Recursivite corrige

October 20, 2021

1 Définition et premier exemple

Définition:

Une fonction est dite récursive si elle s'appelle elle même.

Exemple:

Les deux fonctions suivantes calculent la puissance n-ieme de x en utilisant uniquement l'opération multiplication.

La première fonction utilise un algorithme "habituel" avec une méthode itérative basée sur une boucle for.

[1]:
```
def puissance_iterative(x,n) :
    resultat = 1
    for i in range(n):
        resultat = x*resultat
    return resultat
```

On peut alors tester cette fonction en calculant 2^8

[2]:
```
test_iteratif = puissance_iterative(2,8)
print(test_iteratif)
```

```
256
```

La deuxième fonction utilise un algorithme récursif qui s'appelle lui même.

[3]:
```
def puissance_recursive(x,n) :
    if n==0 :
        resultat = 1
        return resultat
    else :
        resultat = x*puissance_recursive(x,n-1)
        return resultat
```

[4]:
```
test_recursif = puissance_recursive(2,8)
print(test_recursif)
```

```
256
```

Les deux fonctions donnent le même résultat mais fonctionne différement.

Méthode itérative:

La fonction itérative réalise une boucle en calculant dans l'ordre :

- 1
- puis $1 \times 2 = 2$
- puis $2 \times 2 = 4$
- puis $4 \times 2 = 8$
- puis ...

Elle stocke la valeur du résultat dans un nombre à chaque tour de boucle, et retourne en fin de boucle le résultat final.

Méthode récursive:

La fonction récursive appelle les calculs dans l'ordre inverse :

- si je connais le résultat de 2^7 alors je dois calculer $2 \times 2^7 = 2^8$
- si je connais le résultat de 2^6 alors je dois calculer $2 \times 2^6 = 2^7$
- si je connais le résultat de 2^5 alors je dois calculer $2 \times 2^5 = 2^6$
- si ...
- si je connais le résultat de 2^0 alors je dois calculer $2 \times 2^0 = 2^1$
- je connais le résultat de 2^0 c'est 1
- donc je peux calculer $2^1 = 2 \times 2^0 = 2 \times 1 = 2$
- donc je peux calculer $2^2 = 2 \times 2^1 = 2 \times 2 = 4$
- donc ...
- donc je peux calculer $2^8 = 2 \times 128 = 256$

2 Pile d'appel, critère d'arrêt et terminaison

On reconnait une structure de pile dans l'appel récursif d'une fonction :

- la pile d'appel est vide
- j'empile : "je dois calculer 2 $\times$ puissance_recursive(2,7)"
- j'empile : "je dois calculer 2 $\times$ puissance_recursive(2,6)"
- j'empile : "je dois calculer 2 $\times$ puissance_recursive(2,5)"
- j'empile : ...
- je connais puissance_recursive(2,0) = 1
- je dépile l'appel suivant "je dois calculer 2 $\times$ puissance_recursive(2,0)" et je calcule : puissance_recursive(2,1) = 2 $\times$ 1 = 2
- je dépile l'appel suivant "je dois calculer 2 $\times$ puissance_recursive(2,1)" et je calcule : puissance_recursive(2,1) = 2 $\times$ 1 = 2
- je dépile : ...
- je dépile l'appel suivant "je dois calculer 2 $\times$ puissance_recursive(2,7)" et je calcule : puissance_recursive(2,8) = 2 $\times$ 128 = 256
- la pile d'appel est vide, j'ai finit le résultat est 256.

Le critère qui nous informe quand on doit arrêter d'empiler de nouveaux appels et passer à dépiler c'est lorsque l'on rencontre ce que l'on appelle **le cas d'arrêt**, ici 2^0, dont on connait la valeur à retourner, ici 1. Lors de l'écriture d'une fonction récursive il faut bien s'assurer que les appels successifs rencontrent le cas d'arrêt, ici l'exposant k de 2^k diminue de 1 à chaque appel, pour assurer **la terminaison du programme**.

Par exemple la fonction puissance_recursive(x,n) écrite plus haut ne se termine jamais pour des valeurs de n non entière ou négative, on dit que la terminaison du programme n'est pas assuré pour ces valeurs.

3 Exercice d'application :

Ecrire deux fonctions prenant en argument un entier positif n, qui retournent factorielle n ,$n!$. Une fonction utilisera une méthode itérative et l'autre une méthode récursive. Tester vos deux fonctions.

[5]:
```
def factorielle_classique(n):
    resultat = 1
    for i in range(n):
        resultat = (i+1)*resultat
    return resultat
```

[6]:
```
test_classique = factorielle_classique(3)
print(test_classique)
```

```
6
```

[7]:
```
def factorielle_recursive(n):
    if n==1:
        return 1
    else :
        resultat = n*factorielle_recursive(n-1)
        return resultat
```

[8]:
```
test_recursif = factorielle_recursive(3)
print(test_recursif)
```

```
6
```

4 Récursivité et récurrence

On remarque qu'une fonction récursive est liée à la notion de suite définie par récurrence.

Une suite définie par récurrence est une suite définie par son ou ses premiers termes et par une relation de récurrence, qui définie chaque terme à partir du ou des précédents.

Par exemple la suite (u_n) telle que :

$u_0 = 2$

et

$u_n = \frac{1}{2}\left(u_{n-1} + \frac{3}{u_{n-1}}\right)$

est définie par récurrence.

Le premier terme correspond au cas d'arrêt, si n = 0 alors la valeur de u_n est connue, c'est $u_0 = 2$.

La relation de récurrence, correspond à l'appel de la fonction par elle même, u_n est exprimée en fonction de u_{n-1}.

Enfin à chaque relation de récurrence l'indice de la suite diminue de 1, on atteint donc bien le cas d'arrêt qui est $u_0 = 2$ après n appel de la relation de récurrence, la terminaison du programme est assurée.

Exercice:

Ecrire une fonction récursive qui prend en argument un indice n et retourne le terme de la série u_n. Testez votre fonction pour les premières valeurs de la suite et remarquez qu'elle converge rapidement vers $\sqrt{3}$.

[9]:
```
def u(n):
    if n == 0:
        return 2.
    else:
        return 0.5 * (u(n-1) + 3. / u(n-1))
```

[10]:
```
print(u(4))

print(3**0.5)
```

```
1.7320508075688772
1.7320508075688772
```

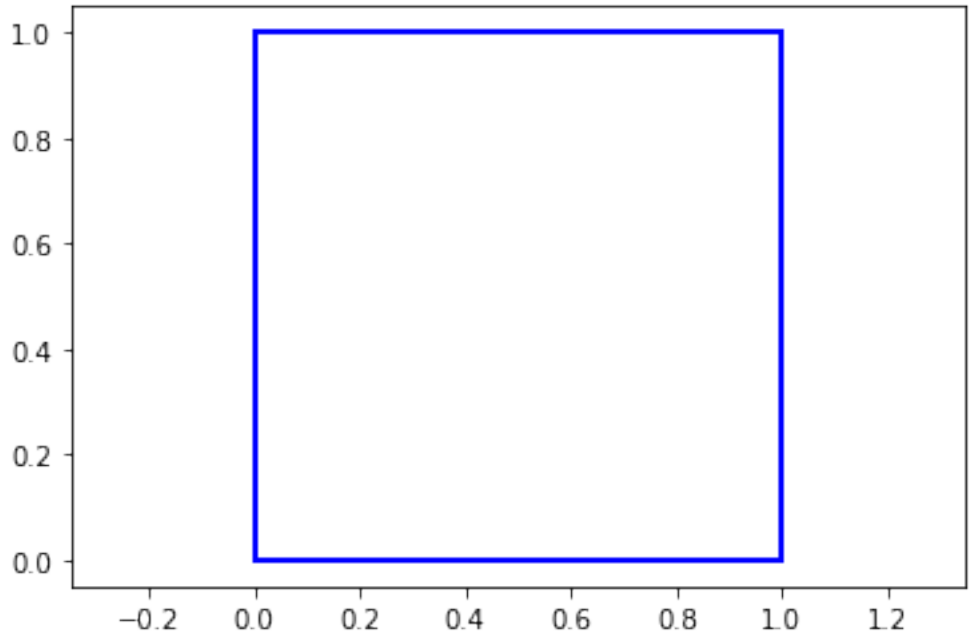
5 Exercice:

On définit la fonction suivante qui repère dans le plan complexe les affixes des quatres sommets d'un carré.

[11]:
```
def carre(a, b):
    return [a, b, b - 1j * (a - b), a + 1j * (b - a), a]
```

On peut alors tracer un carré avec la librairie matplotlib en reliant les quatres coins du carré.

