Récursivité

I. Introduction

I.1. Définition

Déf 1:

Une procédure récursive est une procédure qui s'appelle elle-même, une ou plusieurs fois.

Une procédure non récursive ne comporte donc que des appels à d'autres procédures.

La récursivité est un domaine très intéressant de l'informatique, un peu abstrait, mais très élégant ; elle permet de résoudre certains problèmes d'une manière très rapide, alors que si on devait les résoudre de manière itérative, il nous faudrait beaucoup plus de temps et de structures de données intermédiaires.

L'exemple le plus simple (même s'il n'est pas très intéressant) pour décrire le principe de la récursivité est celui de la fonction factorielle.

Celle-ci est définie de façon directe par :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ \prod_{i=1}^{n} i & \text{si } n \in \mathbb{N}^* \,. \end{cases}$$

Cela correspond au programme itératif suivant :

Algorithme 1 : Factorielle itérative

```
Données : n : entier naturel
Résultat : n!
def fact(n) :
    p ← 1 ;
    pour i de 2 à n faire
        p ← p * i
    finpour
    retourner p
fin
```

Mais on peut aussi définir $n!$ par récurrence :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n \in \mathbb{N}^* \,. \end{cases}$$

Cela correspond au programme récursif suivant :

Algorithme 2 : Factorielle récursive

```
Données : n : entier naturel
Résultat : n!
def fact(n) :
    si n=0 alors
        retourner 1
    sinon
        retourner n *fact(n − 1 )
    finsi
fin
```

Une procédure récursive doit toujours comporter un ou plusieurs cas « de base » qui sont définis de manière non récursive (le cas $n = 0$ dans l'exemple ci-dessus).

I.2. Éxécution d'une procédure récursive

La récursivité utilise toujours la pile du programme en cours.
On appelle « pile » une zone mémoire réservée à chaque programme ; sa taille peut être fixée manuellement par l'utilisateur. Son rôle est de stocker les variables locales et les paramètres d'une procédure.

Dans une procédure récursive, toutes les variables locales ainsi que les adresses de retour des fonctions sont stockées dans la pile, et empilées autant de fois qu'il y a d'appels récursifs. Donc la pile se remplit progressivement, et si on ne fait pas attention on arrive à un « débordement de pile » (stack overflow). En Python la taille de la pile est fixée à 1000 ; mais on peut la modifier par les instructions :

```
import sys
sys.setrecursionlimit(10000)
```

Lorsque la condition d'arrêt est remplie, les variables sont désempilées.
Le programme ci-dessous permet de visualiser la pile des variables locales et des appels lors de l'exécution de la factorielle récursive :

```
from inspect import *

def factoVisu(n):
    print(locals())
    if n==0:
        pile = stack()
        for i in range(len(pile)):
            print(getframeinfo(pile[i][0]))
        return 1
    else:
        res = n*factoVisu(n-1)
        print(locals())
        return res

print(factoVisu(5))
```

```
{'n': 5}
{'n': 4}
{'n': 3}
{'n': 2}
{'n': 1}
{'n': 0}
Traceback(filename='pythontex-files-recursivite\\sympy_default_default.py', lineno=62, function='fa
Traceback(filename='pythontex-files-recursivite\\sympy_default_default.py', lineno=65, function='fa
Traceback(filename='pythontex-files-recursivite\\sympy_default_default.py', lineno=65, function='fa
Traceback(filename='pythontex-files-recursivite\\sympy_default_default.py', lineno=65, function='fa
Traceback(filename='pythontex-files-recursivite\\sympy_default_default.py', lineno=65, function='fa
Traceback(filename='pythontex-files-recursivite\\sympy_default_default.py', lineno=65, function='fa
Traceback(filename='pythontex-files-recursivite\\sympy_default_default.py', lineno=69, function='<m
{'n': 1, 'res': 1}
{'n': 2, 'res': 2}
{'n': 3, 'res': 6}
{'n': 4, 'res': 24}
{'n': 5, 'res': 120}
120
```

I.3. Efficacité

- Un programme récursif **bien conçu** est un peu plus long à l'exécution que sa version itérative correspondante, d'autant plus que celle-ci est simple, comme le montre l'exemple suivant où l'on a calculé les termes de la suite définie par $u_{n+1} = \sqrt{1+u_n}$:

```
import timeit
from math import sqrt

import sys
```

```
sys.setrecursionlimit(10000)

def iteratif(n):
    u = 0
    for i in range(n):
        u = sqrt(1 + u)
    return u

def recursif(n):
    if n == 0:
        return 1
    else:
        return sqrt(1 + recursif(n-1))

print(timeit.timeit(stmt = 'iteratif(1000)', setup = 'from __main__ import iteratif', \
                                  number = 10000))

print(timeit.timeit(stmt = 'recursif(1000)', setup = 'from __main__ import recursif',\
                                   number = 10000))
1.6906291449368072
3.617282511664553
```

- Cela peut être encore pire si l'on abuse de la récursivité. Considérons par exemple la suite de Fibonacci définie par :

$$u_0 = 0 \quad , \quad u_1 = 1 \quad \text{et} \quad \forall n \in \mathbb{N},\ u_{n+2} = u_{n+1} + u_n$$

En voici une version itérative :

Algorithme 3 : Suite de Fibonacci itérative

Données : n : entier naturel
Résultat : n^{e} terme de la suite de Fibonacci

def *fibo(n)* **:**
  si $n = 0$ **alors**
    retourner *0*
  sinon
    $u_0 \leftarrow 0$;
    $u_1 \leftarrow 1$;
    /* à l'étape i, u_0 vaut u_i et u_1 vaut u_{i+1} */;
    pour i *de* 1 *à* $n-1$ **faire**
      $u_0, u_1 \leftarrow u_1, u_0 + u_1$
    finpour
    retourner u_1
  finsi
fin

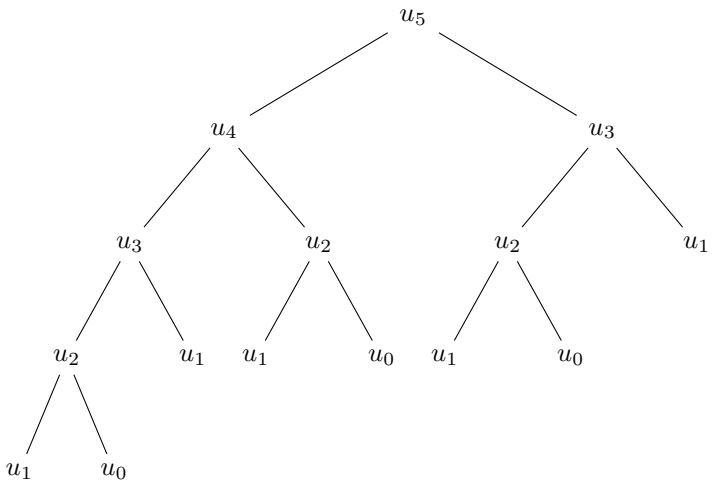
et la version récursive :

Algorithme 4 : Suite de Fibonacci récursive

Données : n : entier naturel
Résultat : n^{e} terme de la suite de Fibonacci

def *fibo(n)* **:**
  si $n = 0$ **alors**
    retourner *0*
  sinon si $n = 1$ **alors**
    retourner *1*
  sinon
    retourner $\mathit{fibo}(n-1) + \mathit{fibo}(n-2)$
  finsi
fin

Voici un arbre qui représente l'exécution de ce programme pour $n = 5$.



Pour calculer u_5, il faut d'abord calculer u_4 et u_3 . Or pour calculer u_4, il faut calculer u_3 etc... Puisque les appels récursifs sont indépendants, la valeur de u_2 par exemple va être calculée 3 fois. Pour des valeurs de n supérieures, on s'aperçoit qu'un grand nombre de valeurs des u_i sont calculées plusieurs fois.

