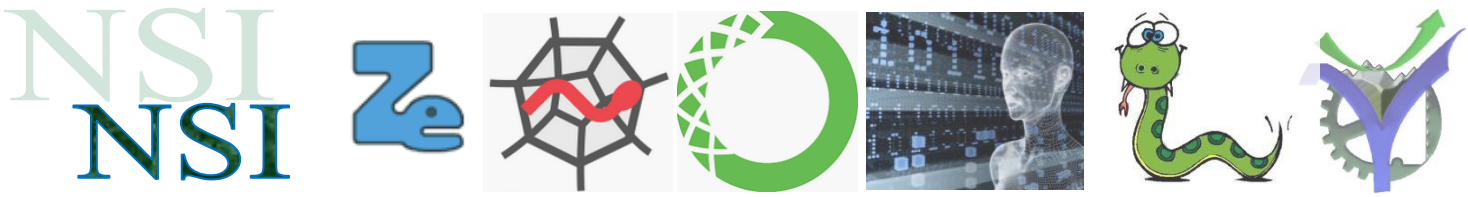
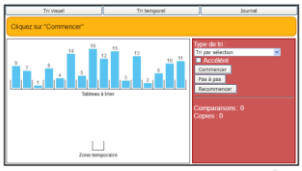


Analyse et mise en œuvre de tris de tables



Résumé :

Un algorithme de **tri** est, en **informatique** ou en mathématiques, un algorithme qui permet d'organiser une collection d'objets selon une relation d'ordre déterminée. Les objets à **trier** sont des éléments d'un ensemble muni d'un ordre total.

Algorithme de tri — Wikipédia
https://fr.wikipedia.org/wiki/Algorithme_de_tri

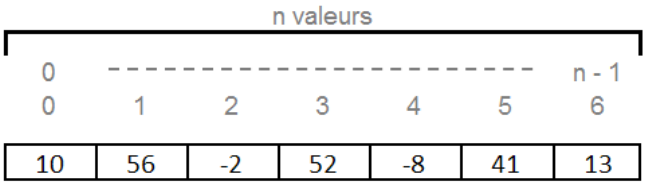
Les réponses aux questions posées seront données dans le document réponse joint.

Sommaire :

1 Le tri par sélection

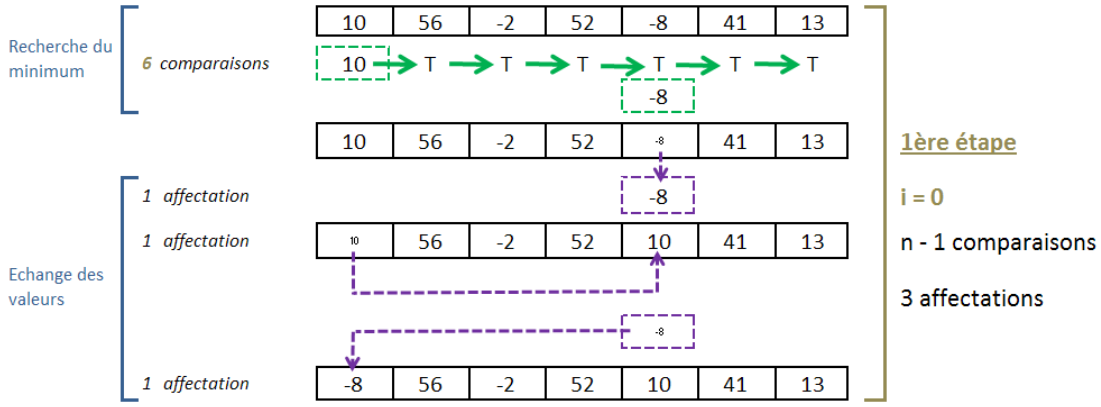
1.1 Découverte de l'algorithme

Pour découvrir l'algorithme analysons le schéma de fonctionnement ci-dessous.



Nous partons d'une table avec n = 7 valeurs :

Première étape :

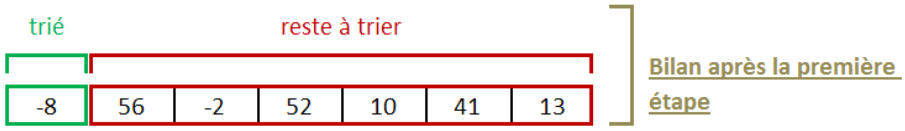


Commentaires :

a) on débute notre travail de tri avec la première case du tableau ici elle contient la valeur 10. Puis on recherche dans les cases restantes la plus petite valeur qui est inférieure à 10. C'est la valeur -8. Cette recherche nous coûte 6 comparaisons.