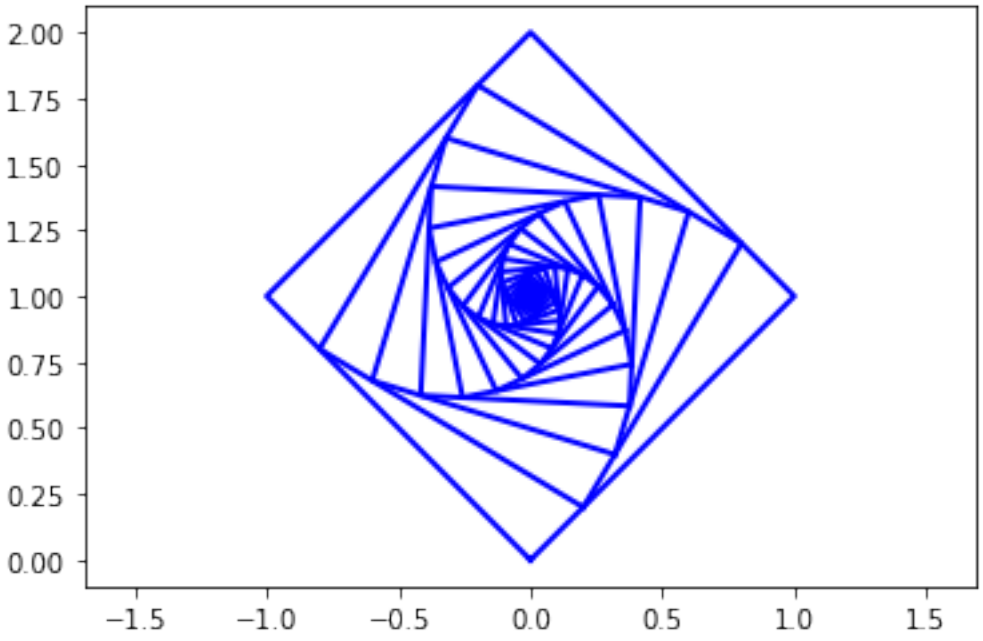
[13]:
```
import matplotlib.pyplot as plt
L = carre(1, 1 + 1j)
plt.axis('equal')
for a in L:
    plt.plot([a.real for a in L], [a.imag for a in L], 'b', lw=2)
plt.show()
```



Écrire une fonction récursive qui permet d'obtenir la figure suivante

[14]:
```
def carrerec(a, b, l, eps=0.01):
    """ l entre 0 et 1
    """
    L = [carre(a, b)]
    if abs(b - a) > eps:
        L.extend(carrerec(a + (b - a) * l, b + (b - a) * 1j * l, l, eps))
    return L
```

[15]:
```
BigL = carrerec(0j, 1 + 1j, .2, eps=0.01)
plt.axis('equal')
for L in BigL:
    plt.plot([a.real for a in L], [a.imag for a in L], 'b', lw=2)
plt.show()
```

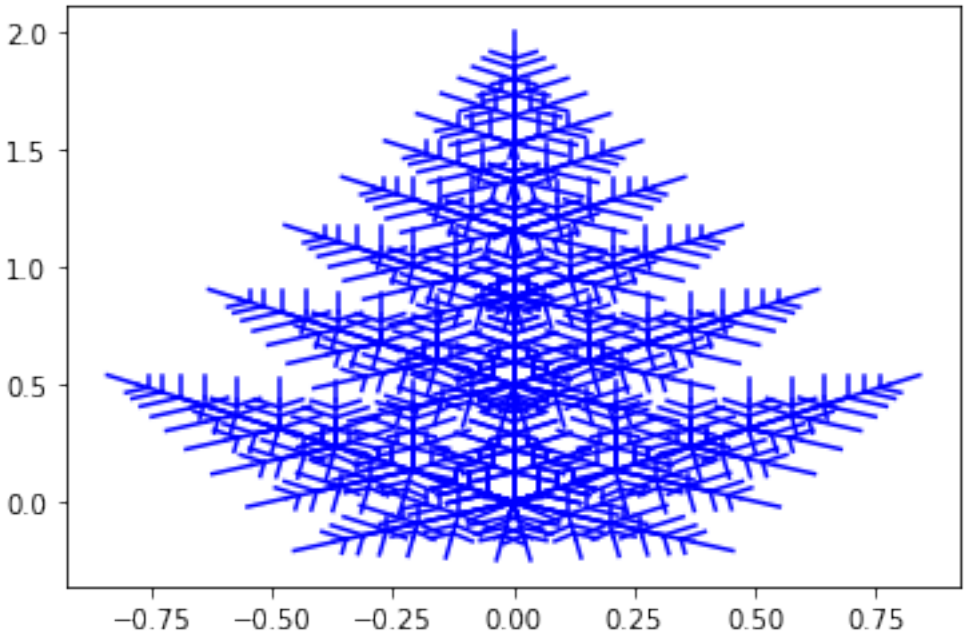


À partir d'un segment $[a, b]$ où a et b sont complexes, on trace le segment $[a, \frac{1}{4}(3a+b)]$ puis on recommence avec a $\leftarrow$ a et b $\leftarrow a + \frac{1}{2}(b-a)e^{i}$; a $\leftarrow$ a et b $\leftarrow a + \frac{1}{2}(b-a)e^{-i}$ et enfin a $\leftarrow \frac{1}{4}(3a+b)$ et b $\leftarrow$ b.

Tracer la figure en partant de a = 0 et b = 2.

[16]:
```
import numpy as np
def sapin(a, b, eps=0.1):
    L = []
    if abs(a - b) < eps:
        return [(a, b)]
    L.extend(sapin(a, a + (b - a) * .5 * np.exp(1j), eps))
    L.extend(sapin(a, a + (b - a) * .5 * np.exp(-1j), eps))
    L.append((a, (3 * a + b) / 4))
    L.extend(sapin((3 * a + b) / 4, b, eps))
    return L
```

[17]:
```
L = sapin(0., 2j)
for a, b in L:
    plt.plot([a.real, b.real], [a.imag, b.imag], 'b')
plt.show()
```



6 Rappels sur la complexité temporelle

L'exécution d'un programme nécessite d'utiliser plus ou moins les capacités de votre ordinateur. Ceci ce quantifie avec deux facteurs principaux: le temps de calcul et la mémoire occupé lors de l'exécution. Pour être plus performant un programme doit demander le moins de ressource possible à l'ordinateur et donc avoir un temps de calcul court et une petite mémoire occupée.

Pour quantifier les besoins nécessaire à l'éxécution d'un programme, on utilise la complexité. On parle de complexité temporelle lorsqu'il s'agit du temps de réponse, et de complexité spatiale pour l'espace mémoire. Ce paragraphe traitera de la complexité temporelle.

Le temps exact en seconde que prend un programme à effectuer un programme dépend de nombreux facteurs techniques propores à chaque machine. On peut estimer l'influence de l'algorithme sur le temps en considérant que chaque opération élémentaire (addition, multiplication, comparaison, ...) représente une unité de coût sans préciser le temps en seconde pour chaque unité de coût. La complexité temporelle est le nombre total d'unité de coût pour l'exécution de cet algorithme.

Enfin on étudie l'évolution de cette complexité en fonction de la taille des données à étudier. En effet l'algorithme ne fera pas le même nombre d'opération selon le problème à traiter par exemple le nombre d'élément d'une liste à trier ou la valeur du nombre à calculer. Lorsqu'on étudie la complexité on étudie donc un équivalent de l'unité de coût en fonction de la taille des données.

Si C(n) est la complexité pour des données de taille n, alors on peut avoir comme résultat : - $C(n) = O(\log(n))$, complexité logarithmique - $C(n) = O(n)$, complexité linéaire - $C(n) = O(n\log(n))$, complexité quasi-linéaire - $C(n) = O(n^2)$, complexité quadratique - $C(n) = O(n^k)$, complexité polynomiale - $C(n) = O(\exp(n))$, complexité exponentielle

Reprenons le programme itératif calculant la puissance n-ième d'un nombre x

[18]:
```
def puissance_iterative(x,n) :
    resultat = 1
    for i in range(n):
        resultat = x*resultat
    return resultat
```

Et calculons sa complexité en fonction de la puissance n

Ce programme commence par affecter 1 à résultat donc on compte 1 unité de coût Ce programme fait une boucle n fois donc on va multiplier par n le coût suivant: dans la boucle le programme effectue une multiplication donc 1 unité de coût

La complexité du programme est donc $C(n) = 1 + n \times 1 = O(n)$ c'est une complexité linéaire selon la puissance calculée.