Plus précisément, notons $A(n)$ le nombre d'additions effectuées par la version récursive. $A(n)$ vérifie :

$$A(0) = A(1) = 0 \quad \text{et} \quad \forall\, n \geqslant 2,\ A(n) = A(n-1) + A(n-2) + 1\,.$$

La suite a_n de terme général $a_n = A(n) + 1$ vérifie donc les relations :

$$a_0 = a_1 = 1 \quad \text{et} \quad \forall\, n \geqslant 2,\ a_n = a_{n-1} + a_{n-2}\,,$$

ce qui permet de trouver facilement (récurrence linéaire d'ordre 2) :

$$\forall\, n \in \mathbb{N},\ A(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right) + 1\,.$$

On voit donc que $A(n) \underset{n\to+\infty}{\sim} \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n+1}$ (avec $\frac{1+\sqrt{5}}{2} \approx 1,618$), alors que le nombre d'additions dans la version itérative est tout simplement égal à $n-1$ ...

- Il existe cependant une implémentation du calcul du n-ième élément de la suite de Fibonacci qui contourne cette difficulté.

Algorithme 5 : Suite de Fibonacci récursive améliorée

Données : n : entier naturel, a et b : valeurs des deux termes précédents de la suite
Résultat : n^{e} terme de la suite de Fibonacci

```
def fibo(n,a,b) :
    si n = 0 alors
        retourner a
    sinon si n = 1 alors
        retourner b
    sinon
        retourner fibo(n − 1, b, a + b)
fin
```

La fonction fibo(n,a,b) ci-dessus renvoie le n^{e} élément de la suite $u_n(a,b)$ définie par :

$$u_n(a,b) = \begin{cases} a & \text{si } n = 0 \\ b & \text{si } n = 1 \\ u_{n-1}(a,b) + u_{n-2}(a,b) & \text{sinon}\,. \end{cases}$$

Pour obtenir le n^{e} terme de la suite de Fibonacci classique, il suffit donc d'appeler fibo$(n,0,1)$. Dans cette version, les valeurs utiles ne sont calculées qu'une seule fois. On a utilisé pour cela la relation :

$$\forall\, n \geqslant 1,\ u_n(a,b) = u_{n-1}(b, a+b)\,.$$

On remarque que dans ce dernier programme l'appel récursif est la dernière instruction exécutée : il n'y a pas de traitement du résultat de l'appel récursif. Un tel programme est dit *récursif terminal*. Il y a une différence fondamentale entre la version récursive et la version récursive terminale : dans la version récursive simple (algorithme 4 ou algorithme 2) , l'appel récursif n'est pas la dernière opération réalisée (il est utilisé dans un calcul), il faut donc que la machine garde en mémoire l'état de son calcul. Cela conduit aux problèmes de mémoire et de temps de calcul cités précédemment. Dans la version récursive terminale, l'appel récursif est la dernière action réalisée : l'interpréteur ou le compilateur sont capables de détecter ces situations et traitent alors le programme comme un programme itératif, ce qui n'engendre pas les problèmes cités précédemment.

- Une autre idée pour éviter les appels redondants est la mémoïsation. Cela consiste à mémoriser au fur et à mesure, dans une variable globale, les valeurs prises par la fonction. Cela consomme malheureusement beaucoup de place mémoire !

  Dans la version ci-dessous, on a utilisé une liste pour stocker les valeurs calculées au fur et à mesure ; au moment de calculer fibo(n), on commence par vérifier s'il existe dans la liste une valeur correspondant à n ; si oui, on renvoie directement la valeur stockée, sinon on la calcule récursivement et on stocke cette nouvelle valeur dans la liste.

Algorithme 6 : Suite de Fibonacci récursive avec mémoïsation

Données : n : entier naturel, a et b : valeurs des deux termes initiaux de la suite
Résultat : n^{e} terme de la suite de Fibonacci

```
def fibo_memo(n) :
    liste = [None] * (n + 1);
    liste[0] = 0 ;
    liste[1] = 1 ;
    def fibo(n) :
        si liste[n] ≠ None alors
            retourner liste[n]
        sinon
            x = fibo_memo(n − 1) + fibo_memo(n − 2) ;
            liste[n] = x ;
            retourner x
        finsi
    fin
    retourner fibo(n)
fin
```

II. Exemples d'algorithmes récursifs

II.1. Exponentiation rapide

Étant donné une opération $\star$ sur des objets (par exemple des réels, des matrices...), l'objectif est de calculer $x^n = x \star x \star \cdots \star x$ en un nombre minimal d'opérations. On suppose que l'opération $\star$ est associative.
Un premier algorithme naïf donne un coût en nombre d'opérations en $O(n)$.

Algorithme 7 : Exponentiation naïve

Données : x : objet d'un certain type, et n : entier naturel
Résultat : la valeur de x^n

```
def puiss(x,n) :
    si n=0 alors
        retourner 1
    sinon
        y = x;
        pour i de 2 à n faire
            y = y ⋆ x
        finpour
        retourner y
    finsi
fin
```

II.1.1. Exponentiation rapide, 1ère version

En utilisant les relations :

$$x^{2n} = (x \star x)^n \quad \text{et} \quad x^{2n+1} = x \star (x \star x)^n$$

on peut écrire l'algorithme récursif suivant :

Algorithme 8 : Exponentiation rapide, récursive, 1re version

```
Données : x : objet d'un certain type, et n : entier naturel
Résultat : la valeur de x^n
def puiss(x,n) :
    si n=0 alors
        retourner 1
    sinon
        si n est pair alors
            retourner puiss(x ⋆ x , n//2)
        sinon
            retourner x ⋆ puiss(x ⋆ x , n//2)
        finsi
    finsi
fin
```

Cherchons le coût de cet algorithme en nombre d'appels à des opérations « élémentaires ».
Pour cela on démontre facilement par récurrence sur k la propriété :

$$\mathcal{P}_k : \text{ si } 2^k \leqslant n < 2^{k+1}, \text{ il y a } k+1 \text{ appels récursifs}$$

ce qui prouve que le nombre d'appels est égal à $\lfloor \log_2(n) \rfloor + 1$. Puisqu'il y a un test, une division entière et au maximum 2 multiplications par appel, le nombre d'opération est un $O(\log_2(n))$.

Pour écrire une version itérative de cet algorithme, nous allons relier son fonctionnement à la l'écriture en base 2 de l'exposant n.
Par exemple, si $n = 89$, n s'écrit en base 2 : $n = 1 + 8 + 16 + 64 = \overline{1011001}$. Et l'on a :

$$x^{89} = x \star x^8 \star x^{16} \star x^{64}\,.$$

La suite x, x^2, x^4, x^8, x^{16}, x^{32}, x^{64} s'obtient par des élévations au carré successives ; le résultat final est le produit de ceux des termes de cette suite qui correspondent à un chiffre 1 dans l'écriture en base 2 de n.