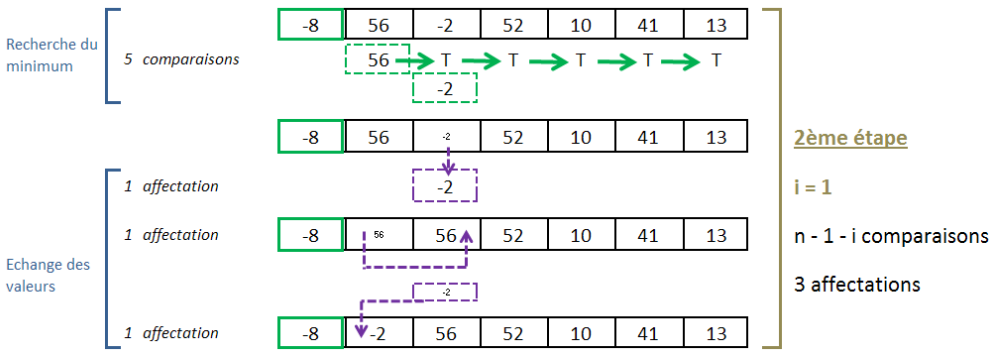
b) si la valeur existe, ici oui c'est -8, alors on procède à l'échange entre les cases du tableau contenant les valeurs 10 et -8, cette opération nous coûte 3 affectations.

Voilà l'état de notre tri après cette première étape :



Deuxième étape du tri :

La première case est triée alors on la laisse de coté et on recommence le même processus que précédemment à partir de la case suivante, ici elle contient 56.

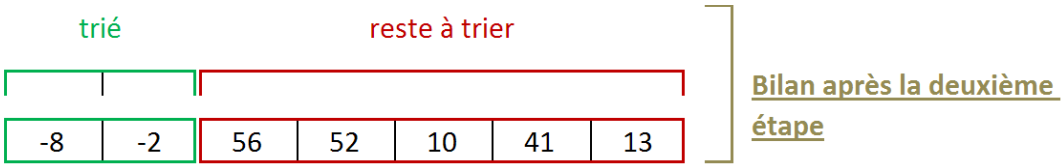


Commentaires :

a) Quelle est la valeur plus petite restante c'est -2 cette recherche nous coûte 5 comparaisons.

b) Il nous faut échanger les valeurs 52 et -2 cela nous coûte 3 affectations.

Voilà l'état de notre tri après la deuxième étape :



Pour terminer le tri il faut recommencer cette procédure pour i = 2 jusqu'à i = n - 2 soit l'avant dernière case. Pour la dernière étape il nous restera à comparer l'avant dernière case du tableau (indice n-2) avec la dernière (indice n-1).

Q1. Faire fonctionner la suite du tri à la main, la première étape a déjà été réalisée, compléter le tableau sur la feuille réponse :

10	56	-2	52	-8	41	13
-8	56	-2	52	10	41	13

1.2 Description de l'algorithme en pseudo code[1]

On considère un tableau t [ ] de n éléments numérotés de 0 à n-1. L'objectif consiste à réaliser le tri par sélection de ce tableau. Voilà l'algorigramme en pseudo code :

```
pour i de 0 à n - 2
    min ← i
    pour j de i + 1 à n - 1
        si t [ j ] < t [ min ], alors
            min ← j
        fin si
    fin pour
    si min ≠ i, alors
        échanger t [ i ] et t [ min ]
    fin si
fin pour
```

[1] Source : https://fr.wikipedia.org/wiki/Tri_par_sélection

1.3 Codage et essai en python

Script1. Coder l'algorithme de tri par sélection en Python en complétant tri par selection depart eleve.py

1.4 Comment remplir une table avec 25000 valeurs aléatoires

Pour remplir une table aléatoirement on fait appel à la bibliothèque random et à une compréhension de liste. L'exemple donné ci-dessous rempli une table de 25000 valeurs chacune étant tirée au sort entre 0 et 1 million.

```
table_a_trier = [random.randrange(0,1000000) for i in range(25000)]
longueur = len(table_a_trier)
```

1.5 Étude expérimentale de la complexité de l'algorithme

Faites fonctionner votre script avec des listes contenant beaucoup de valeurs pour pouvoir ensuite étudier expérimentalement la complexité de l'algorithme. Pour appel la mesure du temps de calcul est indiquée au paragraphe 5.1.

