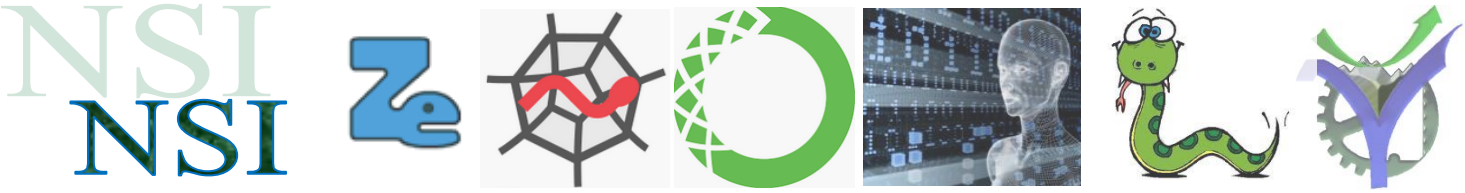


Introduction à la complexité des algorithmes

Résumé :

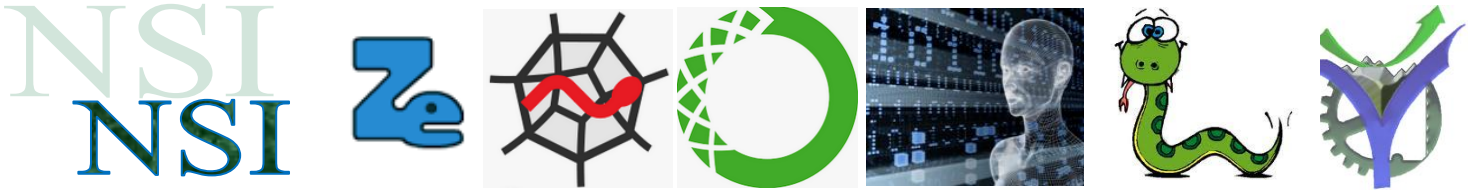
– Utilisation de la complexité algorithmique dans un programme

" "Faites-le d'abord fonctionner. Puis, faites-le bien. Enfin, faites en sorte qu'il aille vite". Ce principe et ses variantes, est considéré comme la règle d'or de la programmation. Il est attribué à Kent Beck, qui lui-même l'attribue à son père."

Alex Martelli, Python en concentré, chapitre 17

Sommaire :

Note : répondre aux questions sur la feuille réponse en annexe.



1 Introduction

L'algorithmique étudie à la fois les algorithmes, c'est à dire l'existence et l'écriture de solutions à des problèmes posés en traitement de l'information, mais également à leurs performances. Ou autrement dit quel algorithme est le plus performant pour résoudre un problème donné.

Les critères de performances sont définis comme par exemple **la rapidité** : si mon algorithme répond avec un certain temps de traitement pour n données d'entrée que se passe t-il pour 2 * n données ?

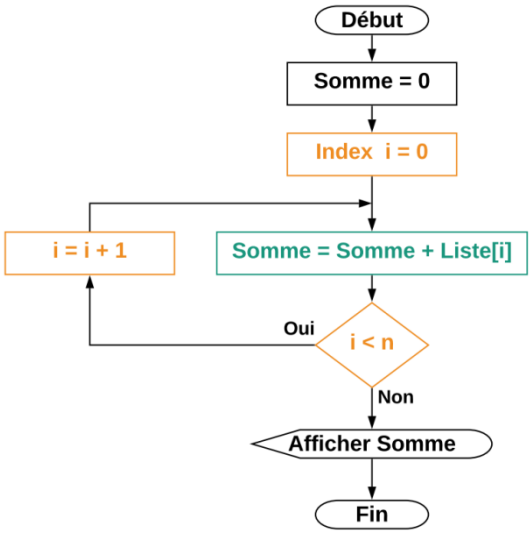
Le critère peut également être **l'occupation mémoire** nécessaire au fonctionnement de l'algorithme ou bien **la bande passante utilisée** via une connexion internet.

La complexité n'est pas à considérer selon le pont de vue humain dans le sens de 'facile à comprendre' mais dans le sens de plus ou moins rapide à produire les résultats attendus en fonction du nombre des entrées.

2 Un premier exemple

Calculons la somme de tous les éléments d'une liste 'Liste' de n nombres. Voilà le pseudo code :

```
Somme <— 0
Pour i parcourant tous les éléments de la liste
Faire
        Somme <— Somme + Liste[i]
Fin pour
Afficher Somme
```



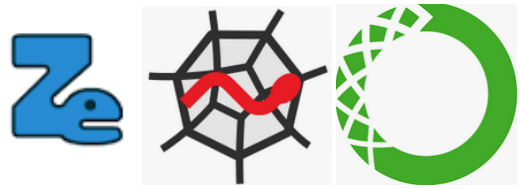
Quel est le nombre d'opérations effectuées ?

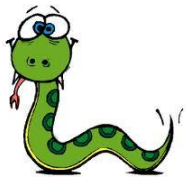
Pour répondre à cette question il faut recenser combien de fois est exécuté chacune des opérations. L'organigramme ci-contre nous permet de bien mettre en évidence le cheminement de l'algorithme en particulier pour la réalisation de la répétition (boucle pour en orange) et nous pouvons faire le bilan :

1 [noir] + 2 * n [Vert] + 2 * n [Orange] + 1 [Orange fin de boucle] + 1 [noir] = 4 n + 3

La complexité de cet algorithme en fonction du nombre n est équivalent quand n est très grand à 4•n donc **il est linéaire en n**. Autrement dit quand le nombre d'entrées double alors la durée de l'algorithme double également.