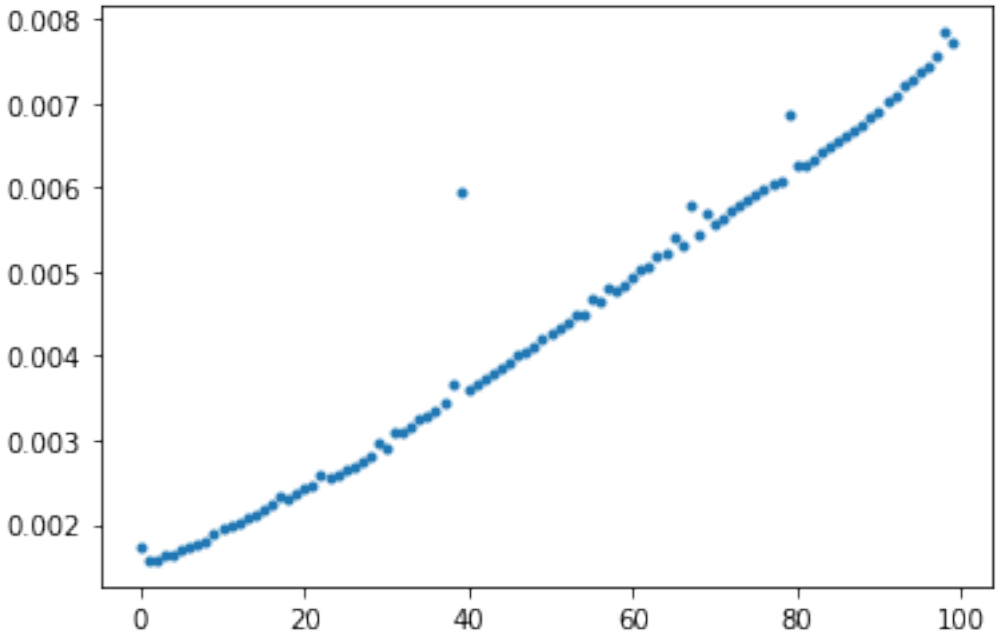
On peut vérifier ce résultat à l'aide du programme ci-dessous qui mesure 1000 fois le temps d'exécution de la fonction puissance_iterative pour les puissance de 2 entre 0 et 99.

[19]:
```
import time
import matplotlib.pyplot as plt
resultat = 100*[0]
for j in range(1000):
    for i in range(100):
        depart = time.clock()
        puissance_iterative(2,i)
        arrive = time.clock()
        resultat[i]=resultat[i]+arrive-depart

plt.plot(resultat,'.')
plt.show()
```

```
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:6: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead

C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:8: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
```



7 La complexité temporelle dans le cas d'un programme récursif

Avec un programme itératif nous avons pu estimer la complexité du programme à l'aide d'une expression explicite $C(n)$ est directement une fonction de n. En effet l'écriture d'un programme itératif explicite directement toutes les opérations effectuées.

Ce n'est pas le cas pour un programme récursif où par définition le programme s'appelle lui même. A la lecture du code du programme on est seuleument capable d'exprimer la complexité pour une taille n en fonction de la complexité pour une taille inférieure.

Reprenons par exemple le programme récursif calculant aussi la puissance n-ième d'un nombre x.

[20]:
```
def puissance_recursive(x,n) :
    if n==0 :
        resultat = 1
        return resultat
    else :
        resultat = x*puissance_recursive(x,n-1)
        return resultat
```

Exprimons sa complexité en lisant la structure du programme.

Si n = 0, alors on affecte 1 au résultat donc $C(0) = 1$

Si n > 0, alors on éffectue toute les opérations pour calculer puissance_recursive(x,n-1) soit $C(n-1)$ et on ajoute une opération en multipliant par x le résultat. On en déduit $C(n) = C(n-1)+1$

On obtient donc une relation de récurrence qui définit la complexité :

$C(n) = C(n-1)+1$ et $C(0) = 1$

Il s'agit d'une suite arithmétique que l'on peut résoudre avec $C(n) = n + 1 = O(n)$ On en déduit donc une complexité linéaire aussi.

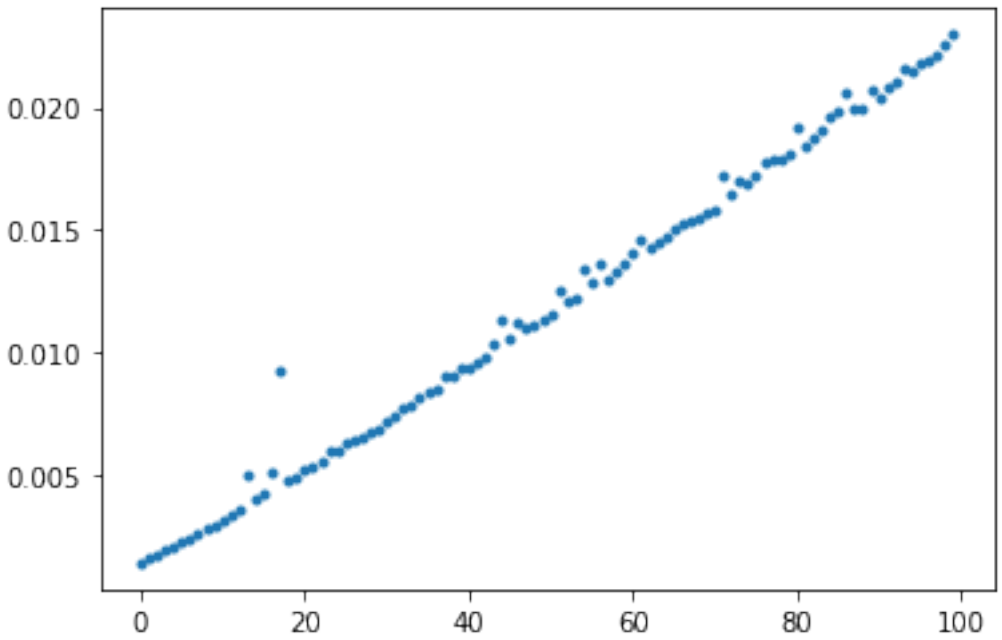
Et on peut vérifier ce résultat de la même façon que pour le programme itératif avec le programme suivant.

[21]:
```
import time
import matplotlib.pyplot as plt
resultat = 100*[0]
for j in range(1000):
    for i in range(100):
        depart = time.clock()
        puissance_recursive(2,i)
        arrive = time.clock()
        resultat[i]=resultat[i]+arrive-depart

plt.plot(resultat,'.')
plt.show()
```

```
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:6: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead

C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:8: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
```



Pour les deux programmes ci-dessus on obtient dans les deux cas une complexité linéaire, on dit que les deux programmes ont une complexité équivalente. Lorsque l'on veut améliorer un programme on utilise un programme ayant le même but avec une complexité inférieure. Prenons l'exemple de la fonction puissance_rapide(x, n) ci-dessous.

Dans le programme ci-dessous x//y renvoi le quotient de la division euclidienne de n par 2, et x%y renvoi le reste de la division euclidienne de x par 2.

[22]:
```
def puissance_rapide(x, n):
    if n == 0:
        return 1
    else:
        r = puissance_rapide(x, n // 2)
        if n % 2 == 0:
            return r * r
        else:
            return x * r * r
```

Evaluons sa complexité en lisant le programme :

si n=0, alors on affecte 1 au résultat donc $C(0) = 1$

si n>0, alors on effectue le nombre d'opération pour calculer puissance_rapide(x, n/2) puis on effectue une multiplication, donc $C(n) = C(n/2) + 1$

On a donc pour $C(n)$ la relation de récurrence $C(n) = C(n/2) + 1$ et $C(0) = 1$. Pour évaluer $C(n)$ on peut effectuer le changement de variable $n = 2^k$ donc $C(2^k) = C(2^(k-1)) + 1$ et $C(2^0) = C(0) + 1 = 2$. Il s'agit donc d'une suite arithmétique avec k d'où $C(2^k) = k + 2$ or $2^k = n$ d'où $k = \log_2(n)$ et $C(n) = \log_2(n) + 2$.