Script2. Faites fonctionner votre algorithme et remplir le tableau suivant :

Tri par sélection en S

n valeurs	500	1000	2000	4000	8000	16000	32000
Durée							

1.6 Tracé de la courbe avec Matplotlib

Pour tracé les résultats de nos analyses il suffit de faire appel à la bibliothèque Matplotlib avec la liste des valeurs à tracer en abscisses et en ordonnées.

Le principe le plus simple est indiqué ci-dessous :

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

import matplotlib
import matplotlib.pyplot as plt

abscisses = [ --- LISTE DES VALEURS DES ABSCISSES --- ]
ordonnees = [ --- LISTE DES VALEURS DES ORDONNEES --- ]
fig, ax = plt.subplots()
ax.plot(abscisses, ordonnees, 'o-')
ax.set(xlabel='Abscisses', ylabel='Ordonnées', title='TITRE')
ax.grid()
plt.show()
```

Script3. Réaliser le script qui trace la courbe montrant l'évolution de la durée du traitement du tri par sélection en fonction du nombre d'éléments de la table.

1.7 Réflexion sur la complexité

a) Étude pratique

La classification de la complexité algorithmique est illustrée d'une manière utilitaire dans la table ci-dessous :

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

Q2. En exploitant vos résultats précédents, donner une indication sur la complexité de l'algorithme de tri par sélection.

b) Étude théorique[2]

Nous observons que notre algorithme est organisé en deux boucle for imbriquées. Pour réaliser l'étude théorique de sa complexité nous allons l'étudier en considérant l'opération de comparaison :

si t [j] < t [min] , alors min ← j

La première boucle fonctionnant avec l'indice i est exécutée (n - 2) - 0 + 1 ⇨ soit n - 1 fois.

La deuxième boucle fonctionnant sur l'indice j est exécutée de (n - 1) - (i + 1) + 1 ⇨ Soit (n - i - 1) fois.

L'opération de comparaison est à l'intérieur de la deuxième boucle elle est donc exécutée :

- Pour le premier passage i = 0 ⇨ (n - 1) fois
- Pour le passage suivant i = 1 ⇨ (n - 2) fois

  et ainsi de suite ……
- Jusqu'au dernier passage où i = n - 2 ⇨ (n - (n - 2) - 1) = 1

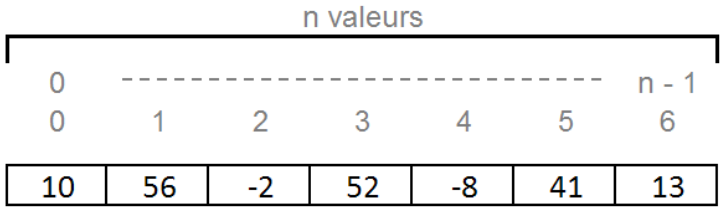
Le nombre total de test est égal à la somme (n-1) + (n-2) + (n-3) + ---------- + 1 = n (n - 1) / 2 (Somme des n-1 premiers entiers). La complexité de l'algorithme est en $O(n^2)$.

[2] En complément voir l'étude ici : https://rmdiscala.developpez.com/cours/LesChapitres.html/Cours4/TPSchap4.6.htm.

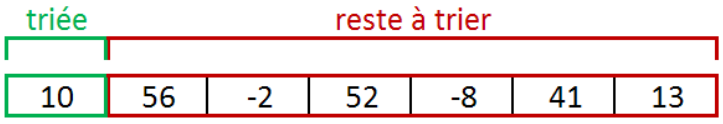
2 Le tri par insertion

(Le tri des joueurs de cartes)

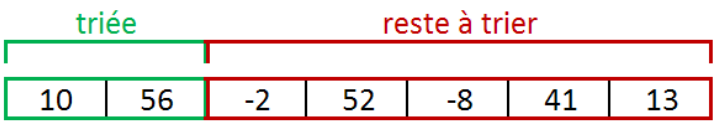
La situation de départ une liste de valeurs à trier, ici 7.



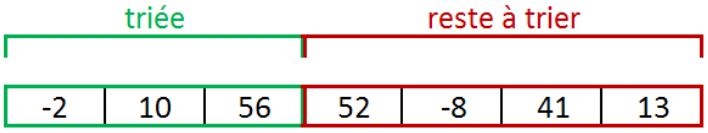
La première étape consiste à couper la liste en deux parties :



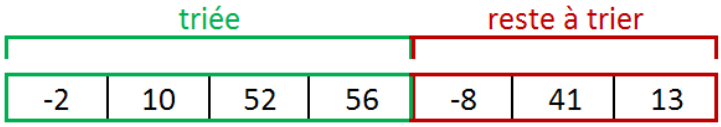
Ensuite on prend les éléments dans la zone ‘reste à trier’ un par un et on les insère à la bonne place :



On insère 56 à la bonne place

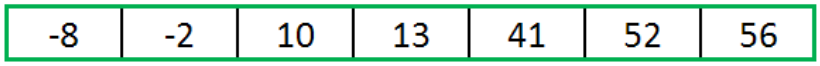


Et ensuite -2



Et ensuite 52

Et ainsi de suite jusqu’au résultat final toute la liste est triée :



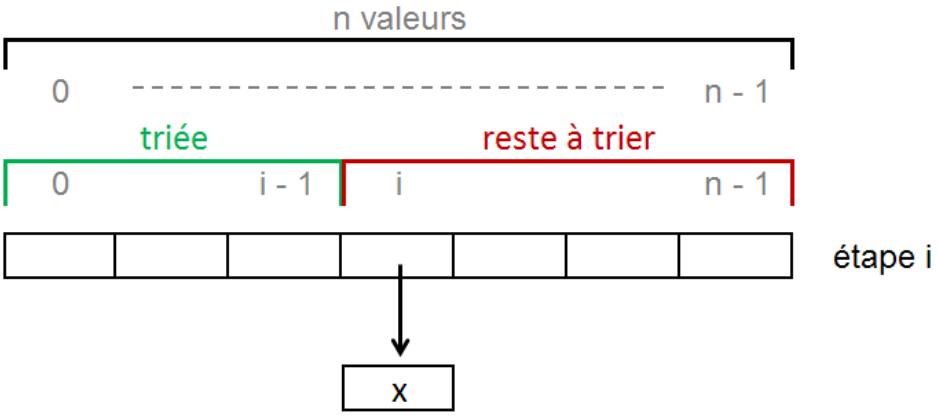
Je l’avais dit mais on ne m’écoute jamais !!

2.1 Description de l'algorithme en pseudo code[3]

On considère un tableau t[ ] de n éléments numérotés de 0 à n-1 . L'objectif consiste à réaliser le tri par sélection de ce tableau. Voilà l'algorigramme en pseudo code :

