Algorithmique et Programmation 1 – TD - TP 7
ALGORITHMIQUE
CORRECTION

Exercice 1 - Calculs élémentaires de complexité

Soit l'algorithme, exprimé en Python, de recherche séquentielle dans une liste triée vu en cours.

```
1  def recherche_sequentielle_liste_triee(liste, elem):
2      """List x Elem --> Bool
3      Vérifie si l'élément elem appartient à la liste triée"""
4      if elem > liste[len(liste) - 1] :
5          return False
6      else :
7          i = 0
8          while liste[i] < elem :
9              i = i + 1
10         if liste[i] == elem :
11             return True
12         else :
13             return False
```

1. Quelle est la mesure de complexité à utiliser pour évaluer la complexité de cet algorithme ?

   — Comparaison de l'élément `elem` à un élément de `liste`. Le nombre de comparaisons dépend de `elem`, de `liste` et de n, la longueur de la liste.
   — Calcul de la longueur de la liste : comme elle n'est calculée qu'une fois, quelle que soit les données du problème, inutile de calculer la complexité correspondante.

2. Calculer la complexité de cet algorithme dans le meilleur cas

   Meilleur des cas : `elem > liste[n-1]`, une seule comparaison

3. Calculer la complexité de cet algorithme dans le pire cas

   Pire des cas : `elem == liste[n-1]` ou `liste[n-2] < elem <liste[n-1]`. $n+2$ comparaisons
   — 1 comparaison avant de rentrer dans la boucle
   — n comparaisons dans la boucle
   — 1 comparaison en fin de boucle

4. Calculer la complexité moyenne de cet algorithme

- Soit $q = p(\texttt{elem} \in \texttt{liste})$, et $1-q = p(\texttt{elem} \notin \texttt{liste})$
- Nombre de comparaisons si `elem` $\in$ `liste` :
  - On suppose que la place de `elem` dans `liste` est équiprobable. Donc $p(\texttt{elem == liste[i]}) = \frac{1}{n}$
  - Si `elem == liste[i]`, il faut faire $i+3$ comparaisons
  - Nombre moyen de comparaisons si `elem` $\in$ `liste` :

$$\frac{1}{n}\sum_{i=0}^{n-1}(i+3) = \frac{1}{n}\sum_{i=1}^{n}(i+2) = \frac{1}{n}\frac{n(n+1)}{2} + 2n = \frac{n+5}{2}$$

- Nombre de comparaisons si `elem` $\notin$ `liste` :
  - $n+1$ cas possibles (`elem <liste[0]; liste[0] < elem <liste[1]; ...; liste[n-2]< elem > liste[n-1]; elem > liste[n-1]`)
  - Il faut faire 1 test si `elem > liste[n-1]` ; et $i+3$ tests si `elem < liste[i]`
  - Nombre moyen de comparaisons si `elem` $\notin$ `liste` :

$$\frac{1}{n+1}(1+\sum_{i=0}^{n-1}(i+3)) = \frac{1}{n+1}(1+\sum_{i=1}^{n}(i+2)) = \frac{1}{n+1} + \frac{n}{2} + \frac{2n}{n+1} \approx_{n\to\infty} \frac{n}{2} + 2$$

- Complexité moyenne :

$$\text{coût } moy(n) = q\frac{n+5}{2} + (1-q)\frac{n+4}{2}$$

Complexité de l'ordre de $\frac{n}{2}$

Exercice 2 - Calculs élémentaires de complexité

Soit l'algorithme suivant, exprimé en Python.

```
1  def mystere(liste, elem):
2      """ """
3      resultat = [0, -1, -1]
4      for i in range(len(liste)):
5          if liste[i] == elem:
6              resultat[0] = resultat[0] + 1
7              resultat[2] = i
8              if resultat[1] == -1:
9                  resultat[1] = i
10     return resultat
```

1. Que fait cet algorithme ? Donnez la chaine de documentation correspondante

```
def nb_occurences(liste, elem):
    """List x Elem --> List
    Retourne une liste contenant le nombre d'apparitions de l'élément
    dans la liste, l'indice de sa première et dernière apparition"""
    resultat = [0, -1, -1]
    for i in range(len(liste)):
        if liste[i] == elem:
            resultat[0] = resultat[0] + 1
            resultat[2] = i
            if resultat[1] == -1:
                resultat[1] = i
    return resultat
```

2. Quelle est la mesure de complexité à utiliser pour évaluer la complexité de cet algorithme ?

   — Comparaison de l'élément `elem` à un élément de `liste`. Le nombre de comparaisons dépend de `elem`, de `liste` et de n, la longueur de la liste.
   — Affectation d'un élément dans `resultat`