[1] Tracé avec Lucidchart : https://www.lucidchart.com/pages/fr






Exemple de résultats on observe que la règle obtenue du doublement de la durée de l'algorithme est de plus en plus vérifiée pour n grand.

```
n =      100 valeurs  le temps d'éxécution :   0.000046 secondes
n =      200 valeurs  le temps d'éxécution :   0.000061 secondes
n =      400 valeurs  le temps d'éxécution :   0.000142 secondes
n =     1000 valeurs  le temps d'éxécution :   0.000332 secondes
n =     2000 valeurs  le temps d'éxécution :   0.000716 secondes
n =      400 valeurs  le temps d'éxécution :   0.000122 secondes
n =     1000 valeurs  le temps d'éxécution :   0.000335 secondes
n =     2000 valeurs  le temps d'éxécution :   0.000708 secondes
n =     4000 valeurs  le temps d'éxécution :   0.001314 secondes
n =    10000 valeurs  le temps d'éxécution :   0.003236 secondes
n =    20000 valeurs  le temps d'éxécution :   0.006700 secondes
n =    40000 valeurs  le temps d'éxécution :   0.014664 secondes
n =   100000 valeurs  le temps d'éxécution :   0.034079 secondes
n =   200000 valeurs  le temps d'éxécution :   0.068755 secondes
n =   400000 valeurs  le temps d'éxécution :   0.141120 secondes
n =  1000000 valeurs  le temps d'éxécution :   0.348514 secondes
n =  2000000 valeurs  le temps d'éxécution :   0.739935 secondes
n =  4000000 valeurs  le temps d'éxécution :   1.497242 secondes
```

3 Mesure des durées d'exécution d'une portion de code en Python

La librairie time de python permet cette mesure voilà la manière de procéder :

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import time

# Mesure de la durée : début de la mesure
start_time = time.clock()

# Tâche à réaliser et à chronométrer
#
#




# Mesure de la durée  : fin de la mesure
end_time = time.clock()

# Affichage de la durée de traitement
print("Le temps d'éxécution : %10f secondes" % ( end_time - start_time  ))
```

Le script est ici : Mesure_de_la_duree_en_Python.py

3.1 Analyse pour une liste unique

Le script pour analyser la complexité de l'addition des valeurs d'une liste est ici :

Complexite_boucle_simple.py

Q1. Faites fonctionner ce script. Remplir le tableau et vérifier le coût linéaire de l'algorithme.

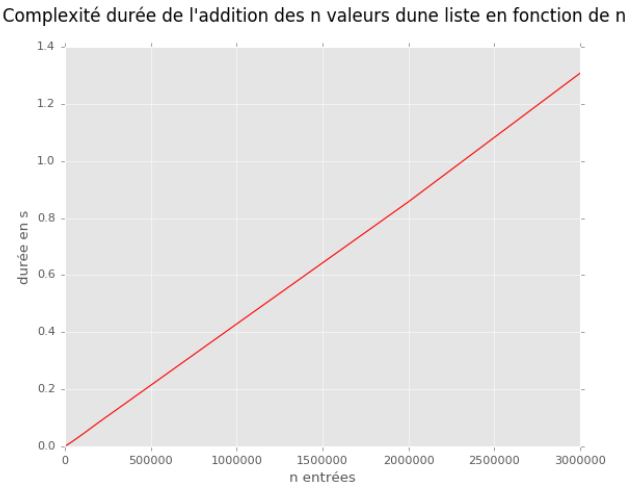
```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import random
import time

# On demande le nombre de valeurs dans la table
nombre_n = input("Indiquez le nombre de valeurs dans la liste : ")
nombre_n = int(nombre_n)

# Initialisation de la table
table_a_traiter = [random.randrange(0,1000000) for i in range(nombre_n)]
longueur = len(table_a_traiter)

# Pour le calcul de la durée de l'algorithme
start_time = time.clock()

# Boucle simple calcul de la somme des éléments
somme = 0
for i in range(longueur):
    somme = somme + table_a_traiter[i]

# Calcul de la durée
duree_time = time.clock() - start_time

# Affichage des résultats
print("n = %8d valeurs \nsomme = %14d \nle temps d'éxécution : %10f secondes"
      % (longueur, somme, duree_time))
```

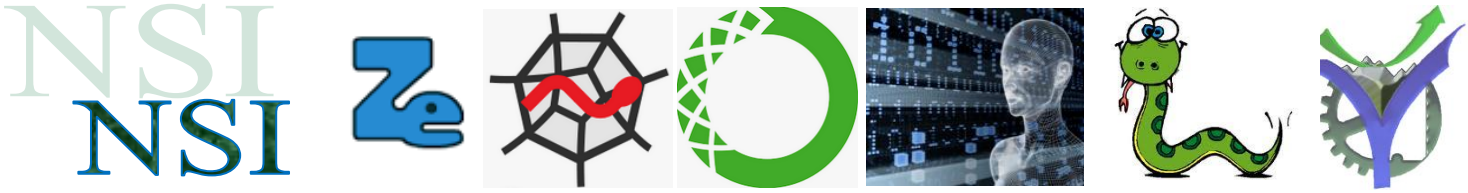
3.2 Analyse pour plusieurs listes avec tracé de courbe

Vous pouvez tester ce script qui fait l'analyse pour un ensemble de valeurs n définies dans une liste.

Pour chacun des calculs le temps mesuré est enregistré ce qui permet le tracé de la courbe directement avec Matplotlib.

Complexite_boucle_multiple.py





4 Étude comparative de trois algorithmes

Nous reprenons l'exemple proposé dans le document cité en référence[2]. Nous cherchons à déterminer si un nombre N est premier avec trois méthodes différentes :

4.1 Proposition de trois méthodes différentes

Méthode A1 : Tester s'il existe un diviseur du nombre N dans l'intervalle 2, N-1.

Méthode A2 : Nous réduisons l'intervalle de recherche en remarquant que l'étude de l'intervalle 2, N/2 est suffisant.

Méthode A3 : Nous réduisons encore l'intervalle en considérant que si N résiste à tout essai de division par les entiers inférieurs ou égaux à racine de N, il est premier[3].

4.2 Comparaison des trois algorithmes étude théorique

Analyse de la complexité de l'algorithme A1 :