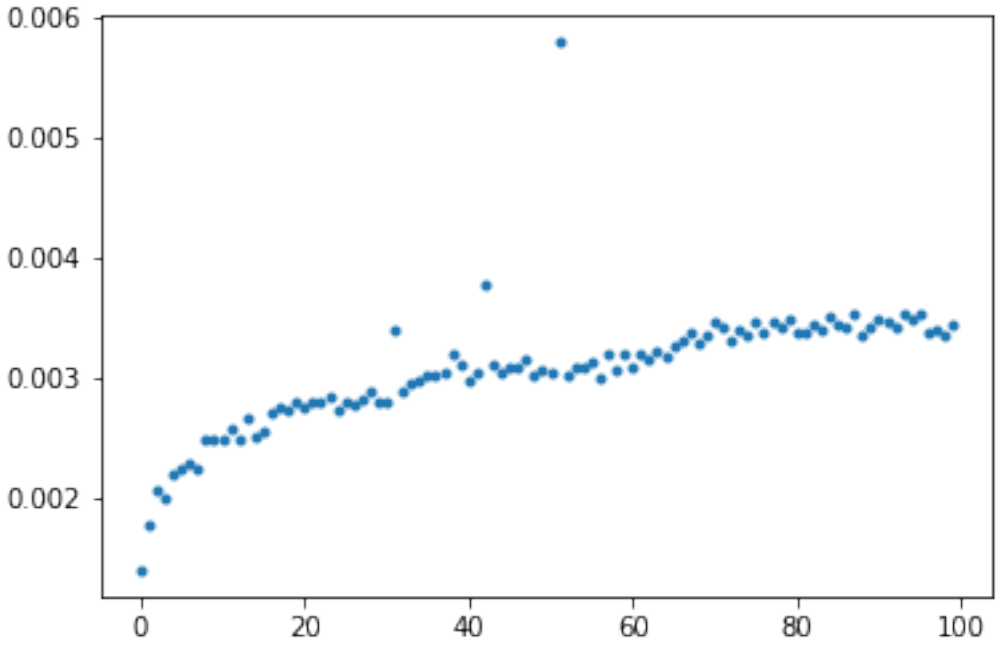
Il s'agit donc d'une complexité logarithmique et donc d'un programme plus performant que les précédents pour calculer la puissance n-ième de x. Dans certains cas un fonction récursive permet de diminuer la complexité temporelle de l'algorithme. Vérifions ce résultat à nouveau.

[23]:
```
import time
import matplotlib.pyplot as plt
resultat = 100*[0]
for j in range(1000):
    for i in range(100):
        depart = time.clock()
        puissance_rapide(2,i)
        arrive = time.clock()
        resultat[i]=resultat[i]+arrive-depart

plt.plot(resultat,'.')
plt.show()
```

```
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:6: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
```

```
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:8: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
```



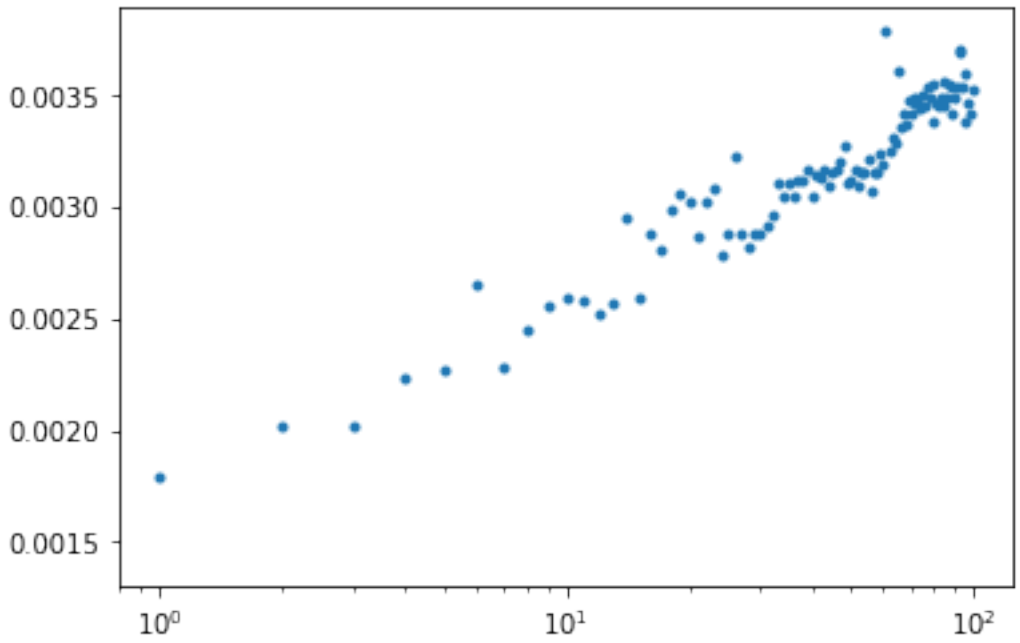
On remarque que la complexité est sub-linéaire, pour s'assurer qu'il s'agit bien d'une complexité logarithmique traçons plutôt le temps d'éxecution en fonction de $\log(n)$ avec :

[24]:
```
import time
import matplotlib.pyplot as plt
resultat = 100*[0]
for j in range(1000):
    for i in range(100):
        depart = time.clock()
        puissance_rapide(2,i)
        arrive = time.clock()
        resultat[i]=resultat[i]+arrive-depart

plt.semilogx(resultat,'.')
plt.show()
```

```
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:6: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
```

```
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:8: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
```



7.1 Exercice

En reprenant les programmes ci-dessous écrit pour le sujet précédent évaluer leur complexité temporelle et vérifier votre résultat.

7.1.1 le calcul de factorielle itératif

[25]:
```python
def factorielle_classique(n):
    resultat = 1
    for i in range(n):
        resultat = (i+1)*resultat
    return resultat
```

complexité $C(n) = 1 + n = O(n)$

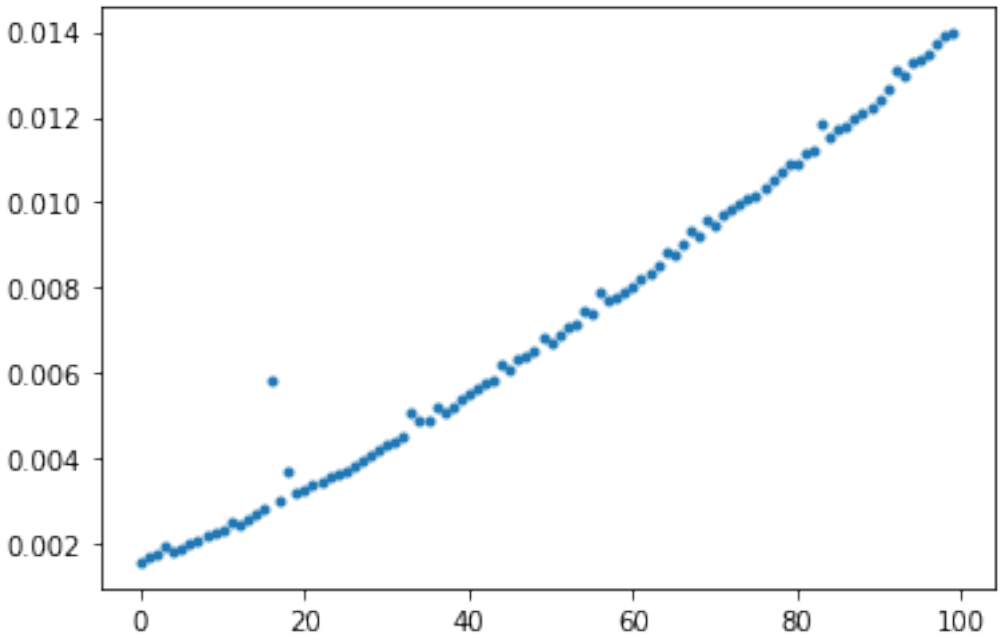
[26]:
```python
import time
import matplotlib.pyplot as plt
resultat = 100*[0]
for j in range(1000):
    for i in range(100):
        depart = time.clock()
```

```
        factorielle_classique(i)
        arrive = time.clock()
        resultat[i]=resultat[i]+arrive-depart

plt.plot(resultat,'.')
plt.show()
```

```
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:6: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead

C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:8: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
```



7.1.2 le calcul de factorielle récursif

[27]:
```
def factorielle_recursive(n):
    if n==1:
        return 1
    else :
        resultat = n*factorielle_recursive(n-1)
```

```
        return resultat
```