Cela donne l'algorithme itératif suivant :

Algorithme 9 : Exponentiation rapide, itérative

```
Données : x : objet d'un certain type, et n : entier naturel
Résultat : la valeur de x^n
def puiss(x,n) :
    p = 1;
    tant que n > 0 faire
        si n impair alors
            p = x * p
        finsi
        x = x ⋆ x;
        n = n//2
    fintq
    retourner p
fin
```

II.1.2. Exponentiation rapide, 2ème version

En utilisant les relations :

$$x^{2n} = x^n \star x^n \quad \text{et} \quad x^{2n+1} = x \star x^n \star x^n$$

on peut écrire l'algorithme récursif suivant :

Algorithme 10 : Exponentiation rapide, récursive, 2e version

Données : x : objet d'un certain type, et n : entier naturel
Résultat : la valeur de x^n

```
def puiss(x,n) :
    si n=0 alors
        retourner 1
    sinon
        y = puiss(x, n//2);
        si n est pair alors
            retourner y ⋆ y
        sinon
            retourner x ⋆ y ⋆ y
        finsi
    finsi
fin
```

Une version itérative de cet algorithme est plus difficile. On utilise ici encore l'écriture binaire de n, mais cette fois-ci on la lit de gauche à droite. Par exemple, dans le cas de $n = 89 = 1 + 8 + 16 + 64 = \overline{1011001}$, on écrira

$$x = x^{64} \star x^{16} \star x^8 \star x\,.$$

Dans cette écriture, x^{64} correspond à x élevé au carré 6 fois, 6 correspondant à la position du 1 de gauche, puis x^{16} correspond à x élevé au carré 4 fois, 4 correspondant à la position du second 1 à partir de la gauche etc... Cela donne le programme itératif suivant :

Algorithme 11 : Exponentiation rapide, itérative, 2e version

Données : x : objet d'un certain type, et n : entier naturel
Résultat : la valeur de x^n

```
def puiss(x,n) :
    /* on commence par chercher l'écriture de n en base 2        */;
    liste = vide;
    tant que n ≠ 0 faire
        si n pair alors
            ajouter 0 à la fin de liste
        sinon
            ajouter 1 à la fin de liste
        finsi
        n = n//2;
    fintq
    p = 1;
    tant que longueur(liste)> 0 faire
        p = p ⋆ p;
        b =liste.pop();
        /* dernier élément de la liste, que l'on supprime        */;
        si b = 1 alors
            p = p ⋆ x
        finsi
    fintq
fin
```

Voici les programmes Python correspondant avec les tests de durée.

```python
from time import time

import sys
sys.setrecursionlimit(10000)

nmax = 2000 # pour les test de durée

# Versions récursives d'abord

def puiss_rec1(x, n):
```

```
11      if n == 0:
12          return 1
13      else:
14          return x * puiss_rec1(x, n-1)
15
16  def puiss_rec2(x, n):
17      if n == 0:
18          return 1
19      else:
20          if n%2 ==0:
21              return puiss_rec2(x*x, n//2)
22          else:
23              return x * puiss_rec2(x*x, n//2)
24
25  def puiss_rec3(x, n):
26      if n == 0:
27          return 1
28      else:
29          y = puiss_rec3(x, n//2)
30          if n%2 == 0:
31              return y*y
32          else:
33              return x*y*y
34
35  print("Calcul des i^j par diférentes méthodes:\n")
36  debut = time()
37  for i in range(1,20):
38      for j in range(0,nmax):
39          puiss_rec1(i,j)
40  print("Récursif primaire", "i<=20 et j<=",nmax, "--->", time() - debut)
41
42  debut = time()
43  for i in range(1,100):
44      for j in range(0,nmax):
45          puiss_rec2(i,j)
46  print("Récursif 1ère version", "i<=100 et j<=",nmax, "--->",time() - debut)
47
48  debut = time()
49  for i in range(1,100):
50      for j in range(0,nmax):
51          puiss_rec3(i,j)
52  print("Récursif 2ème version", "i<=100 et j<=",nmax, "--->",time() - debut)
53
54  # Versions itératives correspondantes
55
56  def puiss_iter1(x, n):
57      p = 1
58      for i in range(0,n):
59          p *= x
60      return p
61
62  def puiss_iter2(x, n):
63      res = 1
64      var = x
65      while n!= 0:
66          if n%2 != 0:
67              res *= var
68          var *= var
69          n = n//2
70      return res
71
```

```python
def puiss_iter3(x, n):
    if n==0:
        return 1
    # écriture de n en base 2
    liste_bin = []
    while n!= 0:
        liste_bin.append(n%2)
        n = n//2
    # calcul
    liste_bin.pop()
    # le premier chiffre est forcéement un 1
    res = x
    while len(liste_bin) !=0:
        res *= res
        if liste_bin.pop() == 1:
            res *= x
    return res

debut = time()
for i in range(1,20):
    for j in range(0,nmax):
        puiss_iter1(i,j)
print("Itératif primaire", "i<=20 et j<=",nmax, "--->",time() - debut)

debut = time()
for i in range(1,100):
    for j in range(0,nmax):
        puiss_iter2(i,j)
print("Itératif 1ère version", "i<=100 et j<=",nmax, "--->",time() - debut)

debut = time()
for i in range(1,100):
    for j in range(0,nmax):
        puiss_iter3(i,j)
print("Itératif 2ème version", "i<=100 et j<=",nmax, "--->",time() - debut)
```

```
Calcul des i^j par diférentes méthodes:

Récursif primaire i<=20 et j<= 2000 ---> 15.54701852798462
Récursif 1ère version i<=100 et j<= 2000 ---> 5.559236288070679
Récursif 2ème version i<=100 et j<= 2000 ---> 2.73252010345459
Itératif primaire i<=20 et j<= 2000 ---> 6.501466751098633
Itératif 1ère version i<=100 et j<= 2000 ---> 5.050424814224243
Itératif 2ème version i<=100 et j<= 2000 ---> 2.6194374561309814
```

II.2. Permutations d'un ensemble

Soit E un ensemble.
On veut créer une liste formée de toutes les permutations de E, et pour cela on procède ainsi : pour chaque élément $x \in E$, on crée la liste des permutations de $E \setminus \{x\}$, et on rajoute x à la fin de chacune d'elles.
Cela donne le programme Python suivant :

```python
def permutations(E):
    n = len(E)
    if n == 1:
        return [E]
    else:
        liste_permus = []
        for i in range(0, n):
            C = E[:i] + E[i+1:]
            L = permutations(C)
            for perm in L:
                perm.append(E[i])
                liste_permus.append(perm)
        return liste_permus

print(len(permutations([ 1, 2, 3, 4, 5, 6, 7, 8,9,10 ])))
print(permutations(['a','b','c','d']))
```

```
3628800
[['d', 'c', 'b', 'a'], ['c', 'd', 'b', 'a'], ['d', 'b', 'c', 'a'], ['b', 'd', 'c', 'a'], ['c', 'b',
```

II.3. Tours de Hanoï

On dispose d'une pile de n disques de tailles différentes et de trois tours positionnées l'une à côté de l'autre. Les disques sont disposés sur la tour de gauche par taille décroissante et on veut les amener tous sur la tour de droite dans le même ordre, en utilisant éventuellement la tour du milieu. Il faut cependant respecter une règle : à chaque coup, tous les disques doivent se trouver sur une tour et un disque ne peut pas être posé sur un disque de diamètre inférieur

On peut aborder le problème de la façon suivante : pour envoyer une tour de taille n de la place de gauche à la place de droite :

- on envoie les $n-1$ premiers disques sur la place du milieu, ce qui revient à résoudre le jeu pour $n-1$ disques avec le départ à gauche et l'arrivée au milieu ;
- on envoie le disque n à droite ;
- on envoie les $n-1$ premiers disques sur la place de droite, ce qui revient à résoudre le jeu pour $n-1$ disques avec le départ au milieu et l'arrivée à droite.

Ainsi la résolution pour n disques fait appel à la résolution pour $n-1$ disques, deux fois. Construire un algorithme récursif est alors très simple, mais il nous manque notre condition d'arrèt : résoudre le problème à 0 disque consiste à ne rien faire. On peut donc proposer l'algorithme suivant :

Algorithme 12 : Tours de Hanoï récursif

Données : n : entier indiquant le nombre de disques à déplacer,
dep, int, arr : chaines de caractères pour indiquer les tiges de départ, d'arrivée et intermédiaire
Résultat : pas de sortie, on affiche les déplacements au fur et à mesure

```
def Hanoi(n, dep, int, arr) :
    si n=0 alors
        retourner rien
    sinon
        Hanoi(n-1, dep, arr, int);
        afficher "déplacement d'un disque de " dep " vers " arr ;
        Hanoi(n-1, int, dep, arr)
    finsi
fin
```

Par exemple, en appelant ce programme avec les arguments (3, "*gauche*", "*milieu*", "*droite*") l'affichage produit est le suivant : *gauche* → *droite* ; *gauche* → *milieu* ; *droite* → *milieu* ; *gauche* → *droite* ; *milieu* → *gauche* ; *milieu* → *droite* ; *gauche* → *droite*.