3. Calculer la complexité de cet algorithme dans le meilleur cas

   Meilleur cas : `elem` $\notin$ `liste`
   — Comparaisons : n comparaisons de `elem` à `liste[i]`
   — Affectations : 3 affectations en phase d'initialisation

4. Calculer la complexité de cet algorithme dans le pire cas

   Pire des cas : $\forall i \in [1, \ldots, n-1]$, `elem == liste[i]`
   — Comparaisons : n comparaisons de `elem` à `liste[i]`, n comparaisons de `resultat[1]` à `-1` $\Rightarrow 2n$ comparaisons
   — Affectations : 3 affectations en phase d'initialisation, n affectations pour `resultat[0]`, n affectations pour `resultat[2]`, 1 affectation pour `resultat[1]` $\Rightarrow 2n+4$ affectations

5. Calculer la complexité moyenne de cet algorithme

   — Soient $q_0 + q_1 + \ldots + q_n = 1$ avec :
      — q_0 : probabilité que `elem` n'apparaisse pas dans `liste`
      — q_1 : probabilité que `elem` apparaisse 1 fois dans `liste`
      — ...
      — q_n : probabilité que `elem` n'apparaisse n fois dans `liste`
   — Comparaisons :
      — Si `elem` n'apparait pas dans `liste` : n comparaisons
      — Si `elem` apparait 1 fois dans `liste` : $n+1$ comparaisons
      — ...
      — Si `elem` apparait n fois dans `liste` : $2n$ comparaisons
      — Nombre moyen de comparaisons :

$$\text{coût } moy(n) = \sum_{i=0}^{n} (n+i) q_i$$

   — Affectations :
      — Si `elem` n'apparait pas dans `liste` : 3 affectations
      — Si `elem` apparait 1 fois dans `liste` : 6 affectations
      — Si `elem` apparait 2 fois dans `liste` : 8 affectations
      — ...
      — Si `elem` apparait k fois dans `liste` : $4+2k$ affectations
      — Si `elem` apparait n fois dans `liste` : $2n+4$ affectations
      — Nombre moyen d'affectations :

$$\text{coût } moy(n) = 3q_0 + \sum_{i=1}^{n} (2i+4) q_i$$

Exercice 3 - Recherche de nombres

1. Générer un tableau de 100 nombres aléatoires, tous distincts, compris entre 0 et 1000.

```
import random

liste_aleatoire = []
for i in range(100):
    nb = random.randrange(1001)
    while nb in liste_aleatoire:
        nb = random.randrange(1001)
    liste_aleatoire.append(nb)
```

2. Écrire et implémenter l'algorithme permettant de rechercher le nombre maximal de la liste.

```
def max_liste(liste) :
    """Liste --> Int
    Retourne l'élement maximum d'une liste d'entiers"""
    max = liste[0]
    for i in range(1, len(liste)):
        if max < liste[i]:
            max = liste[i]
    return max
```

3. Écrire une fonction permettant de calculer la moyenne des nombres de la liste.

```
def moy_liste(liste) :
    """Liste --> Float
    Retourne la moyenne d'une liste d'entiers"""
    somme = liste[0]
    for i in range(1, len(liste)):
        somme = somme + liste[i]
    return somme/len(liste)
```

4. Ecrire une fonction qui renvoie le nombre d'éléments de la liste strictement inférieurs à la moyenne de la liste.

```
def nombre_inf_moyenne(liste):
    """Liste --> Int
    Retourne le nombre d'éléments inférieurs à la moyenne d'une liste d'entiers"""
    moy = moy_liste(liste)
    nb_inf = 0
    for i in range(len(liste)):
        if liste[i] < moy :
            nb_inf = nb_inf + 1
    return nb_inf
```

5. Ecrire une fonction qui renvoie la liste des carrés des éléments de la liste.

```
def carres(liste) :
    """Liste --> Liste
    Retourne la liste des carrés de la liste d'entrée"""

    liste_carres = []
    for i in range(len(liste)):
        liste_carres.append(liste[i]*liste[i])
    return liste_carres
```

6. La *variance* d'une liste de nombres est égale à la différence entre la moyenne des carrés des éléments de la liste et le carré de la moyenne des éléments de la liste.
Ecrire une fonction qui renvoie la variance de la liste.

```
def variance_liste(liste):
    """Liste --> Float
    Retourne la variance de la liste"""

    liste_carres = carres(liste)
    moy_carre = moy_liste(liste_carres)
    moyenne = moy_liste(liste)
    carre_moyenne = moyenne * moyenne
    variance = moy_carre - carre_moyenne
    return variance
```