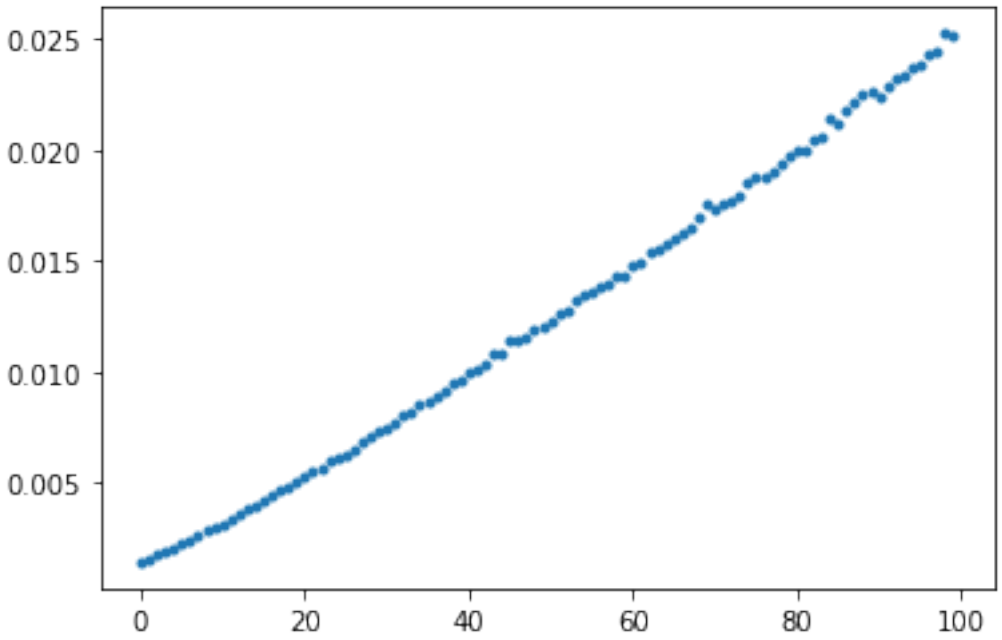
$C(n) = C(n-1) + 1$ et $C(1) = 1$ donc $C(n) = n = O(n)$

[28]:
```
import time
import matplotlib.pyplot as plt
resultat = 100*[0]
for j in range(1000):
    for i in range(100):
        depart = time.clock()
        factorielle_recursive(i+1)
        arrive = time.clock()
        resultat[i]=resultat[i]+arrive-depart

plt.plot(resultat,'.')
plt.show()
```

```
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:6: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead

C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:8: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
```



Que pouvez-vous conclure sur l'intérêt d'utiliser une fonction récursive par rapport à une fonction itérative pour évaluer la fonction factorielle ?

Dans les deux cas la complexité est linéaire, le seul intérêt est dans l'écriture du programme on n'a pas besoin d'écrire une boucle.

7.1.3 la suite définie par récurrence

[29]:
```
def u(n):
    if n == 0:
        return 2.
    else:
        return 0.5 * (u(n-1) + 3. / u(n-1))
```

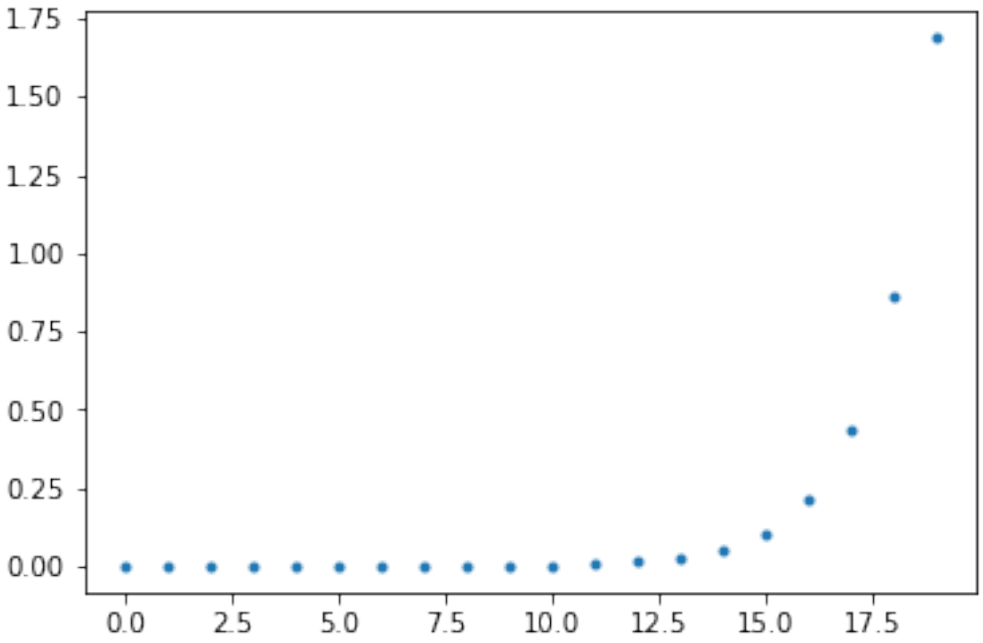
$C(n) = 2 * C(n-1) + 1$ et $C(0) = 1$ donc $C(n) = O(\exp(n))$ c'est une complexité exponentielle

[30]:
```
import time
import matplotlib.pyplot as plt
resultat = 20*[0]
for j in range(10):
    for i in range(20):
        depart = time.clock()
        u(i)
        arrive = time.clock()
        resultat[i]=resultat[i]+arrive-depart

plt.plot(resultat,'.')
plt.show()
```

```
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:6: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead

C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:8: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
```

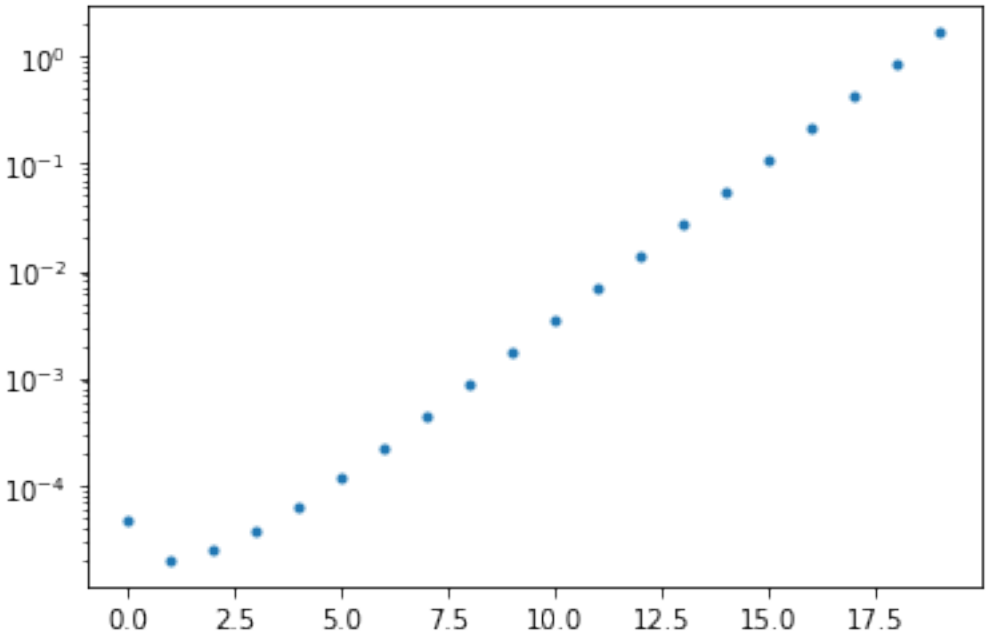


[31]:
```
import time
import matplotlib.pyplot as plt
resultat = 20*[0]
for j in range(10):
    for i in range(20):
        depart = time.clock()
        u(i)
        arrive = time.clock()
        resultat[i]=resultat[i]+arrive-depart

plt.semilogy(resultat,'.')
plt.show()
```

```
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:6: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead

C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:8: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
```



si on compare au même programme mais écrit légèrement différement suivant

[32]:
```
def u(n):
    if n == 0:
        return 2.
    else:
        intermediaire = u(n-1)
        return 0.5 * (intermediaire + 3. / intermediaire)
```