Pour résoudre le problème à n tours, on a besoin de résoudre deux fois le problème à $n-1$ tours, et d'effectuer une opération supplémentaire. La relation de récurrence sur le nombre d'opérations T_n est donc :

$$T_n = 2T_{n-1} + 1 \quad \text{soit} \quad T_n + 1 = 2(T_{n-1} + 1)$$

dont la solution, étant donné que $T_0 = 0$, vaut $T_n = 2^n - 1$.

II.4. Évaluation d'un nombre écrit en chiffres romains

Les chiffres romains sont :

M=1000 D=500 C=100 L=50 X=10 V =5 I=1

Les nombres inférieurs à 10 s'écrivent :

I =I 2 =II 3=II 4=IV 5=V 6=VI 7=VII 8= VIII 9=IX X=10

Tous les nombres finissant par 1, 2 ou 3 se terminent dans l'écriture romaine par I, II ou III. Idem pour 6, 7 ou 8.
Cela est valable aussi bien pour les unités que pour les dizaines et centaines ; par exemple, 30 s'écrit XXX, 300 s'écrit CCC.
Tous les nombres finissant par 4 se terminent par IV. Ceux finissant par 40 se terminent par XL etc...
Tous les nombres finissant par 9 se terminent par IX. Ceux finissant par 90 se terminent par XC etc...
Exemples :

47 = XLVII 97 = CXVII 149 = CXLIX 1990 = MCM 1999 = MCMXCIX 2016 = MMXVI

On veut écrire un algorithme permettant de donner la valeur d'un nombre écrit en chiffres romains. Le principe est le suivant :

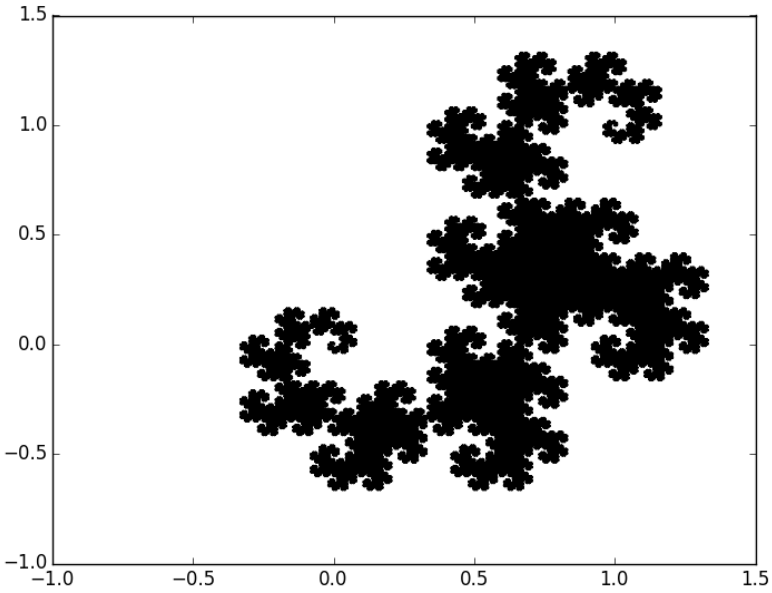
- si le nombre romain a un seul chiffre, sa valeur est celle correspondante ;
- sinon, si le 1er chiffre a une valeur inférieure au 2e, alors on soustrait sa valeur de tout le reste, sinon on l'additionne.

À faire :

1. Écrire le programme récursif correspondant.
2. Écrire un programme qui réalise l'opération inverse (nombre en base 10 $\longrightarrow$ nombre en chiffres romains).

II.5. La courbe du dragon

À partir d'un segment $[AB]$, on construit le point C tel que le triangle ACB dans le sens direct soit isocèle et rectangle en C, et on recommence le procédé pour les segments $[AC]$ et $[BC]$, etc...
La courbe du dragon à la n^e étape est la réunion de courbe du dragon d'ordre $n-1$ construite sur $[AC]$ et de celle d'ordre $n-1$ construite sur $[BC]$.
La figure ci-dessous représente la courbe obtenue pour $n = 20$, $A(0,0)$ et $B(1,1)$.



Indications : On démontrera d'abord que si A a pour coordonnées (x, y) et B pour coordonnées (z, t) les coordonnées de C sont (u, v) avec :

$$u = \frac{x - y + z + t}{2} \quad \text{et} \quad v = \frac{x + y - z + t}{2}.$$

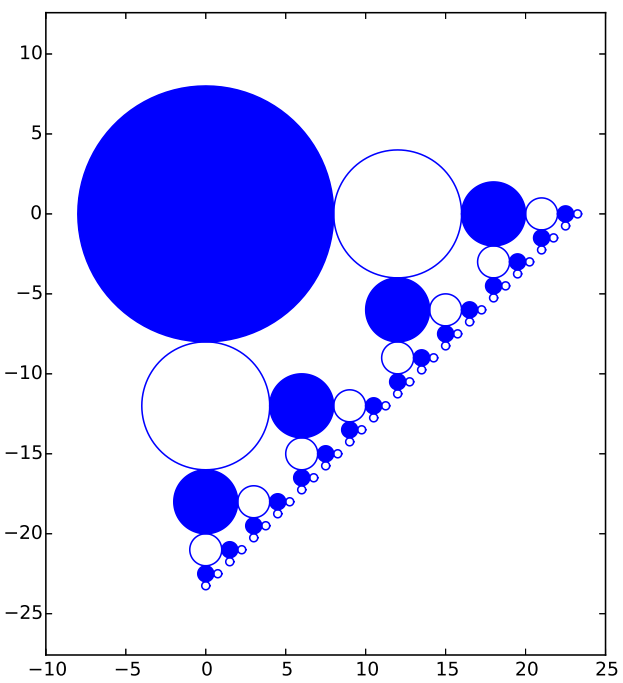
II.6. D'autres courbes

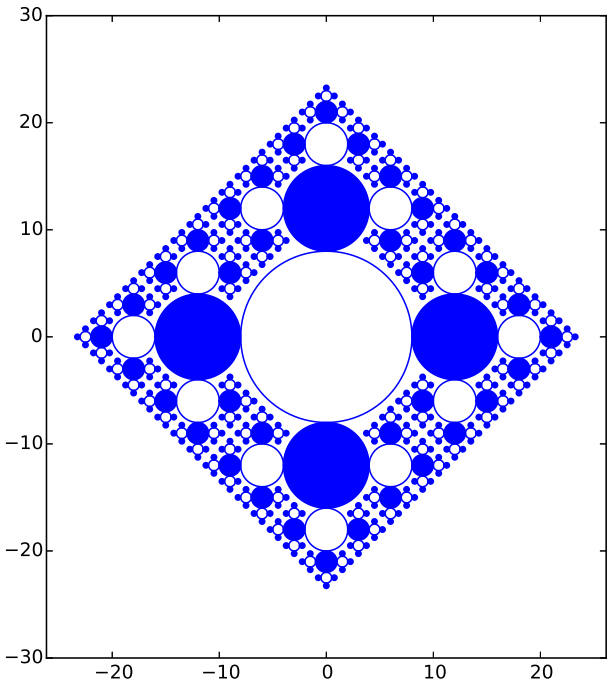
II.6.1. Bubbles

On donne une fonction `circle([x, y], r, rempli)` qui trace à l'écran un cercle (si `rempli == False`) ou un disque (si `rempli == True`) de centre le point de coordonnées (x, y) et de rayon r.

```
import math
import matplotlib.pyplot as plt

def circle( coords, r, rempli=False):
    X, Y = [], []
    for t in range(101):
        X.append(coords[0] + r*math.cos(t*math.pi/50))
        Y.append(coords[1] + r*math.sin(t*math.pi/50))
    if rempli:
        plt.fill(X, Y, color='blue')
    else:
        plt.plot(X, Y, color='blue')
```

Écrire à partir de là deux fonctions récursives permettant d'obtenir les figures ci-dessous.