```
N_est_premier <— vrai
Pour d parcourant tous les éléments entre 2 et N-1
Faire
        Si ( N est divisible par d )
        Alors
                N_est_premier <— faux
        Fin si
Fin pour
Afficher N_est_premier
```

Pour étudier sa complexité il n'est pas nécessaire d'étudier l'ensemble de l'algorithme, mais de nous concentrer sur la partie la plus consommatrice de temps à savoir **le test de la division de N par d**.

```
N_est_premier <— vrai
Pour d parcourant tous les éléments entre 2 et N-1
Faire
        Si ( N est divisible par d )
        Alors
                N_est_premier <— faux
        Fin si
Fin pour
Afficher N_est_premier
```

Ce test est réalisé **dans une boucle**, pour connaitre le nombre d'itérations de la boucle il suffit de calculer le nombre total d'itérations effectué à savoir : indice_fin - indice_début + 1

[2] http://www.comp.sci.sitew.com/fs/root/491hm-1_introduction_a_la_complexite_des_

[3] Sur les nombres premiers http://villemin.gerard.free.fr/Wwwgvmm/Premier/propriet.htm

Nous obtenons alors :

(N-1) - 2 + 1 = N - 2 opérations

Q2. Déduire de la même manière le coût de l'algorithme A2 en fonction de N.

Q3. Déduire de la même manière le coût de l'algorithme A3 en fonction de N.

Q4. Conclure quant à la rapidité des trois algorithmes quand N est très grand.

4.3 Comparaison des trois algorithmes étude expérimentale

Algorithme A1 : Test_nombre_premier_algo_A1.py

Script_complexité_1. Tester l'algorithme avec les nombres premiers ci-dessous, vous reporterez vos résultats dans le tableau sur la feuille réponse :

```
[17509, 35023, 70061, 140143, 280001, 560017, 1120001]
```

Un exemple de résultat :