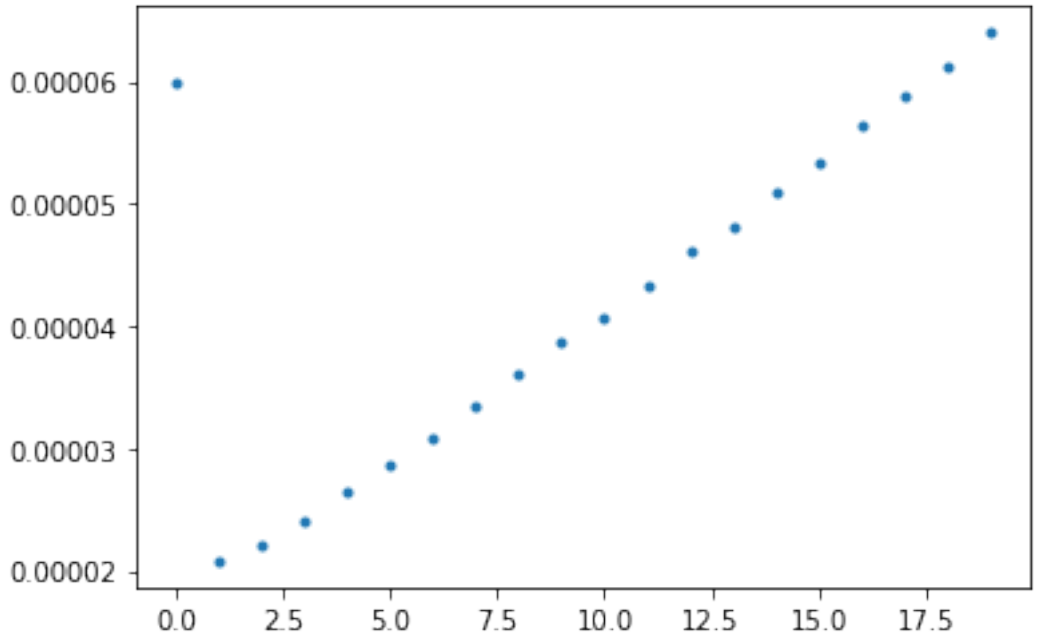
$C(n) = C(n-1) + 1$ et $C(0) = 1$

[33]:
```
import time
import matplotlib.pyplot as plt
resultat = 20*[0]
for j in range(10):
    for i in range(20):
        depart = time.clock()
        u(i)
        arrive = time.clock()
        resultat[i]=resultat[i]+arrive-depart

plt.plot(resultat,'.')
plt.show()
```

```
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:6: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
```

```
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:8: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
```



On en déduit que la récursivité permet d'écrire naturellement les relations de récurrence, mais qu'il faut faire attention à la complexité.

7.2 Exercice

Soit l une liste triée par ordre croissant et x un élément à chercher dans cette liste.

Ecrire un programme récursif qui prend en argument la liste triée l et l'élément à rechercher x et qui renvoie True si x est dans l, et False si x n'est pas dans l. On utilisera une méthode de recherche par dichotomie.

Puis évaluer et vérifier sa complexité.

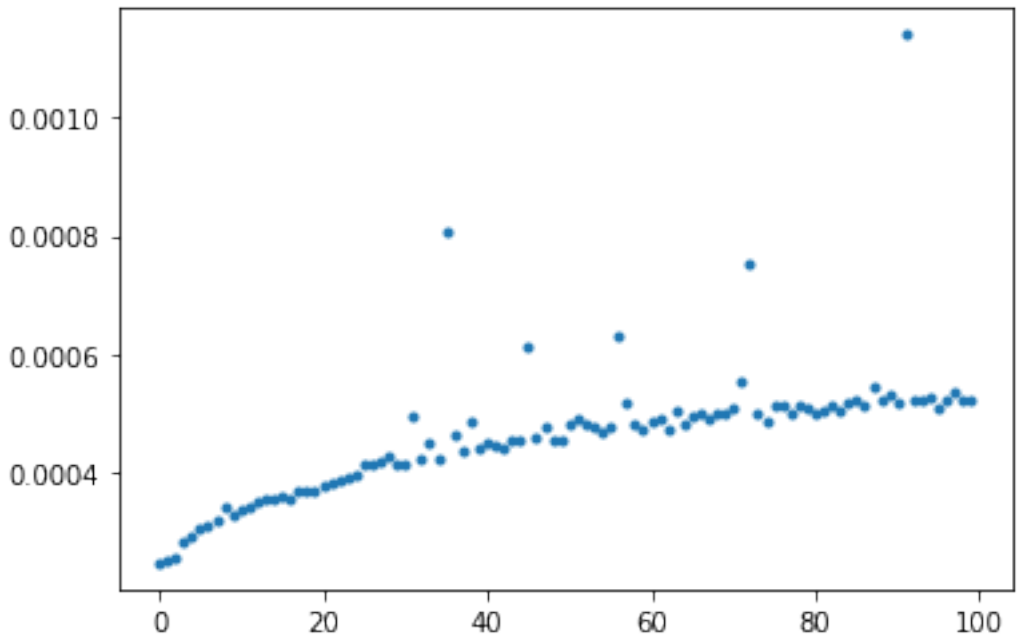
[34]:
```
def dichotomie(x,l):
    n = len(l)
    if n==0:
        return False
    elif x < l[n//2]:
        return dichotomie(x,l[0:n//2])
    elif x > l[n//2]:
        return dichotomie(x,l[n//2+1:n])
    else :
        return True
```

complexité logarithmique

[35]:
```
import time
import matplotlib.pyplot as plt
import random as rd
resultat = 100*[0]
for j in range(100):
    for i in range(100):
        x = rd.randint(0,i)
        l = list(range(i+1))
        depart = time.clock()
        dichotomie(x,l)
        arrive = time.clock()
        resultat[i]=resultat[i]+arrive-depart

plt.plot(resultat,'.')
plt.show()
```

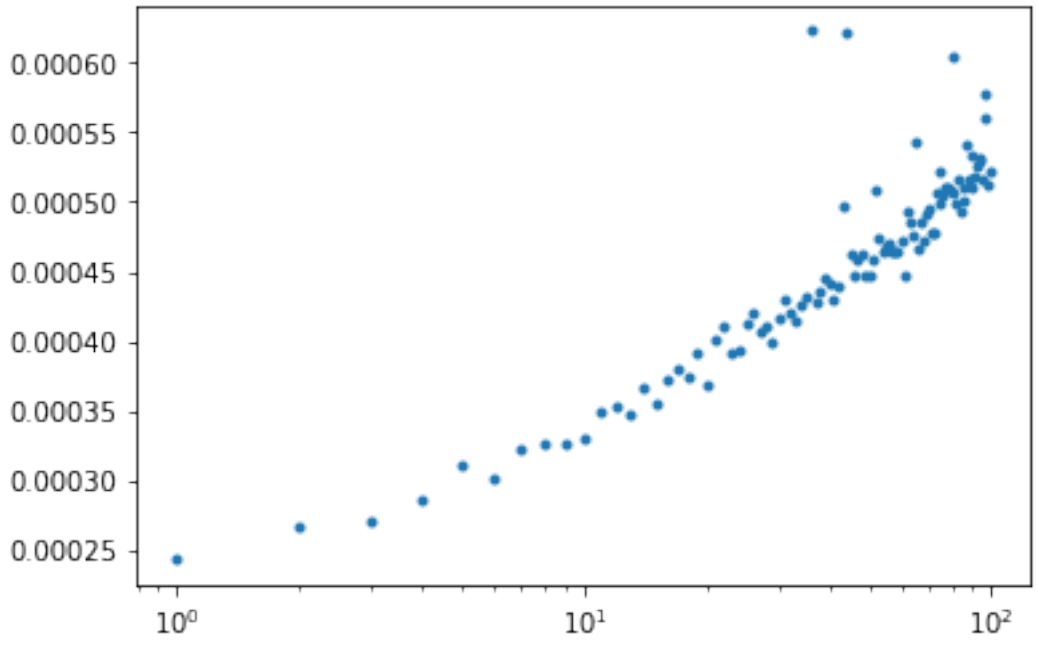
```
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:9: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
  if __name__ == '__main__':
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:11: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
  # This is added back by InteractiveShellApp.init_path()
```



[36]:
```
import time
import matplotlib.pyplot as plt
import random as rd
resultat = 100*[0]
for j in range(100):
    for i in range(100):
        x = rd.randint(0,i)
        l = list(range(i+1))
        depart = time.clock()
        dichotomie(x,l)
        arrive = time.clock()
        resultat[i]=resultat[i]+arrive-depart

plt.semilogx(resultat,'.')
plt.show()
```

```
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:9: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
  if __name__ == '__main__':
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:11: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
  # This is added back by InteractiveShellApp.init_path()
```



7.3 Exercice

On considère la suite de Fibonacci définie par :

$F_0 = 0$

$F_1 = 1$

$F_n = F_{n-1} + F_{n-2}$

Écrire une fonction récursive basée sur ces relations qui prend n en argument et renvoie F_n.

Quelle est sa complexité ?

Accélérer le calcul de F_n en écrivant plutôt une fonction récursive auxiliaire qui prend en arguments F_{n1}, F_n et $k > 0$ et renvoie F_{n+k} (on pourra poser $F_{-1} = 1$).