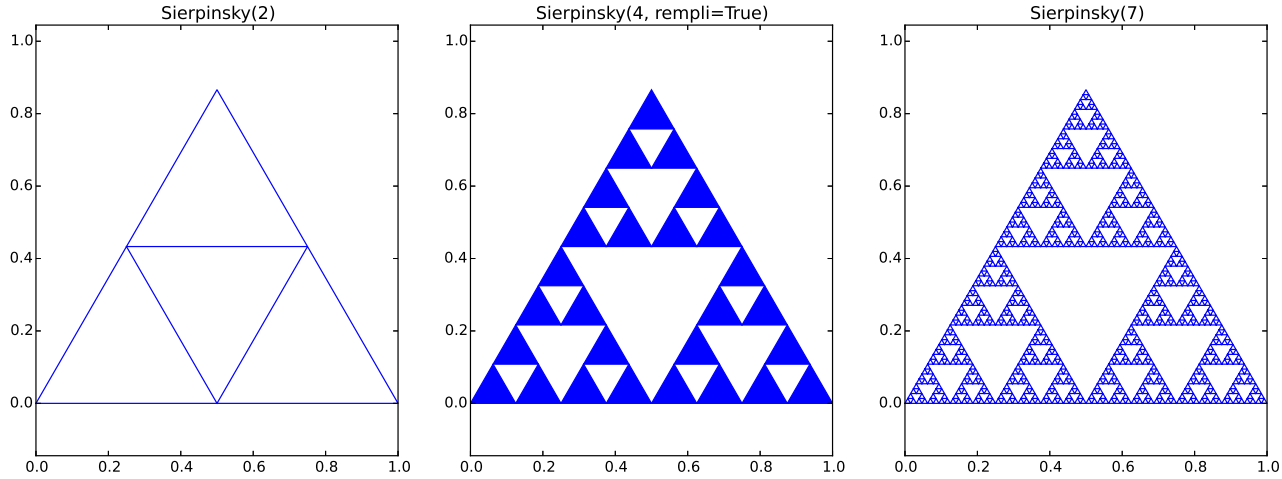
II.6.2. Triangle de Sierpinski

On donne une fonction `polygone(liste)` qui trace le polygone (plein ou non) dont la liste des sommets est donnée.

```
import math
import matplotlib.pyplot as plt

def polygone(rempli=False, *args ):
     X, Y = [], []
     for point in args:
         X.append(point[0])
         Y.append(point[1])
     X.append(args[0][0])
     Y.append(args[0][1])
```

```
        if rempli:
            plt.fill(X, Y, color='blue')
        else:
            plt.plot(X, Y, color='blue')
```

Écrire une fonction permettant de réaliser les graphiques ci-dessous (les triangles sont équilatéraux).



III. La méthode Diviser Pour Régner

III.1. Généralités

Les algorithmes de type *Diviser Pour Régner* sont très classiques en informatique. Les meilleurs algorithmes de tri fonctionnent par exemple sur le principe Diviser pour régner (tri rapide, tri fusion). De manière générale, ou peut décrire un algorithme de type diviser pour régner de la façon suivante.

Déf 2:

Un algorithme est de type Diviser Pour Régner s'il comprend les étapes suivantes :

- condition d'arrêt ;
- découpage du problème en sous-problèmes ;
- traitement des sous-problèmes ;
- rassemblement des solutions aux sous-problèmes en une solution au problème principal.

Le tri fusion par exemple respecte cette philosophie : le problème principal est coupé exactement en deux, chaque sous-liste est triée, puis on effectue une fusion des deux sous-listes triées pour obtenir la liste principale triée. Dans le cas du tri fusion, le découpage ne coûte rien, mais la recombinaison est coûteuse. à l'inverse dans le tri rapide, le découpage est coûteux (on sépare la liste autour d'un pivot), mais la recombinaison est immédiate.

III.2. Multiplication rapide de polynômes-1re version

Il s'agit d'écrire un algorithme pour multiplier deux polynômes de degré $\leqslant n-1$ $(n \geqslant 1)$. Un tel polynôme $P = \sum_{k=0}^{n-1} a_k X^k$ sera représenté par la liste $[a_0, a_1, \ldots, a_{n-1}]$ de longueur n.

La méthode naïve consistant à utiliser les formules :

$$\text{si} \qquad P = \sum_{k=0}^{n-1} a_k X^k \quad \text{et} \quad Q = \sum_{k=0}^{n-1} b_k X^k$$

$$\text{alors} \qquad PQ = \sum_{k=0}^{2n-2} c_k X^k \quad \text{avec} \quad c_k = \sum_{i+j=k} a_i b_j$$

a une complexité en $\mathcal{O}(n^2)$: en effet, il faut faire $2n-2$ additions et $2 \times (1+2+\cdots+(n-1)) + n = n^2$ multiplications.

Nous exposons ci-dessous l'algorithme de Karatsuba (1960), qui a une complexité en $\mathcal{O}(n^{\log_2 3}) \approx \mathcal{O}(n^{1,58})$.
On suppose dans un 1er temps que $n = 2m$ est pair. Pour $P,\ Q \in \mathbb{K}_{n-1}[X]$, on écrit :

$$\begin{cases} P = P_1 + X^m P_2 \\ Q = Q_1 + X^m Q_2 \end{cases}$$

avec $P_i,\ Q_i \in \mathbb{K}_{m-1}[X]$. Soient :

$$R_1 = P_1 Q_1 \quad , \quad R_2 = P_2 Q_2 \quad \text{et} \quad R_3 = (P_1 + P_2)(Q_1 + Q_2)\,.$$

On a alors :

$$PQ = R_1 + (R_3 - R_2 - R_1)X^m + X^{2m} R_2$$

On ne réalise ainsi que 3 multiplications de polynômes de tailles moitié (on ne compte pas les multiplications par X^m et X^{2m} qui reviennent simplement à décaler les indices) ; on a évidemment fait des additions supplémentaires, mais l'on considère le coût d'une addition de polynômes comme négligeable devant celui d'une multiplication (en effet, c'est en $\mathcal{O}(n)$ contre $\mathcal{O}(n^2)$).
Puisque les additions et les multiplications par X^m ou X^{2m} prennent un temps proportionnel à n, on peut dire que le temps d'exécution $T(n)$ vérifie une relation de la forme :

$$T(n) = 3T\left(\frac{n}{2}\right) + an\,\cdot$$

On utilise alors le théorème général suivant.

Théorème 1:

Si T vérifie la relation de récurrence $T(n) = aT\left(\frac{n}{b}\right) + \alpha n^c$, avec a, b et c des entiers positifs et α un réel positif, alors :

- si $c < \log_b(a)$, on a $T(n) = \mathcal{O}(n^{\log_b(a)})$;
- si $c = \log_b(a)$, on a $T(n) = \mathcal{O}(n^c \log_b(n))$;
- $c > \log_b(a)$, on a $T(n) = \mathcal{O}(n^c)$.