7. L'*écart-type* d'une liste de nombres est égal à la racine carrée de la variance de la liste.
Ecrire une fonction qui renvoie l'écart-type de la liste.

```
import math

def ecart_type(liste):
    """Liste --> Float
    Retourne l'écart-type de la liste"""
    var = variance_liste(liste)
    return math.sqrt(var)
```

8. La méthode `mean` du module prédéfini `numpy` permet de calculer la moyenne d'un tableau [1].
La méthode `var` permet elle de calculer la variance d'un tableau [2].
La méthode `std` permet elle de calculer l'écart type d'un tableau [3].
Comparez les résultats que vous obtenez avec ceux obtenus par ces méthodes.

```
import numpy

moyenne = numpy.mean(liste_aleatoire)
variance_numpy = numpy.var(liste_aleatoire)
ecart_numpy = numpy.std(liste_aleatoire)
```

9. Grâce à la méthode `time()` du module prédéfini `time` [4], donner le temps pour faire les calculs de moyenne avec votre fonction et celles de Numpy.

```
import time
# Moyenne
debut = time.time()
moyenne_numpy = numpy.mean(liste_aleatoire)
fin = time.time()
temps_numpy = fin - debut
print("Moyenne numpy = ", moyenne_numpy, "en ", temps_numpy, "secondes")
debut = time.time()
moyenne = moy_liste(liste_aleatoire)
fin = time.time()
temps_moy = fin - debut
print("Moyenne = ", moyenne, "en ", temps_moy, "secondes")
print("Moyenne: différence de temps entre numpy et la fonctions définie :",
 temps_numpy - temps_moy)
```

1. https://docs.scipy.org/doc/numpy/reference/generated/numpy.mean.html
2. https://docs.scipy.org/doc/numpy/reference/generated/numpy.var.html
3. https://docs.scipy.org/doc/numpy/reference/generated/numpy.std.html
4. https://docs.python.org/fr/3/library/time.html#module-time

```
#Variance
debut = time.time()
variance_numpy = numpy.var(liste_aleatoire)
fin = time.time()
temps_numpy = fin - debut
print("Variance numpy = ", variance_numpy, "en ", temps_numpy, "secondes")
debut = time.time()
variance = variance_liste(liste_aleatoire)
fin = time.time()
temps_moy = fin - debut
print("Variance = ", variance, "en ", temps_moy, "secondes")
print("Variance: différence de temps entre numpy et la fonctions définie :",
temps_numpy - temps_moy)

#Ecart-type
debut = time.time()
ecart_numpy = numpy.std(liste_aleatoire)
fin = time.time()
temps_numpy = fin - debut
print("Ecart type numpy = ", ecart_numpy, "en ", temps_numpy, "secondes")
debut = time.time()
ecart = ecart_type(liste_aleatoire)
fin = time.time()
temps_moy = fin - debut
print("Ecart type = ", ecart, "en ", temps_moy, "secondes")
print("Ecart-type: différence de temps entre numpy et la fonctions définie :",
temps_numpy - temps_moy)
```

10. Que conclure sur la complexité temporelle ?

Tout dépend des algorithmes implémentés ! Dans la correction donnée ici, les fonctions implémentées sont plus rapides, donc ont une meilleure complexité temporelle, que les méthodes prédéfinies de numpy.

```
#Moyenne
Moyenne numpy =  504.64 en  7.295608520507812e-05 secondes
Moyenne =  504.64 en  6.9141387939453125e-06 secondes
Moyenne: différence de temps entre numpy et la fonctions définie : 6.604194641113281e-05
#Variance
Variance numpy =  80035.37040000001 en  5.817413330078125e-05 secondes
Variance =  80035.37040000004 en  2.7894973754882812e-05 secondes
Variance: différence de temps entre numpy et la fonctions définie : 3.0279159545898438e-05
#Ecart-type
Ecart type numpy =  282.9052321891555 en  4.8160552978515625e-05 secondes
Ecart type =  282.9052321891556 en  4.1961669921875e-05 secondes
Ecart-type: différence de temps entre numpy et la fonctions définie : 6.198883056640625e-06
```

11. La *médiane* d'une liste de nombre entiers tous différents de longueur paire est la moyenne des valeurs centrales de la liste après classement en ordre croissant.
Par exemple, `mediane([4, 3, 7, 9, 12, 1 ])` = $\frac{4+7}{2}$ = 5,5.
Nous voulons calculer la mediane d'une liste, sans avoir à trier cette liste. Pour cela :