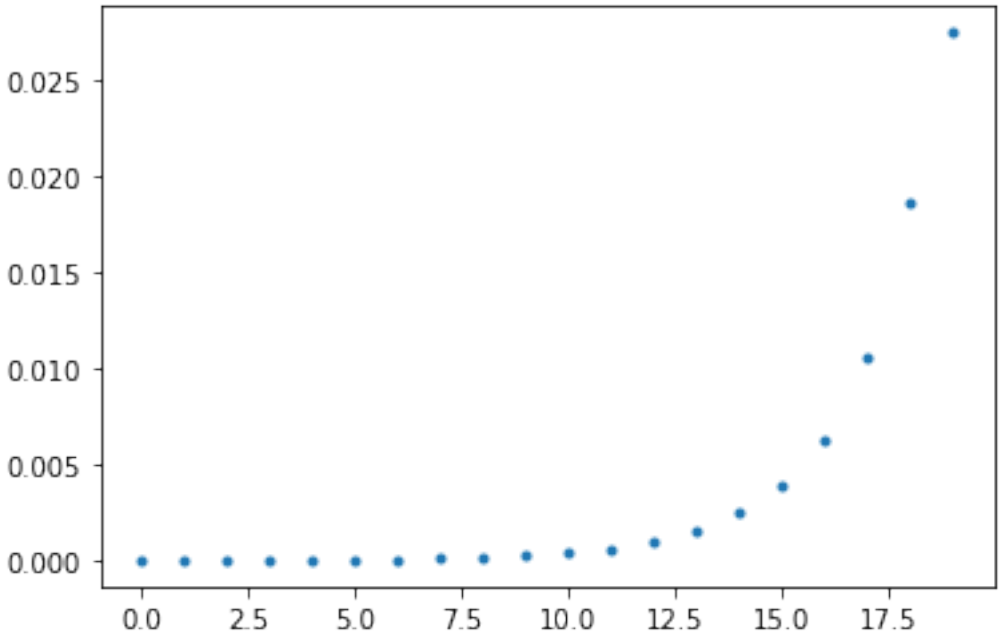
Quelle est la nouvelle complexité ?

[37]:
```
def Fibonacci(n):
    if n == 0:
        resultat = 0
        return resultat
    elif n == 1:
        resultat = 1
        return resultat
    else :
        resultat = Fibonacci(n-1) + Fibonacci(n-2)
        return resultat

import time
import matplotlib.pyplot as plt
resultat = 20*[0]
for j in range(10):
    for i in range(20):
        depart = time.clock()
        Fibonacci(i)
        arrive = time.clock()
        resultat[i]=resultat[i]+arrive-depart

plt.plot(resultat,'.')
plt.show()
```

```
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:17: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:19: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
```



On a la relation de récurrence $C(n) = C(n-1) + C(n-2)$ à chaque appel de la fonction, on appelle deux fois la fonction Fibonacci à chaque appel, on obtiendra donc une complexité exponentielle.

Pour trouver la complexité il faut chercher le nombre Φ tel que $\Phi^n = \Phi^{n-1} + \Phi^{n-2}$ on trouve $\Phi = \frac{1+\sqrt{5}}{2}$ et $C(n) = O(\Phi^n)$

Pour écrire le nouveau programme il faut établir une nouvelle relation de récurrence, écrivons les premiers termes

$F_{n+1} = F_n + F_{n-1}$
$F_{n+2} = 2F_n + F_{n-1}$
$F_{n+3} = 3F_n + 2F_{n-1}$
$F_{n+4} = 5F_n + 3F_{n-1}$

On reconnait les coefficients de la suite de Fibonacci

$F_{n+4} = F_5 \times F_n + F_4 \times F_{n-1}$

On en déduis que

$F_{n+k} = F_{k+1} \times F_n + F_k \times F_{n-1}$

d'où avec n=k+1

$F_{2k+1} = F_{k+1}^2 + F_k^2$

et avec n = k

$F_{2k} = F_k \times (F_{k+1} + F_{k-1})$

ce qui permet de retrouver une complexité quasi-linéaire

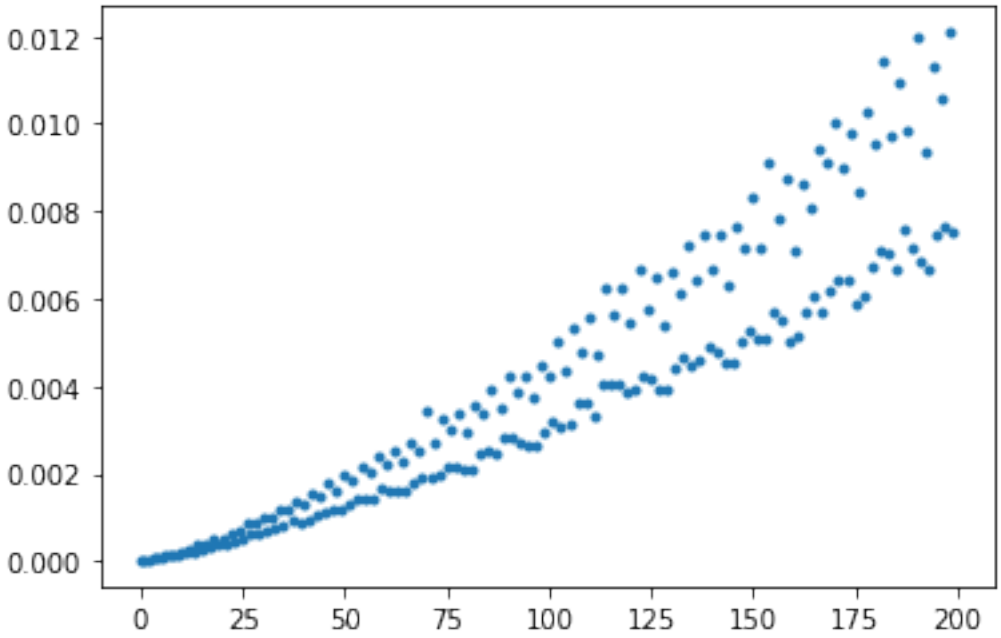
[38]:
```
def Fibonacci_accelere(n):
    if n == -1:
        resultat = 1
        return resultat
    elif n == 0:
```

```
        resultat = 0
        return resultat
    elif n == 1:
        resultat = 1
        return resultat
    elif n == 2:
        resultat = 1
        return resultat
    elif n % 2 == 0:
        k = n//2
        resultat =␣
 ↪Fibonacci_accelere(k)*(Fibonacci_accelere(k+1)+Fibonacci_accelere(k-1))
        return resultat
    elif n % 2 == 1:
        k = n // 2
        resultat = Fibonacci_accelere(k+1)**2+Fibonacci_accelere(k)**2
        return resultat

import time
import matplotlib.pyplot as plt
resultat = 200*[0]
for j in range(20):
    for i in range(200):
        depart = time.clock()
        Fibonacci_accelere(i)
        arrive = time.clock()
        resultat[i]=resultat[i]+arrive-depart

plt.plot(resultat,'.')
plt.show()
```

```
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:28: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:30: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
```



8 Complexité spatiale

La complexité spatiale est analogue à la complexité temporelle, sauf qu'on évalue l'espace mémoire nécessaire à l'exécution de la fonction en fonction de la taille des données.

Prenons l'exemple de la fonction itérative de calcul de la puissance n-ième de x

[39]:
```
def puissance_iterative(x,n) :
    resultat = 1
    for i in range(n):
        resultat = x*resultat
    return resultat
```

Ce programme exécute une boucle de 0 à n-1 et affecte la nouvelle valeur de resultat. L'espace mémoire utilisé est juste 1 nombre stocké dans la variable resultat. On obtient donc une complexité spatiale C(n) = O(1), elle est indépendante de n.

Prenons maintenant l'exemple de la fonction récursive

[40]:
```
def puissance_recursive(x,n) :
    if n==0 :
        resultat = 1
        return resultat
    else :
        resultat = x*puissance_recursive(x,n-1)
        return resultat
```

Cette fois-ci un appel de la fonction puissance_recursive consiste à stocker une valeur de résultat. Pour un appel de la fonction, on a donc une complexité spatiale de O(1). Mais n appels

récursifs sont stockés dans la pile d'appels lors de l'exécution de la fonction, en effet puissance(x,n) demande d'appeler puissance(x,n-1) puis puissance(x,n-2) puis ... jusqu'au critère d'arrêt puissance(x,0). Donc la complexité spatiale totale est de $C(n) = n \times O(1) = O(n)$, elle est linéaire.

Enfin pour la fonction récursive accélérée

[41]:
```
def puissance_rapide(x, n):
    if n == 0:
        return 1
    else:
        r = puissance_rapide(x, n // 2)
        if n % 2 == 0:
            return r * r
        else:
            return x * r * r
```

Encore une fois on stocke un nombre r pour un appel donc la complexité d'un appel est $O(1)$. Pour calculer le nombre d'appels on introduit k tel que $n = 2^k$ et on fait réalise les appels suivant puissance_rapide(x, $n = 2^k$), puis puissance_rapide(x, $n/2 = 2^{k-1}$), puis puissance_rapide(x, $n/4 = 2^{k-2}$), ... Donc on effectue $k = log_2(n)$ appels. La complexité spatiale de la fonction est donc $C(n) = log_2(n) \times O(1) = O(\ln n)$, elle est logarithmique.