☞ *Démonstration:*

- Supposons dans un 1er temps que n est une puissance de b : $n = b^m$.
  On a alors $T(b^m) = aT(b^{m-1}) + \alpha b^{mc}$.
  En divisant le tout par a^m (on suppose $a \neq 0$), on obtient $\frac{T(b^m)}{a^m} = \frac{T(b^{m-1})}{a^{m-1}} + \frac{\alpha b^{mc}}{a^m}$, soit encore $\frac{T(b^m)}{a^m} - \frac{T(b^{m-1})}{a^{m-1}} = \frac{\alpha b^{mc}}{a^m}$.
  En sommant ces relations pour $k \leqslant m$, on a une somme télescopique dont le résultat est :

$$\frac{T(b^m)}{a^m} - T(1) = \alpha \sum_{k=2}^{m} \frac{b^{kc}}{a^k}.$$

  soit :

$$T(b^m) = a^m T(1) + \alpha a^m \times \sum_{k=2}^{m} \left(\frac{b^c}{a}\right)^k .$$

  Trois cas se distinguent :
  - Si $c = \log_b(a)$: dans ce cas $b^c = a$ et la somme vaut $m-1$ et on a $T(b^m) = a^m T(1) + \alpha(m-1)a^m$, ce qui nous donne puisque $m = \log_b(n)$:
    $$T(n) = \mathcal{O}(\log_b(n) a^{\log_b(n)}) = \mathcal{O}(n^c \log_b(n)).$$
  - Si $c < \log_b(a)$: dans ce cas, la somme converge et on a $T(b^m) = \mathcal{O}(a^m)$ soit encore $T(n) = \mathcal{O}(a^{\log_b(n)}) = \Theta(n^{\log_b(a)})$.
  - Si $c > \log_b(a)$: alors la somme diverge donc est équivalente quand $m \to +\infty$ à $\left(\frac{b^c}{a}\right)^m$ et on a $T(b^m) = \mathcal{O}(a^m \times (\frac{b^c}{a})^m)$ soit encore $T(n) = \mathcal{O}(n^c)$.
- Dans le cas général, on encadre n entre deux puissances successives de b et on obtient les mêmes résultats (car la fonction T est croissante, comme on peut le vérifier par récurrence).

Ainsi dans le cas de l'algorithme de Karatsuba : $a = 3$, $b = 2$ et $c = 1$, et $\log_2 3 \approx 1,58 > 1$, de sorte que

$$T(n) = \mathcal{O}(n^{\log_2 3}) \approx \mathcal{O}(n^{1,58})$$

L'algorithme est décrit ci-dessous.

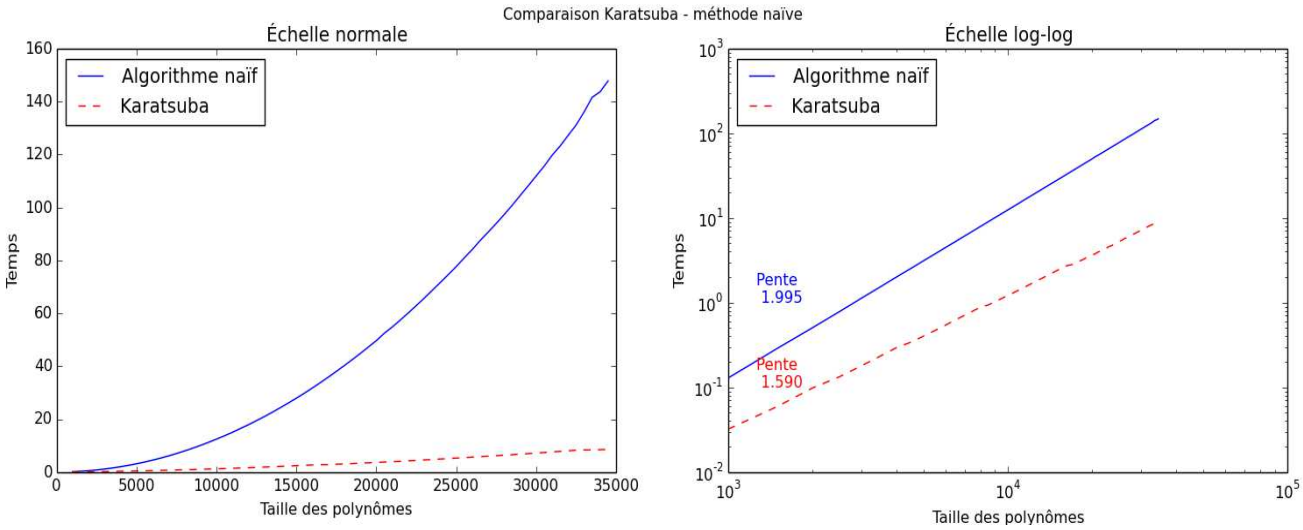
Algorithme 13 : Algorithme de Karatsuba

Données : Deux polynômes P et Q de degrés $\leqslant n-1$ représentés par des listes de longueur n
Résultat : Le produit PQ

```
def Karatsuba(P,Q) :
    si n=1 alors
        retourner PQ
    sinon
        m = n//2;
        Décomposer P = P_1 + X^m P_2 et Q = Q_1 + X^m Q_2;
        R_1 =Karatsuba(P_1, Q_1);
        R_2 =Karatsuba(P_2, Q_2);
        Former P_3 = P_1 + P_2 et Q_3 = Q_1 + Q_2;
        R_3 =Karatsuba(P_3, Q_3);
        retourner R_1 + X^m(R_3 - R_2 - R_1) + X^{2m} R_2
    finsi
fin
```

La figure ci-dessous illustre l'écart des performances ; on a également fait un tracé des temps en utilisant une échelle logarithmique : si $T(n) \approx an^\alpha$ alors $\ln(T(n))$ est une fonction affine de $\ln n$, de pente α.

On remarque que les pentes calculées de ces droites sont très proches des valeurs théoriques (pour calculer ces pentes, j'ai utilisé le module `stats` de `scipy` et la fonction `linregress`).



III.3. L'algorithme de Strassen

Une stratégie Diviser pour Régner peut être appliquée par exemple au produit de deux matrices comme le montre l'algorithme de Strassen (1969). Lorsque l'on fait un produit matriciel classique, entre deux matrices de taille n, il faut de l'ordre de n^3 opérations élémentaires (multiplications, additions). Les formules mathématiques ne laissent guère de possibilités.

L'idée de l'algorithme de Strassen est de faire ce produit par blocs, mais d'une façon bien particulière. Si on cherche par exemple à effectuer le produit des matrices de taille n (n pair) $A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}$ et $B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$ et que l'on note $C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$ le produit, faire un produit par bloc classique demande 8 multiplications matricielles de matrices carrées de taille $\dfrac{n}{2}$ et 4 sommes, ce qui n'apporte aucun avantage en termes de complexité :

$$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \qquad C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$

$$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \qquad C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

Mais Strassen a eu l'idée de définir des matrices intermédiaires, au nombre de 7. L'idée principale est que la somme de matrices demande un nombre d'opérations plus faible que la multiplication et qu'il faut donc réduire

le nombre de multiplications, quitte à augmenter le nombre d'additions. Les matrices intermédiaires sont les suivantes :

$$M_1 = (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \qquad M_2 = (A_{2,1} + A_{2,2})B_{1,1} \qquad M_3 = A_{1,1}(B_{1,2} - B_{2,2})$$
$$M_4 = A_{2,2}(B_{2,1} - B_{1,1}) \qquad M_5 = (A_{1,1} + A_{1,2})B_{2,2} \qquad M_6 = (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2})$$
$$M_7 = (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})$$

On récupère alors la matrice C grâce aux relations :

$$C_{1,1} = M_1 + M_4 - M_5 + M_7$$
$$C_{2,1} = M_2 + M_4$$
$$C_{1,2} = M_3 + M_5$$
$$C_{2,2} = M_1 - M_2 + M_3 + M_6$$

On n'a donc besoin que de 7 multiplications matricielles de matrices carrées de taille $\frac{n}{2}$ et 18 sommes, ce qui apporte un vrai plus en terme de complexité comme nous allons le voir.