```
>>> (executing lines 1 to 26 of "Test_nombre_premier_algo_A1.py")
Entrez le nombre à tester : 1120001
Le nombre  1120001  est premier
Le temps d'éxécution :   0.421184 secondes
```

Script_complexité_2. Modifier le script Python précédent pour tester les performances de l'algorithme A2. Portez vos résultats dans le tableau.

Script_complexité_3. Faire de même pour tester les performances de l'algorithme A3. Portez vos résultats dans le tableau.

Q5. Les résultats expérimentaux sont-ils conforme à la théorie ?

5 Éléments pour l'étude théorique de la complexité

5.1 La notation O (lire Grand O)

Pour la comparaison de la complexité d'algorithmes nous nous plaçons dans le cas le plus défavorable, donc pour les grandes valeurs de N.

Quand nous étudions la complexité algorithmique, ce qui nous intéresse le plus est le comportement asymptotique du comportement de l'algorithme, c'est-à-dire la « forme » de la formule pour n grand. Nous gardons donc seulement le « plus gros » terme de la formule sans sa constante : nous utilisons la notation « grand O » des mathématiques, et disons que l'algorithme est en O(n).

Définition. Soit $g(n)$ une fonction positive. On définit l'ensemble $O(g(n))$ par :

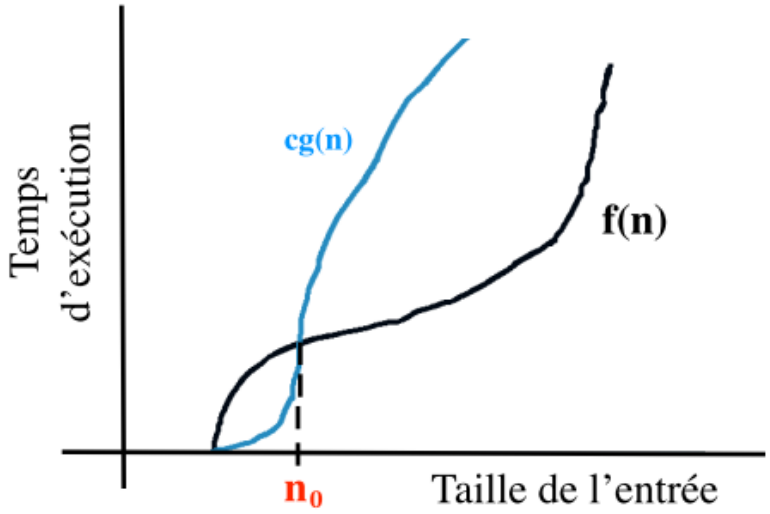
$$O(g(n)) = \{f(n) \mid (\exists\, c > 0), (\exists\, n_0 \geq 0) \text{ tels que } : 0 \leq f(n) \leq c.g(n); (\forall n \geq n_0)\}.$$

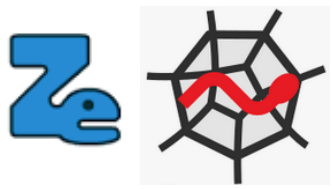
Il s'agit de l'ensemble des fonctions bornées supérieurement par la fonction $g(n)$, à des constantes multiplicatives près. Lorsque $f(n) \in O(g(n))$, on dit que la fonction $g(n)$ est une **borne supérieure asymptotique** pour la fonction $f(n)$, et l'on écrit par raison de simplicité : $f(n) = O(g(n))$.

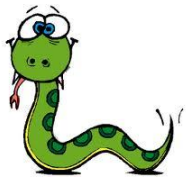
5.2 L'interprétation géométrique

L'interprétation géométrique de f(n) = O(g(n))

Voir (1)








5.3 Comparaison de quelques complexités

Famille d'algorithmes	Notation	Exemples
Algorithmes **constants**	$\Theta(1)$	Echange deux valeurs
Algorithmes **logarithmiques**	$\Theta(\log n)$	Recherche binaire (dichotomique)
Algorithmes **linéaires**	$\Theta(n)$	Recherche séquentielle
Algorithmes **quasi-linéaires**	$\Theta(n \log n)$	Tri par fusion
Algorithmes **quadratiques**	$\Theta(n^2)$	Tri par sélection

Un résultat intéressant à savoir le comportement de l'algorithme quand on double le nombre des entrées (2):

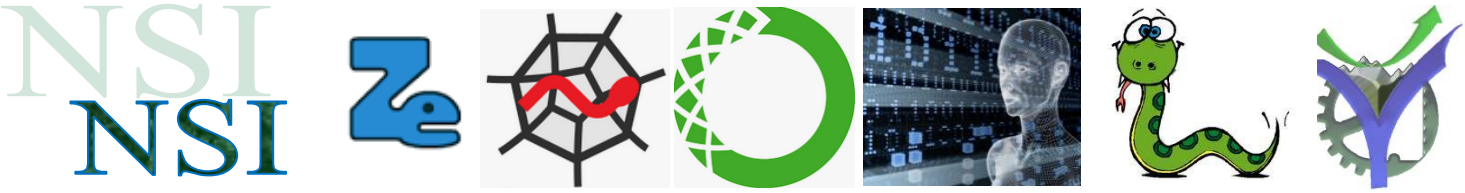
Complexité	Nom courant	Temps quand on double la taille de l'entrée	Max n
$O(n)$	linéaire	prend 2 fois plus de temps	10^{12}
$O(1)$	constant	prend le même temps	pas de limite
$O(n^2)$	quadratique	prend 4 fois plus de temps	10^6
$O(n^3)$	cubique	prend 8 fois plus de temps	10 000
$O(\log n)$	logarithmique	prend seulement une étape de plus	$10^{10^{12}}$
$O(n \log n)$	linearithmique	prend deux fois plus de temps $+ \log n$	10^{11}
$O(2^n)$	exponentiel	prend tellement de temps que c'est inconcevable	30

Dans ces tableaux le log(n) est le $\log_2(n) = \log(n) / \log(2)$. Nous pouvons remarquer que pour les grandes valeurs de n : $\log_2(n)$ est équivalent à log(n) la division par log(2) est négligée.

Nous pouvons réaliser le tracé de quelques complexités :

Comparaison_de_complexite_d_algorithmes.py

Q6. Tester l'algorithme comment justifieriez-vous l'appellation linearithmique au vu du tracé des 4 courbes de complexité $O(n)$, $O(n^2)$, $O(n \cdot \log n)$, $O(\log n)$?



6 Recherche dans une liste triée[4]

Reprenons les deux exemples proposés dans la page Wikipédia traitant de la complexité des algorithmes à savoir retrouver un élément **dans une liste triée**. Cela nous arrive quand on recherche une information dans un dictionnaire par exemple.

6.1 Premier algorithme : **la recherche linéaire**

(Naïve = parcourir tous les éléments de la liste)

S'il y a N éléments dans la liste, la durée de traitement va dépendre de la position de l'élément recherché. Nous voyons que nous pouvons déterminer une durée de traitement **au meilleur cas** (l'élément recherché est le premier sur la liste) ou **au pire cas** (l'élément recherché est au dernier rang de la liste).

Pour l'étude de la complexité des algorithmes nous ne nous intéresserons qu'au pire cas. Pour notre recherche linéaire il nous faudra donc N étapes. La complexité est en $O(n)$.

6.2 Deuxième algorithme : **la recherche dichotomique**.

Voir (3)

Nous allons ici exploiter la propriété que la liste est triée. Cet algorithme demandera dans le pire des cas de séparer en deux la liste, puis de séparer à nouveau cette sous-partie en deux, ainsi de suite jusqu'à trouver l'élément recherché s'il est présent, retourné faux sinon.

Le nombre d'étapes maximales nécessaires sera le nombre entier qui est immédiatement plus grand que $\log_2(n)$.

Une explication possible[5] :

D'autre part, pour déterminer la complexité de l'algorithme, on peut chercher à exprimer le nombre total d'opérations effectuées k (un entier naturel non nul) en fonction de la taille n de l'instance. De par le fonctionnement de la dichotomie, n est divisé par 2 à chaque itération de la boucle de recherche, la taille finale de l'instance est donc $\lfloor n/2^k \rfloor$ (en partie entière). Puisque la plus petite taille possible d'une instance est de 1, on a $1 \leqslant n/2^k$; en multipliant par 2^k (positif) on obtient $2^k \leqslant n$; puis par composition avec le logarithme décimal (qui est croissant) $k \times \log(2) \leqslant \log(n)$; enfin en divisant par $\log(2)$ non nul : $k \leqslant \log(n)/\log(2) = \log_2(n)$. On obtient ainsi une complexité logarithmique.

Définition.
On appelle **fonction logarithme de base 2** la fonction définie sur $]0, +\infty[$ par :

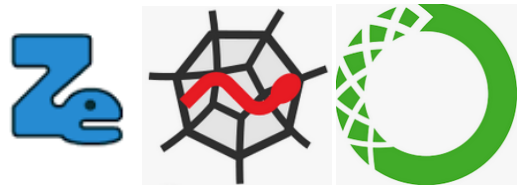
$$\log_2 x = \frac{\ln x}{\ln 2}$$

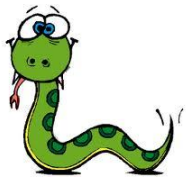
$$\log_2 1 = 0 \quad \text{et} \quad \log_2 2 = 1$$

Plus généralement : $\log_2 2^p = p$ pour tout $p \in \mathbb{Z}$

⇦

[4] https://fr.wikipedia.org/wiki/Analyse_de_la_complexit%C3%A9_des_algorithmes
[5] https://fr.wikipedia.org/wiki/Recherche_dichotomique






Voilà le résultat d'un script python qui vérifie le calcul du nombre d'étapes :