```
pour i de 1 à n - 1
    # mémoriser T[ i ] dans x
    x ← T[ i ]
    # décaler vers la droite les éléments T[ 0 ] ..T[ i - 1 ]
    # qui sont plus grands que x en partant de T[ i - 1 ]
    j ← i
    tant que j > 0 et T[ j - 1 ] > x
        T[ j ] ← T[ j - 1 ]
        j ← j - 1
    fin tant que
    # placer x dans le "trou" laissé par le décalage
    T[ j ] ← x
fin pour
```

On peut résumer la situation au début de l'étape n°i qui réalise le placement de l'élément n°i de la liste dans la partie de la liste triée :

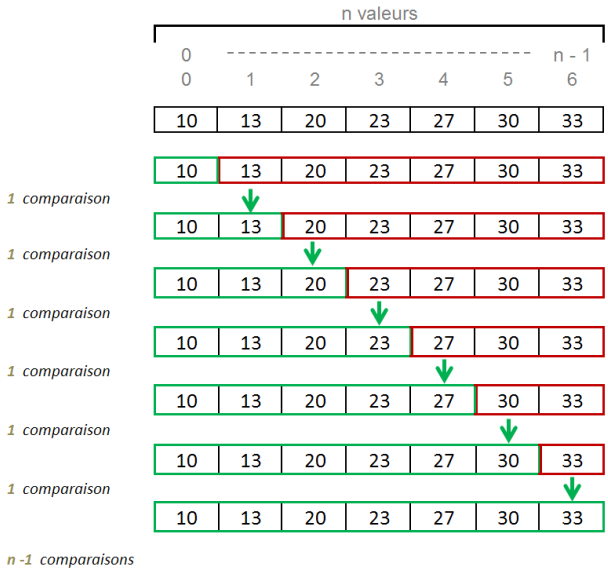


Script4. Compléter le script Python tri par insertion depart eleve.py pour réaliser un tri par insertion d'une liste.

[3] Source : https://fr.wikipedia.org/wiki/Tri_par_sélection

2.2 Analyse théorique de la complexité

En préambule que se passe t-il si nous cherchons à trier un tableau de n éléments déjà triés ? Étudions le cas sur un exemple :



Notre table contenant n valeurs est déjà triée. **C'est le cas le plus favorable.**
Nous pouvons analyser que l'algorithme par insertion réalise n - 1 comparaisons pour finalement ne rien faire de plus.

La complexité de cet algorithme est donc en $O(n)$ dans ce cas.

Et maintenant qu'en est-il du cas le plus défavorable ?

Notre table contenant n valeurs est déjà triée mais à l'envers ! **C'est le cas le plus défavorable.**

Nous pouvons analyser l'algorithme en comptabilisant les opérations nous obtenons :

Pour l'étape n°i : i•comparaison + i•affectation + 2 affectations

Pour l'ensemble des étapes i varie entre 1..n-1 donc en additionnant l'ensemble des opérations nous trouvons :

$$\frac{n(n-1)}{2}(comparaison + affectation) + 2(n-1)affectation$$

La complexité de cet algorithme est donc en $O(n^2)$ dans ce cas.

2.3 Analyse expérimentale de la complexité

a) Analyse pour le cas favorable

Remplir la table à trier avec une compréhension de liste comme ci-dessous. Les éléments, ici 10000, sont donc déjà triés.

