(n)$.

O.d.g. : On distingue 6 grandes classes de complexité.

— Complexité **linéaire**, noté $O(n)$: le nombre d'opération est proportionnel à la taille de l'entrée.
Ex : sommer les éléments d'une liste, rechercher les éléments d'une liste.

— Complexité **sous-linéaire**, noté $O(\log n)$: le nombre d'opérations présente une dépendance logarithmique par rapport à la taille de l'argument.
Ex : recherche dichotomique dans un tableau.

— Le temps **quasi-linéaire**, noté $O(n\log n)$: le $\log$ qui apparaît est en base quelconque puisque le résultat diffère d'une constante multiplicative près.
Ex : crible d'Ératosthène pour déterminer les nombres premiers inférieurs à n.

— Complexité **quadratique**, noté $O(n^2)$: le nombre d'opérations possède une dépendance quadratique par rapport à la taille de l'argument.
Ex : algorithmes de tri simples.

— Complexité **polynomiale**, noté $O(n^k)$: le nombre d'opérations possède une dépendance polynomiale à l'exposant k par rapport à la taille de l'argument.
Ex : multiplication de matrices carrés en $O(n^3)$.

— Complexité **exponentielle**, noté $O(c^n)$: le nombre d'opérations possède une dépendance exponentielle par rapport à la taille de l'argument.
Ex : en pratique, inutilisables.

En considérant une fréquence de calcul $f = 1\ \text{GHz} = 10^9\ \text{s}^{-1}$, on peut dresser le tableau des durées de calcul pour différentes tailles n :

n	10	10^2	10^3	10^4	10^5	10^6
$O(\log n)$	1 ns	2 ns	3 ns	4 ns	5 ns	6 ns
$O(n)$	10 ns	100 ns	1 μs	10 μs	100 μs	1 ms
$O(n\log n)$	10 ns	200 ns	3 μs	40 μs	500 μs	6 ms
$O(n^2)$	100 ns	10 μs	1 ms	0,1 s	10 s	10^3 s
$O(n^3)$	1 μs	1 ms	1 s	1000 s	278 h	31 an
$O(2^n)$	1 μs	4.10^{13} an	–	–	–	–

Définition – Complexité dans le pire cas, dans le meilleur cas, en moyenne

Certains algorithmes ont un temps d'exécution qui dépend non seulement de la taille n des données mais également de l'organisation des données elles-mêmes. Dans ce cas on distingue trois sous-types de complexités :

— la complexité **dans le pire des cas** : c'est un majorant du temps d'exécution possible pour toutes les entrées possibles d'une même taille.

— la complexité **dans le meilleur des cas** : c'est un minorant du temps d'exécution possible pour toutes les entrées possibles d'une même taille.

— la complexité **en moyenne** : c'est une évaluation du temps d'exécution moyen portant sur toutes les entrées possible d'une même taille supposées équiprobables.

Exemple ou exercice d'application – *Recherche d'un élément dans une liste*

On propose ci-contre un algorithme qui cherche une valeur `val` dans une liste `liste`.
Évaluer le nombre d'opérations effectuées dans le meilleur cas, dans le pire cas et en moyenne.

```
def recherche(val,liste):
    for elem in liste :
        if elem == val :
            return True
    return False
```

Remarque : au programme, on étudiera toujours la complexité dans le pire cas, qui majore le temps d'exécution, ce qui permet de dimensionner correctement les machines pour faire face à toute éventualité.

III Quelques exemples usuels

III.1 Recherche de `min` et `max`

On propose ci-dessous deux algorithmes permettant de déterminer le minimum et le maximum d'une liste.

```
def minimum(liste):
    """ Recherche du minimum d'une liste"""
    mini = liste[0]
    # Affectation de la première valeur
    for elem in liste:
    #Boucle sur tous les élèments
        if elem < mini:
        #Test d'inégalité sur l'élément
            mini = elem
            #Réaffectation du minimum
    return mini
```

```
def maximum(liste):
    """ Recherche du maximum d'une liste"""
    maxi = liste[0]
    # Affectation de la première valeur
    for elem in liste:
    #Boucle sur tous les élèments
        if elem > maxi:
        #Test d'inégalité sur l'élément
            maxi = elem
            #Réaffectation du maximum
    return maxi
```

Pour ces deux algorithmes, on notera n le nombre d'éléments de la liste.

— La première affectation compte pour 1 opération.

— Pour chaque élément de la boucle, il y a 2 instructions : un test d'inégalité et une affectation le cas échéant.

— La boucle `for` va effectuer n opérations.

Finalement, dans le pire des cas, ces algorithmes effectuent $2n+1$ instructions. On en déduit que leur complexité est en $O(n)$.