```
Entrez votre nombre n : 125

Le nombre n est égal à  :  125
     log2(n) vaut alors :  6.965784284662088
     il faut dichotomer :  7  fois

Démonstration

 Dichotomie n°   1  nombre =      62.50
 Dichotomie n°   2  nombre =      31.25
 Dichotomie n°   3  nombre =      15.62
 Dichotomie n°   4  nombre =       7.81
 Dichotomie n°   5  nombre =       3.91
 Dichotomie n°   6  nombre =       1.95
 Dichotomie n°   7  nombre =       0.98
```

A vous de jouer :

Script_complexité_4. Compléter le script en python pour montrer le nombre de dichotomie pour diviser successivement un nombre N.

Nombre_etapes_en_cas_de_dichotomie.py

6.3 Pseudo code de l'algorithme de recherche dichotomique

La liste triée s'appelle Liste elle contient N éléments indicé de 0 à N-1 (liste Python). On recherche l'indice de la valeur v.

Explication littérale de l'algorithme :

Les indices Début et Fin donne l'intervalle de recherche au départ maximal. On calcul l'indice milieu découpant la liste en deux parties.

On test ensuite pour savoir dans quel direction se déplacer en fonction de la valeur recherchée v, est-elle dans la première partie ? Ou dans la deuxième ?

Puis on répète la méthode jusqu'à trouver la valeur.

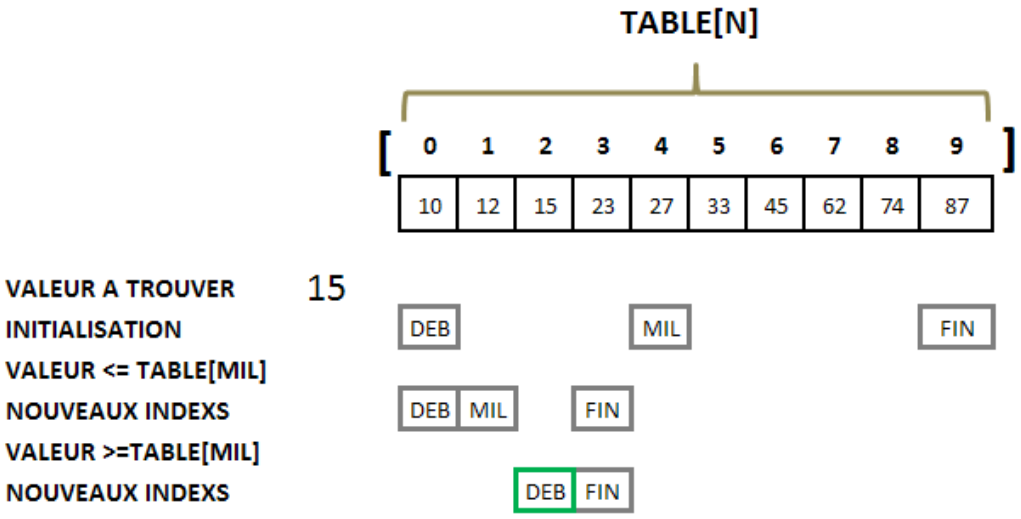
On arrête le processus dans le cas où l'on 'tombe' directement sur la valeur recherchée.

Pseudo code de l'algorithme :

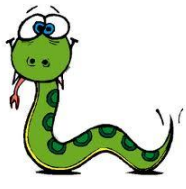
```
début  = 0
fin            = N-1
trouvé         = Faux

