Cours 5 – Révisions de PTSI – Complexité

I Généralités

I.1 Complexité temporelle

I.2 Opérations élémentaires

Exemples : calculs de sommes Ecrivons des fonctions Python pour calculer les sommes suivantes puis déterminons leurs complexités temporelles (on ne comptera ici que les opérations arithmétiques).

1. $\sum_{k=0}^{n-1} k$
2. $\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} ij$
3. $\sum_{i=0}^{n-1}\sum_{j=0}^{i} i+j$

```
def somme1(n):
    S=0
    for k in range(n):
        S=S+k
    return S
```

Nombre d'opérations : n additions

```
def somme2(n):
    S=0
    for i in range(n):
        for j in range(n):
            S=S+i*j
    return S
```

Nombre d'opérations : n^2 additions et n^2 multiplications

```
def somme3(n):
    S=0
    for i in range(n):
        for j in range(i+1):
            S=S+i+j
    return S
```

Nombre d'opérations : $2 \times (1 + 2 + \ldots + n) = n(n+1)$ additions

I.3 Ordres de grandeur

ponentielle, factorielle...)
Par exemple,

— si $C(n) = 2n^2 - 3n + 1$, on pourra écrire $C(n) = O(n^2)$

— si $C(n) = n^2 - n\log(n) + 2^n$, on pourra écrire $C(n) = O(2^n)$

I.4 Complexité en mémoire

II Algorithmes de référence

Moyenne et écart-type

1) Moyenne d'une liste de n nombres

```
def moyenne(L):
    S=0
    for x in L:
        S=S+x
    return S/len(L)
```

Analyse de complexité : n additions, 1 division.
La complexité est linéaire ($C_{\texttt{moyenne}}(n) = O(n)$).

2) Variance/Ecart-type d'une liste de n nombres

```
def variance(L):
    L2=[x**2 for x in L]
    return moyenne(L2)-moyenne(L)**2
```

Analyse de complexité : $n+1$ mises au carré, 1 soustraction, deux appels à `moyenne` sur des listes de taille n.
$C_{\texttt{variance}}(n) = n + 2 + 2 * C_{\texttt{moyenne}}(n) = 3n + 4 = O(n)$. La complexité est linéaire.

```
import numpy as np
def ecarttype(L):
    return np.sqrt(variance(L))
```

Analyse de complexité : 1 appel à `variance` sur une liste de taille n, un calcul de racine carrée.
La complexité est toujours linéaire.

Recherche dans une liste/un tableau

1) Recherche du maximum dans une liste
2) Recherche d'un élément dans une liste
3) Recherche (par dichotomie) d'un nombre dans une liste triée dans l'ordre croissant

```
#renvoie la valeur du maximum dans la liste L
def recherche_max(L):
    max=L[0]
    for x in L:
        if x>max:
            max=x
    return max
```

Analyse de complexité : n comparaisons. La complexité est linéaire.

```
#renvoie True si la valeur x apparaît dans le tableau L, False sinon
def recherche(x,L):
    for e in L:
        if x==e:
            return True
    return False
```

Analyse de complexité :

Dans le meilleur des cas (`x` est le premier élément de `L`) : une seule comparaison. $C(n) = O(1)$. La complexité est constante.

Dans le pire des cas (`x` est le dernier élément de `L`, ou n'appartient pas à `L`) : n comparaisons. $C(n) = O(n)$. La complexité est linéaire.

```
#renvoie True si la valeur x apparaît dans le tableau L trié, False sinon
def recherche_dichotomie(x,L):
    #on cherche x entre les positions g et d
    #g et d sont des indices, pas des valeurs
    g=0
    d=len(L)-1
    #tant qu'il y a encore un endroit où chercher
    while d>=g:
        m=(d+g)//2 # milieu entre g et d
        #on regarde si x est à la position m
        if x==L[m]:
            return True
        #si x est + grand, on cherche dans la moitié droite
        elif x>L[m]:
            g=m+1
        #sinon, on va à gauche
        else:
            d=m-1
    return False
```

Analyse de complexité :

Dans le meilleur des cas (`x` est situé au milieu de `L`) : la complexité est constante.

Dans le pire des cas (`x` n'est pas dans la liste) : on effectue 3 comparaisons par itération. Le nombre d'itérations est environ $\log_2(n)$. La complexité est logarithmique.