Dans l'algorithme naïf de multiplication par blocs, si $T(n)$ représente le nombre d'opérations, on a la relation de récurrence suivante :

$$T(n) = 8 * T\left(\frac{n}{2}\right) + 4\left(\frac{n}{2}\right)^2 .$$

Avec les notations du théorème précédent, on a donc $\alpha = 1, a = 8, b = 2$ et $c = 2$. D'où $c < \log_b(a)$ et donc $T(n) = \mathcal{O}(n^3)$.

Dans le cas de l'algorithme de Strassen, on trouve

$$T(n) = 7T(n) + 18\left(\frac{n}{2}\right)^2$$

soit $\alpha = \frac{9}{2}, a = 7, b = 2$ et $c = 2$. Encore une fois, $c < \log_b a$ et donc $T(n) = \mathcal{O}(n^{\log_2(7)}) = \Theta(n^{2,807})$ ce qui est mieux...

Implementation

On se contentera d'écrire l'algorithme de Strassen pour des matrices carrées dont la taille est une puissance de 2 (ce n'est pas vraiment une limitation car n'importe quelle matrice peut devenir de cette forme en complétant les lignes et les colonnes par des 0.)

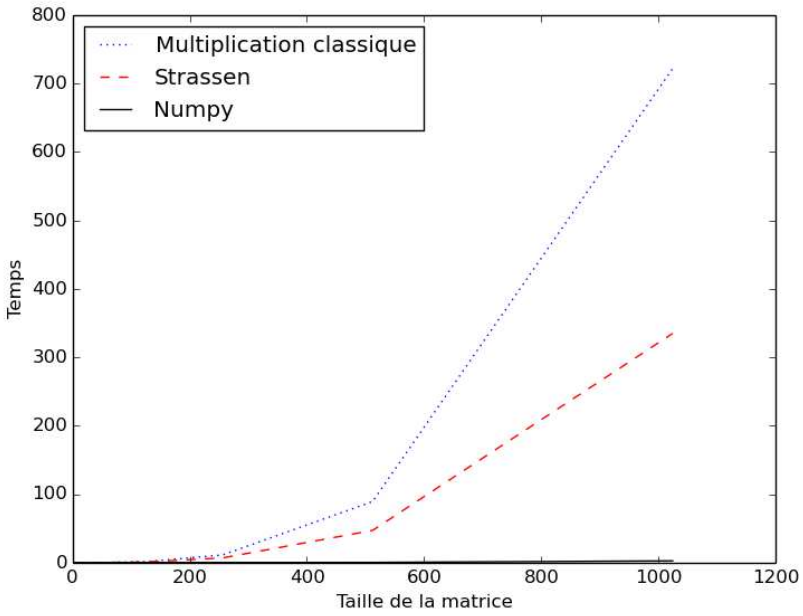
Pour les matrices de « petite » taille, on écrira une procédure de calcul du produit matriciel qui utilise les formules habituelles, car l'algorithme de Strassen fait perdre du temps si n est petit.

Les matrices seront définies par des tableaux de `NumPy`. On rappelle ci-dessous quelques commandes sur ces tableaux.

```
import numpy as np

A = np.random.random( (x,y) )
# renvoie une matrice de dimensions x*y avec des coefficients
# aléatoires entre 0 et 1
A.shape
# renvoie le tuple (x,y) correspondant à la taille de la matrice
A[i,j]
# élément d'indices i et j
# les indices commencent à 0
A[i:]
# renvoie la ième ligne de A
A[:,j]
# renvoie la jeme colonne de A
np.concatenate( (A,B), axis=0)
# concatène des matrices de tailles compatibles
# dans la direction verticale
np.concatenate( (A,B), axis=1)
# concatène des matrices de tailles compatibles
# dans la direction horizontale
np.dot(A,B)
```

```
    # calcule le produit matriciel AB
```

La figure ci-dessous illustre les performances de la méthode comparée à la multiplication matricielle classique, mais on reste bien loin de la fonction implémentée dans `NumPy` !



III.4. Multiplication rapide de polynômes- 2e version

On présente ici une méthode efficace de multiplication de polynômes utilisant la transformée de Fourier.

Déf 3:

Si A est un polynôme de degré n dont les coefficients sont $a_0, a_1, \dots, a_n$ et ω une racine primitive ℓ-ième de l'unité, la transformée de Fourier de A est la donnée des valeurs $(A(\omega^0), A(\omega^1), \dots, A(\omega^{\ell-1}))$.

Pour représenter un polynôme informatiquement, on peut utiliser plusieurs méthodes : soit le polynôme est représenté par ses coefficients sous la forme d'une liste, soit par sa valeur en certains points prédéfinis. Ces deux représentations ont leur avantage. Par exemple la représentation par points/valeurs est très efficace pour le calcul du produit de deux polynômes : si A et B sont deux polynômes de degré n dont les valeurs sont connues en $2n+1$ points, alors on connait les valeurs du polynôme AB en ces $2n+1$ points par simple multiplication dans les réels : il en faut $2n+1$. Les résultats classiques d'interpolation nous permettent de dire que cela est suffisant pour connaître exactement AB.

La représentation points/valeurs n'est cependant pas efficace pour l'évaluation d'un polynôme en d'autres points que les points d'interpolation. Dans ce cas, c'est la représentation par coefficients qui est la plus utilisée, couplée avec la méthode de Hörner pour l'évaluation.

Si l'on veut évaluer le polynôme $A = \sum_{i=0}^{n} a_i X^i$, au point x, on écrit :

$$A(x) = a_0 + x(a_1 + x(a_2 + x(\cdots + x(a_{n-2} + x(a_{n-1} + xa_n))\dots)))$$

Cette méthode nécessite de l'ordre de n additions et n multiplications, ce qui est mieux que la méthode naïve en n^2.

Mais concernant la multiplication, la représentation par coefficients est très peu efficace. Si on pose $C = AB$, les coefficients de A étant les a_i, ceux de B les b_i et ceux de C les c_i, on a $c_i = \sum_{k=0}^{i} a_k b_{i-k}$. On comprend qu'il faut alors de l'ordre de n^2 multiplications et additions : c'est un ordre de grandeur supérieur à la représentation points/valeurs. La représentation par coefficients étant plus répandue, il est légitime de se demander si on ne pourrait pas utiliser la représentation points/valeurs pour la multiplication, en allant chercher par exemple des propriétés d'interpolations.

La bonne idée est d'interpoler sur les racines de l'unité. Dans la suite, on considère que A et B sont deux polynômes de degré $\leqslant n-1$.

Commençons par la phase d'évaluation. Soit ω une racine primitive n^{e} de l'unité, on cherche à calculer $A(\omega^k)$ pour $0 \leqslant k \leqslant n-1$:

$$\begin{aligned} A(\omega^k) &= \sum_{i=0}^{n-1} a_i \omega^{ki} \\ &= \sum_{0 \leqslant 2i \leqslant n-1} a_{2i}\omega^{2ki} + \sum_{0 \leqslant 2i+1 \leqslant n-1} a_{2i+1}\omega^{k(2i+1)} \\ &= \sum_{0 \leqslant 2i \leqslant n-1} a_{2i}\omega^{2ki} + \omega^k \sum_{0 \leqslant 2i+1 \leqslant n-1} a_{2i+1}\omega^{2ki} \\ &= P(\omega^{2k}) + \omega^k I(\omega^{2k})\,. \end{aligned}$$