Répéter
        milieu = partie entière ( début + ( fin - début )   /  2 )
        Si Liste [ milieu ] = v
        Alors
                trouvé <- Vrai
        Sinon
                Si v >= Liste [ milieu ]
                Alors
                        début <- milieu + 1
                        Si Liste [ début ] = v
                        Alors
                                milieu <- début
                           trouvé <- vrai
                        Fin si
                Sinon
                        fin <- milieu - 1
                        Si Liste [ fin ] = v
                        Alors
                                milieu <- fin
                           trouvé <- vrai
                        Fin si
                Fin si
        Fin si
Jusqu'à trouvé ou début >= fin
```

Faire 'tourner' l'algorithme 'à la main' :

Deux exemples de recherche dichotomique pour comprendre le processus :






TABLE[N]

0	1	2	3	4	5	6	7	8	9
10	12	15	23	27	33	45	62	74	87

VALEUR A TROUVER 74

INITIALISATION DEB MIL FIN

VALEUR >= TABLE[MIL]

NOUVEAUX INDEXS DEB MIL FIN

VALEUR >=TABLE[MIL]

NOUVEAUX INDEXS DEB FIN

Q7. Effectuez la recherche sur les deux exemples ci-dessous (sur la feuille réponse) :

TABLE[N]

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
7	11	23	41	45	66	71	79	82	94	101	111	124	138	142	157	168	190

VALEUR A TROUVER 157

INITIALISATION DEB MIL FIN

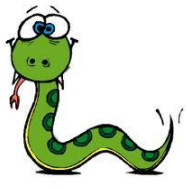
TABLE[N]

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
7	11	23	41	45	66	71	79	82	94	101	111	124	138	142	157	168	190

VALEUR A TROUVER 200

INITIALISATION DEB MIL FIN




Programmation en Python

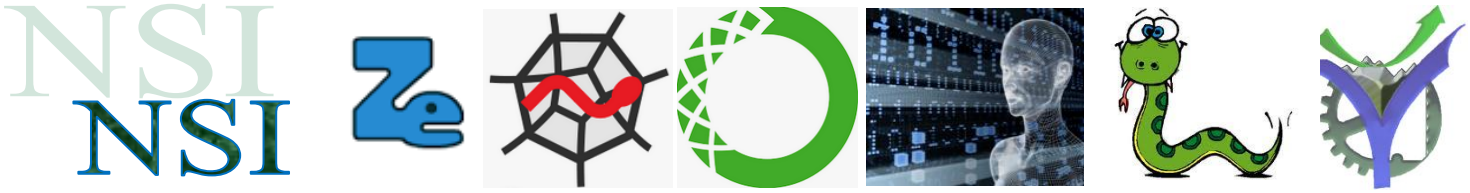
Script_complexité_5. Codez la recherche dichotomique en Python en complétant le script ci-dessous.

Recherche_dichotomique.py

Note : le langage Python n'implémente pas la structure algorithmique répéter … jusqu'à pour la coder il suffit de faire comme ci-dessous, l'instruction break provoquant une sortie immédiate de la structure algorithmique dans laquelle elle est positionnée.

```
# Implémenter un répéter -- jusqu'à (condition) en
# Python

while ( True ) :
        Code à répéter

        if ( condition vrai ) :
                Break
```

```
# Recherche_dichotomique.py

from math import floor
import time

# La liste triée qui servira pour les recherches
liste_triee = [ 7, 11, 23, 41, 45, 66, 71, 79, 82, 94, 101, 111, 124, 138, 142, 157, 168, 190 ]
print("\n\nLa liste : ",liste_triee)
print("Position :  [0,  1,  2,  3,  4,  5,  6,  7,  8,  9,  10,  11,  12,  13,  14,  15,  16,  17]\n")

# Entrée de la valeur à rechercher
element_recherche = int(input("Donner la valeur de l'élément recherché : "))

# Recherche dichotomique

# Initialisation
debut = 0
fin   = len(liste_triee)-1
trouve = False

# Boucle
while ( True ):
    milieu = floor(debut + (fin - debut) / 2)
    if ( liste_triee[milieu] == element_recherche ):
        trouve = True
    else:

    # A COMPLETER



    if (( trouve == True) or (debut >= fin)):
        break

# Affichage du résultat
if ( trouve == True):
    print("Trouvé en position ",milieu," dans la liste")
else:
    print("Pas trouvé")
```



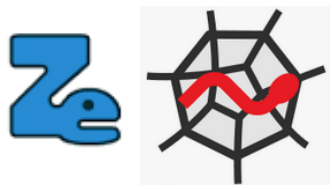
Vérification de la complexité

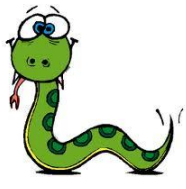
Nous avons vu plus haut que la complexité en log2(n) se vérifie parce qu'une recherche sur une liste deux fois plus longue ne prend qu'une étape de plus, au pire cas.

Q8. A partir de votre script précédent vérifiez que le doublement du nombre des entrées dans la liste ne rajoute qu'une seule étape à la recherche d'une valeur.

Exemple de résultats attendus :

```
La liste :  [7, 11, 23, 41, 45, 66, 71, 79, 82, 94, 101, 111, 124, 138, 142, 157, 168, 190]
Position :  [0,  1,  2,  3,  4,  5,  6,  7,  8,  9,  10,  11,  12,  13,  14,  15,  16,  17]

Donner la valeur de l'élément recherché : 103
Début :  0  Milieu :  8  Fin :  17
Début :  9  Milieu :  13  Fin :  17
Début :  9  Milieu :  10  Fin :  12
Début :  11  Milieu :  11  Fin :  12
Début :  11  Milieu :  11  Fin :  10
Pas trouvé
Avec  4  itérations

>>> (executing lines 1 to 56 of "Recherche_dichotomique_corrigé_verbeux.py")


La liste :  [7, 11, 23, 41, 45, 66, 71, 79, 82]
Position :  [0,  1,  2,  3,  4,  5,  6,  7,  8]