8.0.1 exercice:

Calculer la complexité spatiale du programme récursif suivant:

[42]:
```
def u(n):
    if n == 0:
        return 2.
    else:
        return 0.5 * (u(n-1) + 3. / u(n-1))
```

A chaque appel n on doit stocker en mémoire les résultats des appels $n-1$ et $n-1$ d'où une complexité $C(n) = 2C(n-1)$ c'est à nouveau exponentielle $C(n) = 2^n$

8.0.2 exercice

Soit l une liste triée par ordre croissant et x un élément à chercher dans cette liste.

Reprendre le programme récursif qui prend en argument la liste triée l et l'élément à rechercher x et qui renvoie True si x est dans l, et False si x n'est pas dans l. On utilisera une méthode de recherche par dichotomie.

Puis calculer sa complexité spatiale.

[43]:
```
def dichotomie(x,l):
    n = len(l)
    if n==0:
        return False
    elif x < l[n//2]:
        return dichotomie(x,l[0:n//2])
    elif x > l[n//2]:
        return dichotomie(x,l[n//2+1:n])
    else :
```

```
        return True
```

A chaque appel récursif n on a en mémoire la liste l de taille n et on fait appel à la fonction dichotomie avec en mémoire une liste de taille n/2, et cela jusqu' à n=1, on stocke donc en mémoire des listes de tailles :

$ C(n) = n + n/2 + n/4 + n/8 + \dots + 1 = \sum{0}^{i\{max\} = \log 2(n)} 2^i = 2^{i\{max\} + 1\} - 1 = 2n - 1 = O(n)$

9 Comparaison entre itératif et récursif

À travers les différents exemples rencontrés, on remarque que les programmes récursifs présentent l'avantage d'avoir une écriture plus concise que les programmes itératifs, car on n'a pas besoin d'utiliser de boucle.

On remarque également que dans un contexte où une relation de récurrence existe pour résoudre le problème, le programme récursif est plus simple à écrire que le programme itératif.

Par exemple écrire un programme itératif qui calcule les termes de la suite (u_n) telle que :

$u_0 = 2$

et

$u_n = \frac{1}{2}\left(u_{n-1} + \frac{3}{u_{n-1}}\right)$

est plus compliqué que d'utiliser un programme récursif.

Mais lors de l'écriture d'un programme récursif, il faut faire attention à la complexité.

La complexité temporelle du programme si on ne fait pas attention peut être élevée, par exemple elle peut être exponentielle selon l'écriture du programme qui calcule la suite (u_n). Certains cas particulier comme puissance_rapide où on divise la taille de l'argument en 2 à chaque appel peuvent présenter une complexité récursive inférieure à l'itérative correspondante.

La complexité spatiale est minimale pour un programme itératif, un programme récursif utilisera toujours plus d'espace mémoire que le programme itératif correspondant.

10 Terminaison

Lors de l'écriture d'un programme récursif, il faut faire attention à ce que le programme s'arrête un jour.

Prenons l'exemple de calcul de puissance :

[44]:
```
def puissance_rapide(x, n):
    if n == 0:
        return 1
    else:
        r = puissance_rapide(x, n // 2)
        if n % 2 == 0:
            return r * r
        else:
            return x * r * r
```

Identifiez la touche qui vous permet d'interrompre l'exécution de votre programme manuellement et testez ce programme pour des puissances non entières ou des puissances négatives.

Que ce passe-t-il, et pourquoi ?

On remarque que le programme ne s'arrête jamais ou qu'il affiche l'erreur "maximum recursion depth exceeded" lors des différents appels récursifs on atteint jamais le cas n=0, le programme continue à s'appeler lui même avec des valeurs de n négatives qui tendent vers $-\infty$

Il faut s'assurer lors de l'écriture d'un programme récursif qu'il possède un cas d'arrêt et que ce cas d'arrêt est atteint.

De manière plus formelle, la terminaison est assurée lorsque l'on peut identifier dans le programme un entier positif qui décroît à chaque appel récursif et qui prend une valeur définissant un cas d'arrêt.

11 Récursivité terminale

Il s'agit d'un type de programme récursif particulier qui ont pour objectif d'être plus rapide lors de l'exécution de la pile d'appels récursifs.

Les programmes récursifs écrits utilisent en général l'appel récursif dans une opération ou comme argument d'une fonction. Par exemple dans le programme suivant l'appel récursif intervient dans une multiplication

[45]:
```
def puissance_recursive(x,n) :
    if n==0 :
        resultat = 1
        return resultat
    else :
        resultat = x*puissance_recursive(x,n-1)
        return resultat
```

Une version récursive terminale de ce programme ne fait pas intervenir l'appel récursif dans aucune opération ni comme argument de fonction, ce qui donnerait :

[46]:
```
def puissance_recursive(x,n) :
    def puissance_recursive_terminale(x,k,intermediaire) :
        if k == 0 :
            return intermediaire
        else :
            return puissance_recursive_terminale(x,k-1,intermediaire*x)
    return puissance_recursive_terminale(x,n,1)
```

On remarque dans ce programme que puissance_recursive_terminale n'est appelé que dans les return. Les valeurs renvoyées par l'appel récursif ne sont pas utilisées dans des opérations ou en argument d'autre fonction.

L'opération de récurrence est cette fois-ci effectuée en argument de la fonction récursive et non l'inverse.

Si on détaille l'ordre des appels récursifs de la version terminale on a :

- puissance_récursive appelle puissance_recursive_terminale(x,n,1) donc demande : "combien vaut $x^n \times 1$?"
- puissance_recursive_terminale(x,n,1) ne répond pas mais appelle puissance_recursive_terminale(x,n-1,1*x), donc demande : "combien vaut $x^{n-1} \times (1 \times x)$?"
- puissance_recursive_terminale(x,n-1,x) ne répond pas mais appelle puissance_recursive_terminale(x,n-2,x*x), donc demande : "combien vaut $x^{n-2} \times (x \times x)$?"

- puissance_recursive_terminale(x,n-2,x^2) ne répond pas mais appelle puissance_recursive_terminale(x,n-3,x^2*x), donc demande : "combien vaut $x^{n-3} \times (x^2 \times x)$?"
- etc
- jusqu'à - puissance_recursive_terminale(x,n-(n-1),x^{n-1}) ne répond pas, mais appelle puissance_recursive_terminale(x,0,x^{n-1}*x), donc demande : "combien vaut $x^0 \times (x^{n-1} \times x)$?"
- qui répond directement la valeur de x^n sans avoir à dépiler la pile d'appels, car la variable intermédiare contient déjà le résultat voulu.

11.0.1 exercice:

- Ecrire un programme itératif qui prend en entrée une fonction f, un entier n et un élément x et qui calcule $f^n(x) = fofofo...of(x) = f(f(f(...f(x)...)))$ soit la composé n-ième de f évaluée en x.
- Ecrire une version récursive de ce programme.
- Ecrire une version récursive terminale de ce programme.

[47]:
```
def composee_iterative(f,n,x):
    resultat = x
    for i in range(n):
        resultat = f(resultat)
    return resultat
```

[48]:
```
import numpy as np

for i in range(10):
    print(composee_iterative(np.sin,i,3.141592/2))
```

```
1.570796
0.9999999999999466
0.8414709848078676
0.7456241416655387
0.6784304773607261
0.6275718320491482
0.5871809965734222
0.5540163907556223
0.5261070755028354
0.5021706762685499
```

[49]:
```
def composee_recursive(f,n,x):
    if n == 0:
        resultat = x
        return resultat
    else :
```

```
        resultat = f(composee_recursive(f,n-1,x))
        return resultat
```

[50]:
```
import numpy as np

for i in range(10):
    print(composee_recursive(np.sin,i,3.141592/2))
```

```
1.570796
0.9999999999999466
0.8414709848078676
0.7456241416655387
0.6784304773607261
0.6275718320491482
0.5871809965734222
0.5540163907556223
0.5261070755028354
0.5021706762685499
```

[51]:
```
def composee_recursive_terminale(f,n,x):
    def composee_terminale(f,k,x,intermediaire) :
        if k == 0 :
            return intermediaire
        else :
            return composee_terminale(f,k-1,x,f(intermediaire))
    return composee_terminale(f,n,x,x)
```

[52]:
```
import numpy as np

for i in range(10):
    print(composee_recursive_terminale(np.sin,i,3.141592/2))
```

```
1.570796
0.9999999999999466
0.8414709848078676
0.7456241416655387
0.6784304773607261
0.6275718320491482
0.5871809965734222
0.5540163907556223
0.5261070755028354
0.5021706762685499
```