(a) Ecrire une fonction `delta(liste, elem)` qui calcule la différence entre le nombre de valeurs de la liste supérieures et inférieures à `elem`. Par exemple :

```
>>> delta([4,3,7,9,12,1], 3)
3
>>> delta([4,3,7,9,12,1], 4)
1
>>> delta([4,3,7,9,12,1], 7)
-1
```

```
def delta(liste,elem):
    """ Liste * Int --> Int
    Calcul de la différence entre le nombre de valeurs de la liste
    supérieures et inférieures à elem"""
    d=0
    for k in range(len(liste)):
        if liste[k] > elem:
            d = d + 1
        if liste[k] < elem:
            d = d - 1
    return d
```

(b) Ecrire une fonction qui calcule la médiane d'une liste de longueur paire contenant des entiers tous distints.

```
def mediane(liste):
    """ Liste --> Float
    Calcul de la médiane de la liste de nombres distincts liste delongueur paire"""
    s = 0
    for k in range(len(liste)):
        if delta(liste,lite[k]) == -1 or delta(liste,liste[k]) == 1:
            s = s + liste[k]
    return(s/2)
```

(c) Comparez votre résultat avec celui obtenu par la méthode `median` du module Numpy [5]

```
med = mediane(liste_aleatoire)
print("Mediane : ", med)

med_numpy = numpy.median(liste_aleatoire)
print("Mediane Numpy : ", med_numpy)
```

5. https://docs.scipy.org/doc/numpy/reference/generated/numpy.median.html

12. Après avoir généré un nombre aléatoire, écrire l'algorithme qui permet de dire que ce nombre est dans le tableau. Implémenter cet algorithme.

```
nombre = random.randrange(1001)

def recherche_sequentielle_liste_non_triee(liste, elem):
    """List x Elem --> Bool
    Vérifie si l'élément elem appartient à la liste non triée"""
    appartient = False
    i = 0
    n = len(liste)
    while i < n and not(appartient) :
        if liste[i] == elem :
            appartient = True
        i = i + 1
    return appartient
```

13. La méthode `sort` du module prédéfini `numpy` permet de trier un tableau par ordre croissant [6]. Après avoir trié le tableau, écrire l'algorithme qui permet de dire, en utilisant la recherche séquentielle, si un nombre aléatoire est dans le tableau. Implémenter cet algorithme.

```
liste_triee = numpy.sort(liste_aleatoire)

def recherche_sequentielle_liste_triee(liste, elem):
    """List x Elem --> Bool
    Vérifie si l'élément elem appartient à la liste triée"""
    if elem > liste[len(liste) - 1] :
        return False
    else :
        i = 0
        while liste[i] < elem :
            i = i + 1
            if liste[i] == elem :
                return True
            else :
                return False
```

14. Utilisez à présent la recherche dichotomique pour effectuer cette tâche

```
def recherche_dichotomique(liste, elem):
    """List x Elem --> Bool
    Vérifie si l'élément elem appartient à la liste triée"""
    appartient = False
    inf, sup = 0, len(liste) - 1

    while inf <= sup and not(appartient) :
        med = (inf + sup)//2
        if liste[med] == elem :
            appartient = True
        elif liste[med] > elem:
            sup = med - 1
        else :
            inf = med + 1
    return appartient
```

6. https://docs.scipy.org/doc/numpy/reference/generated/numpy.sort.html

15. Comparer le temps de calcul de ces trois algorithmes. Que dire de la complexité de ces algorithmes ?

```
debut = time.time()
appartient = recherche_sequentielle_liste_non_triee(liste_aleatoire, nombre)
fin = time.time()
temps_rech1 = fin - debut

print("Recherche séquentielle liste non triée :")
if appartient :
    print("Le nombre ", nombre, "appartient à la liste, trouvé en ",
    temps_rech1, "secondes")
else:
    print("Le nombre ", nombre, "n'appartient pas à la liste, trouvé en ",
    temps_rech1, "secondes")

debut = time.time()
appartient = recherche_sequentielle_liste_triee(liste_triee, nombre)
fin = time.time()
temps_rech2 = fin - debut

print("Recherche séquentielle liste triée :")
if appartient :
    print("Le nombre ", nombre, "appartient à la liste, trouvé en ",
    temps_rech2, "secondes")
else:
    print("Le nombre ", nombre, "n'appartient pas à la liste, trouvé en ",
    temps_rech2, "secondes")

debut = time.time()
appartient = recherche_dichotomique(liste_triee, nombre)
fin = time.time()
temps_rech3 = fin - debut

print("Recherche dichotomique liste triée :")
if appartient :
    print("Le nombre ", nombre, "appartient à la liste, trouvé en ",
    temps_rech3, "secondes")
else:
    print("Le nombre ", nombre, "n'appartient pas à la liste, trouvé en ",
    temps_rech3, "secondes")
```