Donner la valeur de l'élément recherché : 72
Début :  0  Milieu :  4  Fin :  8
Début :  5  Milieu :  6  Fin :  8
Début :  7  Milieu :  7  Fin :  8
Début :  7  Milieu :  7  Fin :  6
Pas trouvé
Avec  3  itérations
```






7 En guise de conclusion

D'après (3)

7.1 Quand on met en œuvre un algorithme :

" "Faites-le d'abord fonctionner. Puis, faites-le bien. Enfin, faites en sorte qu'il aille vite". Ce principe et ses variantes, est considéré comme la règle d'or de la programmation. Il est attribué à Kent Beck, qui lui-même l'attribue à son père."

Alex Martelli, Python en concentré, chapitre 17

Avant d'optimiser un code, il faut absolument que le code fonctionne et qu'il soit bien conçu (que l'architecture et sa conception vous conviennent).

Remarque

J-Ch. Arnulfo[25] nous explique que : "L'optimisation d'un programme ne doit pas être l'étape ultime du développement. Les performances se jouent dès la phase de conception et tout au long de l'implémentation."

7.2 Si nous voulons optimiser le code quelques questions préliminaires

Si nous désirons optimiser le code, il faut en fait se poser quelques questions listées par J.-Ch. Arnulfo :

- Est-ce que quelqu'un s'est plaint de la lenteur du logiciel ?
- Est-ce ce bout de code qui pose problème et qui doit être optimisé ? Un profilage sera intéressant à effectuer.
- Est-ce que le gain sera assez significatif pour que l'amélioration soit visible par l'utilisateur ? La fonction en question est-elle assez souvent utilisée ?
- Est-ce que l'optimisation est sans effet de bord ? Pensez à conserver l'ancien code en commentaire pour pouvoir y revenir si nécessaire.
- Est-ce que la lisibilité du code ne va pas être réduite ? En effet si nous écrivons du code avec par exemple des opérations sur bit pour gagner quelques microsecondes celui-ci se complexifie et pourrait devenir illisible. Et cela est source d'ennuis.

7.3 Le profilage d'un programme

Profilage

" Le **profiling** d'un programme permet d'identifier les endroits où celui-ci passe le plus de temps, mais également quelles sont les fonctions qui sont exécutées et combien de fois. Un **profiler** est un programme capable d'analyser un exécutable. Le résultat de l'analyse est appelé un **profile** " (ou profilage). (par M. Idrissi Aouad et O. Zendra[28])

⇦

A. Martelli[29] explique :

"Un profilage met souvent en évidence le fait que les problèmes de performances se situent dans un petit sous-ensemble du code, soit 10 à 20 %, dans lequel votre programme passe 80 à 90 % de son temps. Ces régions cruciales sont également appelées *goulets d'étranglement* ou *points chauds*."

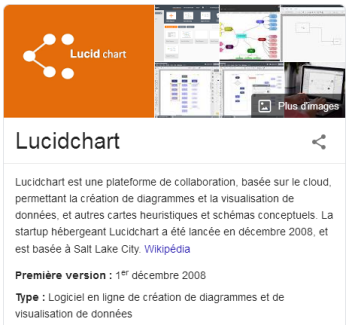
Pour mesurer le temps que prend une partie du code, une solution simple est de doubler (exécuter deux fois) ou de supprimer (ne pas exécuter) cette partie de code.

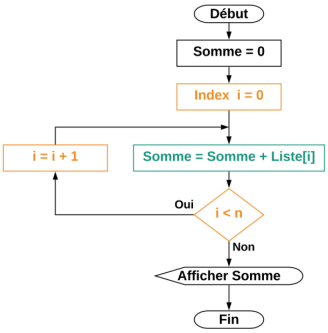
Sinon il est possible de mettre manuellement des chronomètres dans le code.