```
table_a_trier = [i for i in range(10000)]
```

Nous pouvons calculer le temps d'exécution comme ceci :

```
start_time = time.clock()

tri_par_insertion(table_a_trier)

print("n = %8d valeurs le temps d'éxécution : %10f
secondes" % (longueur, (time.clock() - start_time)))
```

Q3. Vérifier expérimentalement que le comportement de l'algorithme dans le cas favorable est bien d'une complexité en $O(n)$.

Tri par insertion cas favorable en mS

n valeurs	1000	2000	4000	10000	20000	40000
Durée						

b) Analyse pour le cas défavorable

Pour remplir la table dans l'ordre inverse avec 4000 éléments par exemple :

```
table_a_trier = [4000-i for i in range(4000)]
```

Q4. Vérifier expérimentalement que le comportement de l'algorithme dans le cas favorable est bien d'une complexité en $O(n^2)$.

Tri par insertion cas défavorable en S

n valeurs	1000	2000	4000	8000
Durée				

3 Deux définitions : le tri en place et le tri stable.

3.1 Le tri stable

On appelle **tri stable** un tri qui conserve l'ordre d'apparition des éléments égaux.

3.2 Le tri en place

On appelle **tri en place** un tri qui n'utilise pas de tableau auxiliaire.

3.3 Quid de nos algorithmes de tris par sélection et fusion ?

Le tri par sélection est un tri en place, il peut être stable selon les détails d'implémentations. Le tri par insertion est un tri stable et en place.

4 En complément un autre exemple de tri : le tri à bulles

4.1 Description de l'algorithme

Le tri à bulle est ainsi nommé car il reprend l'idée des bulles qui remontent à la surface d'un cours d'eau ou bien d'une marre. Dans notre algorithme il s'agit de comparer deux à deux les valeurs de notre table et de les échanger si elles ne sont pas dans l'ordre requis.

On réalisera plusieurs passes successives jusqu'à ce qu'il n'y ait plus aucun échange ce qui signifiera que la table est triée.

Sens des tests successifs →

10	56	-2	52	-8	41	13

Voilà la première passe sur notre table :

n valeurs

	0	1	2	3	4	5	n - 1 6	Test si la plus grande valeur du couple est à droite	Table triée
	10	56	-2	52	-8	41	13	Echange ?	
1ère passe	10	56	-2	52	-8	41	13	non	oui
	10	56	-2	52	-8	41	13	oui	non
	10	-2	56	52	-8	41	13	échange fait	
	10	-2	56	52	-8	41	13	oui	
	10	-2	52	56	-8	41	13	échange fait	
	10	-2	52	56	-8	41	13	oui	
	10	-2	52	-8	56	41	13	échange fait	
	10	-2	52	-8	56	41	13	oui	
	10	-2	52	-8	41	56	13	échange fait	
	10	-2	52	-8	41	56	13	oui	
	10	-2	52	-8	41	13	56	échange fait	non

Commentaires :

a) on voit bien que la valeur 56 qui est dans la case n°1 du tableau va remonter de proche en proche jusqu'à la case la plus à droite car elle a la valeur la plus élevée. C'est bien à l'image d'une bulle qui remonte tranquillement vers la surface (ici le coté droit).

b) à chaque analyse d'une paire de valeurs cote à cote on mémorise le fait qu'il y ait besoin d'effectuer un échange. Et dans ce cas la table n'est pas triée l'indicateur est mis à NON.

c) c'est seulement quand dans une phase aucun échange n'est nécessaire que la table est triée.

Q5. En observant la table ci-dessous que l'on veut trier avec les valeurs les plus élevées à droite. Indiquer combien d'échanges vont être réalisés sur une passe de l'algorithme.

-10	24	-2	-12	30	41	13

Q6. Même question pour celle-ci ?

-4	0	1	12	21	34	43

4.2 L'algorithme en pseudo code

On considère un tableau t [ ] de n éléments numérotés de 0 à n-1 le pseudo algorithme du tri à bulles est donné ci-dessous :