III.2 Moyenne

```
def moyenne(liste):
    """ Calcul de la moyenne d'une liste"""
    nbElem = len(liste)
    # Détermination du nombre d'élements
    somme = 0
    #Initialisation
    for val in liste:
    #Boucle sur les valeurs de la liste.
        somme += val
        #On ajoute la valeur
    return somme/nbElem
```

On notera n le nombre d'éléments de la liste.

— Déterminer la longueur d'une liste demande de la parcourir en entier, donc n opérations.

— La première affectation compte pour 1 opération.

— Pour chaque valeur de la boucle, il y a 1 affectation. La boucle `for` va effectuer n opérations.

— Pour finir, on effectuer une division pour 1 instruction.

Finalement, cet algorithme effectue $2n+2$ instructions, sa complexité est donc $O(n)$.

Exemple ou exercice d'application – *Suite de* CÉSARO

Étant donnée une suite de réels (u_n), on appelle suite de CÉSARO associée la suite (v_n) dont chaque terme est égal à la moyenne des n premiers termes de (u_n) : $v_n = \dfrac{u_0 + \cdots + u_n}{n+1}$.

La fonction suivante prend comme argument une liste `u` formée des premiers termes de la suite (u_n) et renvoie la liste `v` correspondante pour la suite (v_n) associée :

```
def cesaro(u) :
    """Calcul de la liste de Césaro associée à~ la liste u"""
    # initialisation de la liste de Césaro vide
    v=[]
    # boucle sur chaque terme àconstruire
    for k in range(len(u)) :
        # calcul de la moyenne des k+1 premiers termes de u
        m=moyenne(u[:k+1])
        # ajout de ce terme dans la liste de Césaro
        v.append(m)
    # retour de la liste de Césaro
    return(v)
```

1. Déterminer la complexité temporelle de la fonction `cesaro` en fonction du dernier indice n de la liste `u`.
2. Écrire une autre version de la fonction `cesaro` ayant une complexité temporelle "significativement meilleure".

III.3 Tri à bulle

On propose ci-dessous l'algorithme du tri à bulle.

```
def tribulle(liste):
    """Tri àbulle d'une liste"""
    nbElem = len(liste)
    # Détermination du nombre d'élements
    for i in range(n-1,0,-1):
    #Parcours décroissant des indices.
        for j in range(i):
        #Parcours la sous-liste.
            if liste[j]>liste[j+1]:
            #Test inégalité consécutifs
                liste[j],liste[j+1]= liste[j+1],liste[j]
                #Échange des valeurs
    return liste
```

On notera n le nombre d'éléments de la liste.

— Déterminer la longueur d'une liste demande de la parcourir en entier, donc n opérations.

— Il y a deux boucles imbriquées. La boucle sur j va effectuer i tests et échanges, donc $2i$ instructions.

— La boucle sur i varie de n à 1. On a donc $\sum_{i=1}^{n} 2i = 2 \times \dfrac{n(n+1)}{2} = n(n+1)$.

Finalement, cet algorithme effectue $n(n+2)$ instructions, sa complexité est donc $O(n^2)$.

III.4 Recherche dichotomique

On propose ci-dessous l'algorithme de la recherche dichotomique d'une valeur dans une liste triée.

```python
def recherchedicho(liste,val):
    """Recherche dichotomique dans une liste triée"""
    nbElem = len(liste) # Détermination du nombre d'élements
    g = 0
    d = nbElem-1
    #Initialisation indices gauche et droite
    while d-g>0 : #Tant que la droite ne rejoint pas la gauche.
        m = int((d+g)/2) #Calcul indice moyen
        if liste[m]<val : #Comparaison àla valeur de indice moyen
            g = m+1 # On affecte le milieu àgauche
        else :
            d = m # sinon, on affecte le milieu àdroite
    # Une fois l'indice proche déterminé, on vérifie si la valeur existe
    if liste[g] == val :
        return True
    else :
        return False
```

On notera n le nombre d'éléments de la liste.

— Initialisation : 2 affectations.

— À chaque itération, 1 test d'inégalité et 1 affectation.

— La taille de l'intervalle $[g, d]$ est divisée par deux à chaque itération de la boucle.

— Ainsi, l'algorithme effectue k itération où k vérifie $2^k = n \Leftrightarrow k = \log_2(n)$.

— Après la boucle, 1 test.

Finalement, cet algorithme effectue $2k+$ instructions, sa complexité est donc $O(n^2)$.

III.5 Généralisation

Propriété – Boucles et complexité

• Un algorithme contenant **une unique boucle** `for` sur n éléments, chaque étape étant constitué d'instructions élémentaires, aura une complexité en $O(n)$.

• Un algorithme contenant **deux boucles** `for` **imbriquées** sur n éléments, chaque étape étant constitué d'instructions élémentaires, aura une complexité en $O(n^2)$.

• Un algorithme **dichotomique**, qui divise la taille des données par 2 à chaque itération, aura une compléxité en $O(\log n)$.