Enfin, il existe dans certains langages des bibliothèques/modules de profilage. Elles collectent des données lors de l'exécution de votre programme.

8 Bibliographie et ressources

Pour tracer les algorigrammes : https://www.lucidchart.com/pages/fr





1. **HAMEL, Sylvie.** Notation asymptotique.
http://www.iro.umontreal.ca/~hamelsyl/grandO.pdf
[En ligne]

2. **BOUCHEZ TICHADOU, Florent.** *Complexité algorithmique.* 13 septembre 2018.

3. **TRICHET, Eric.** Introduction_complexite_algorithmique.
http://www.irem.unilim.fr/fileadmin/documents/2015_01_04-Introduction_complexite_algorithmique.pdf.
[En ligne]

4. **De SAINT JULIEN**, **Arnaud**. *Complexité et preuves d'algorithmes*. 9 octobre 2018.
http://desaintar.free.fr/python/cours/complexite.pdf
http://desaintar.free.fr/index.php
[En ligne]

Introduction à la complexité feuille réponse

Nom :

Note : / 20 **/ 30**

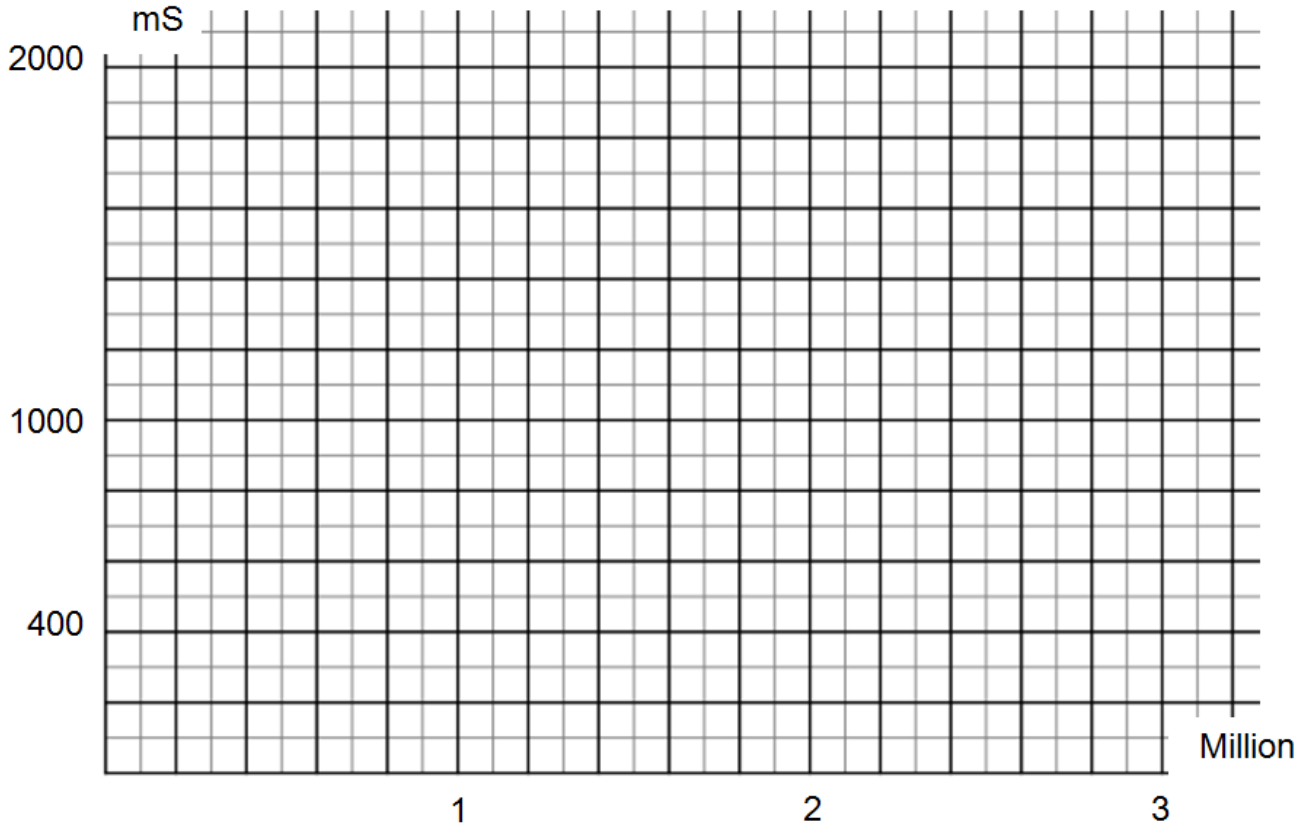
Premier exemple de complexité

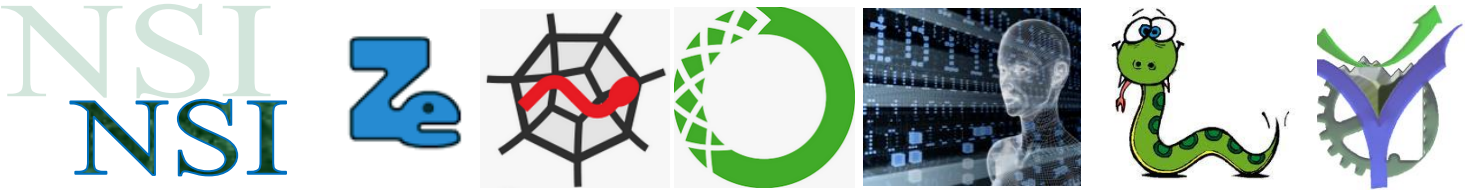
Q1. Vérifier le coût linéaire de l'algorithme. ☐ 2

Longueur de la liste	Durée de traitement mesurée en mS
200000	
400000	
1000000	
2000000	
3000000	

☐ 2

Durée de traitement en fonction du nombre d'éléments dans la liste





Étude comparative de trois algorithmes

Q2. Déduire de la même manière le coût de l'algorithme A2 en fonction de N. ☐ 2

Q3. Déduire de la même manière le coût de l'algorithme A3 en fonction de N. ☐ 2

Q4. Conclure quant à la rapidité des trois algorithmes quand N est très grand. ☐ 2

Comparaison des trois algorithmes étude expérimentale ☐ 2

Nombre premier à tester	Algorithme A1 Script_complexité_1	Algorithme A2 Script_complexité_2	Algorithme A3 Script_complexité_3
17509			
35023			
70061			
140143			
280001			
560017			
1120001			

Q5. Les résultats expérimentaux sont-ils conforme à la théorie ? ☐ 2

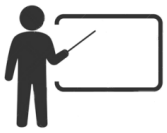
Étude théorique de la complexité

☐ 2

Q6. Tester l'algorithme comment justifieriez-vous l'appellation linearithmique au vu du tracé des 4 courbes de complexité $O(n)$, $O(n^2)$, $O(n \cdot \log n)$, $O(\log n)$?

Recherche dichotomique de nombres dans une liste

Script_complexité_4. Compléter le script en python pour montrer le nombre de dichotomie pour diviser successivement un nombre N.



☐ 4

Q7. Effectuez la recherche sur les deux exemples ci-dessous :

☐ 2

TABLE[N]

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
7	11	23	41	45	66	71	79	82	94	101	111	124	138	142	157	168	190

VALEUR A TROUVER 157

INITIALISATION

DEB								MIL									FIN

☐ 2

TABLE[N]

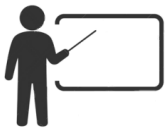
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
7	11	23	41	45	66	71	79	82	94	101	111	124	138	142	157	168	190

VALEUR A TROUVER 200

INITIALISATION

DEB								MIL									FIN

Script_complexité_5. Codez la recherche dichotomique en Python en complétant le script ci-dessous.



4

Q8. A partir de votre script précédent vérifiez que le doublement du nombre des entrées dans la liste ne rajoute qu'une seule étape à la recherche d'une valeur.

2