```
flag ← faux
faire tant que flag=faux
        flag ← vrai
        pour i allant de 0 à n - 2
                si t [ i +1 ] < t [ i ] alors
                        échanger( t [ i +1 ], t [ i ] )
                        flag ← faux
                fin si
        fin pour
fin tant que
```

Q7. Faire fonctionner à la main : compléter le tableau ci-dessous :

10	56	-2	52	-8	41	13
10	-2	52	-8	41	13	56

4.3 Codage de l'algorithme en Python

Script5. Compléter l'algorithme du tri à bulles en Python.
tri a bulles depart eleve.py

4.4 Analyse expérimentale de la complexité

Nous allons déterminer expérimentalement la complexité de cet algorithme :

Script6. Modifier l'algorithme du tri à bulles en Python pour permettre la mesure du temps de calcul du tri.

Q8. Remplir le tableau ci-dessous puis vérifier que la complexité de l'algorithme de tri à bulles est bien en $O(n^2)$.

Tri à bulles en S

n valeurs	500	1000	2000	4000	8000	16000	32000
Durée							

5 Annexes mesures de durées d'exécution, logarithme en Python

5.1 Mesure de durée avec Python < 3.8

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

Mesure_de_la_duree_en_Python.py

```
#!/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  ))
```

5.2 Mesure de durée avec Python >= 3.8 et fstring

Comment mesurer la durée Python 3_8.py

```
from time import process_time

print()
print("CODE DE TEST")

# assigning n = 50
n = 50

# Start the stopwatch / counter
t1_start = process_time()

for i in range(n):
    print(i, end =' ')
    for j in range(1000):
        a = j**j
print()

# Stop the stopwatch / counter
t1_stop = process_time()

print()
print("BILAN")
print(f"Dates fin:{t1_stop:8.5f} S   Début:{t1_start:8.5f} S")
print(f"Temps passé par le programme en millisecondes: {(t1_stop-t1_start)*1000:8.5f} mS")
```

5.3 Calculs des logarithmes en Python

```
from math import *

## LES LOGARITHMES
# logarithme naturel (base e) ln(a) : log(a)  !! ATTENTION
# logarithme base b  : log(a,b)
# logarithme base 2  : log2(a)
# logarithme base 10 : log10(a)
a = 12
print("Logarithme néperien de a Ln(a) :",log(a))
print("Logarithme décimal de a  log(a):",log10(a))
print("Logarithme base 2 de a :",log2(a))
print("Logarithme base 4 de a :",log(a,4))
```

```
Logarithme néperien de a Ln(a) : 2.4849066497880004
Logarithme décimal de a  log(a): 1.0791812460476249
Logarithme base 2 de a : 3.584962500721156
Logarithme base 4 de a : 1.7924812503605783
Valeur a : 1.7924
Arrondi inférieur de a: 1
Arrondi supérieur de a: 2
```

5.4 Quelques algorithmes de tri illustrés

https://interstices.info/les-algorithmes-de-tri/

https://fr.wikipedia.org/wiki/Tri_par_insertion

https://fr.wikipedia.org/wiki/Tri_par_sélection