Ainsi pour calculer l'évaluation sur une racine de l'unité, il suffit de calculer l'évaluation en ω^2 des polynômes de degré $\dfrac{n}{2}$ composés des coefficients d'indice pair (polynôme P) pour l'un et des coefficients d'indice impair (polynôme I) pour l'autre (si ω est une racine n^{e} de l'unité, ω^2 en est une racine $(n/2)^{\text{e}}$). Pour obtenir la transformée de Fourier d'un polynôme A, il faudra faire cette évaluation sur toutes les racines n^{es} de l'unité pour obtenir la liste $[A(\omega^0),\ \ldots,\ A(\omega^{n-1})]$, mais les polynômes P et I restent les mêmes.
On a ainsi divisé le problème de taille n en deux sous problèmes de taille $\dfrac{n}{2}$ qu'il faut traiter tous les deux. Il faudra prévoir n opérations pour séparer les coefficients pairs et impairs et deux opérations à chaque fois (une multiplication et une addition) pour construire la solution. La relation de récurrence vérifiée par le nombre d'opérations est donc de la forme :

$$T(n) = 2T\left(\frac{n}{2}\right) + n + 2n = 2T\left(\frac{n}{2}\right) + 3n$$

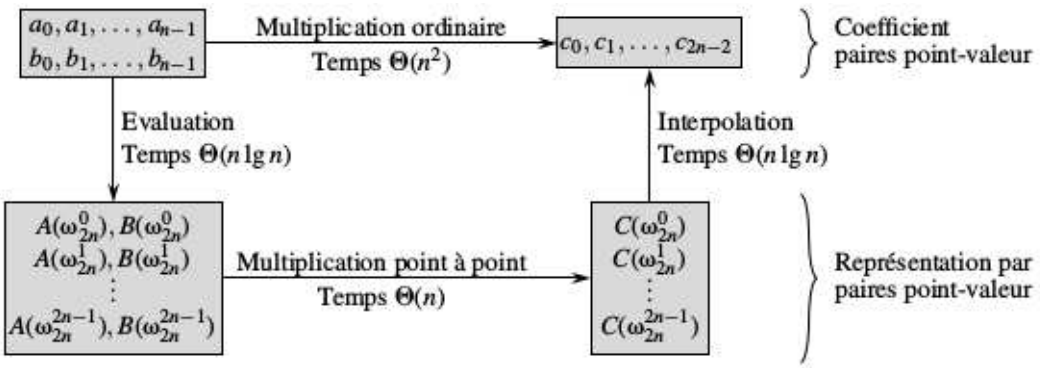
On se trouve dans les conditions du théorème avec $\alpha = 3, a = 2, b = 2$ et $c = 1$. On est donc dans le cas d'égalité et $T(n) = \mathcal{O}(n\log_2(n))$.
On effectue cette opération sur les deux polynômes A et B, puis on fait le produit terme à terme des deux listes pour obtenir la transformée de Fourier du polynôme AB. Cette opération s'effectue en temps linéaire.
Reste à savoir comment retrouver, à partir de sa transformée de Fourier, les coefficients d'un polynôme. Soit P un polynôme de degré $\leqslant n-1$. On note γ_k le k^{e} coefficient de sa transformée : $\gamma_k = P(\omega^k)$ avec $\omega = \exp\left(2\text{i}\dfrac{\pi}{n}\right)$.
On a pour un entier $i \leqslant n-1$ donné :

$$\sum_{k=0}^{n-1} \gamma_k \omega^{-ik} = \sum_{k=0}^{n-1}\sum_{j=0}^{n-1} a_j \omega^{kj}\omega^{-ki} = \sum_{j=0}^{n-1} a_j \sum_{k=0}^{n-1} \omega^{k(j-i)} = \sum_{j=0}^{n-1} a_j \sum_{k=0}^{2n-1} \left(\omega^{(j-i)}\right)^k$$

La deuxième somme est nulle sauf si $j = i$: en effet le fait d'avoir choisi les racines n-ièmes de l'unité assure que $\omega^{j-i} \neq 1$ sauf si $i = j$:

$$\sum_{k=0}^{n-1} \gamma_k \omega^{-ik} = n a_i.$$

Ainsi pour retrouver a_i il suffit de refaire une transformée de Fourier en prenant cette fois ω^{-1} comme élément de base pour l'évaluation. Retrouver les coefficients se fera donc encore avec une complexité de l'ordre de $n\log(n)$.
La stratégie pour multiplier deux polynômes est illustrée par le schéma ci-dessous.



À faire :

1. Écrire une procédure récursive permettant de calculer la transformée de Fourier d'un polynôme de degré $\leqslant n-1$ donné par la liste de ses coefficients. On suppose que n est une puissance de 2.
   Pour cela on pourra utiliser le module `cmath` qui permet de manipuler des nombres complexes.

```python
import cmath as cm

# Un nombre complexe z est représenté par un couple (x,y) de nombres réels
# et est affiché sous la forme x + yj
z = 2+1j
# les parties réelle et imaginaire s'obtiennent par z.real et z.imag

# le module cmath contient entre autres les fonctions suivantes

abs(z)
# calcule le module de z
phase(z)
# calcule un argument de z  entre -pi et pi
polar(z)
# retourne le couple (r, theta)
rect(r,theta)
# retourne le complexe de module r et d'argument theta

# la plupart des fonctions réelles présentes dans le module math
# possèdent leur équivalent complexe dans le module cmath:
# exp, sqrt, pi, ...
```

L'algorithme est expliqué ci-dessous (cette figure ainsi que la précédente sont issues de la « bible » : Algorithmique, de Cormen aux Éditions Dunod (1200 pages !))

```
FFT-RÉCURSIVE(a)
 1  n ← longueur[a]                ▷ n est une puissance de 2.
 2  si n = 1
 3     alors retourner a
 4  ω_n ← e^{2πi/n}
 5  ω ← 1
 6  a^[0] ← (a_0, a_2, ..., a_{n-2})
 7  a^[1] ← (a_1, a_3, ..., a_{n-1})
 8  y^[0] ← FFT-RÉCURSIVE(a^[0])
 9  y^[1] ← FFT-RÉCURSIVE(a^[1])
10  pour k ← 0 à n/2 − 1
11      faire y_k ← y_k^[0] + ω y_k^[1]
12           y_{k+(n/2)} ← y_k^[0] − ω y_k^[1]
13           ω ← ω ω_n
14  retourner y                    ▷ y est supposé être un vecteur colonne.
```

2. Écrire une procédure pour retrouver les coefficients d'un polynôme à partir de sa transformée de Fourier.
3. En déduire une procédure permettant de calculer le produit de deux polynômes A et B donnés par leurs coefficients. On commencera pour cela par chercher la plus grande puissance de 2 supérieure à $\deg A + \deg B$ et on complètera la liste des coefficients par des 0.

III.5. Médiane d'un ensemble

L'algorithme exposé ici permet de calculer le n^{e} élément d'un ensemble de cardinal m (pour une certaine relation d'ordre). Il a une complexité de $\mathcal{O}(m)$. Il s'applique en particulier à la recherche de la médiane d'un ensemble fini (ou de l'élément qui en est le plus proche par excès), en prenant $n = \left\lfloor \frac{m}{2} \right\rfloor + 1$, ce que l'on ne savait faire jusqu'à 1969 qu'en un temps de l'ordre de $m \log m$.

On applique ici aussi le principe « Diviser Pour Régner ».

- On découpe l'ensemble de m éléments en $\frac{m}{k}$ sous-ensembles, chacun de cardinal $\leqslant k$. k est une constant à choisir convenablement, $k \geqslant 3$.
- On cherche les médianes des $\frac{m}{k}$ sous-ensembles, et la médiane de ces médianes, soit x.

– On sépare l'ensemble initial en 2 sous-ensembles : celui des j éléments inférieurs à x et celui des $m-j$ éléments supérieurs à x, les éléments égaux à x étant mis alternativement d'un côté ou de l'autre.
Si j est supérieur à n, on est ramené à chercher le n^{e} élément du 1^{er} sous-ensemble ; s'il est inférieur à n, on cherche le $(n-j)^{\text{e}}$ élément du second.

Lorsque la taille d'un ensemble est inférieure à k, on calculera directement la médiane ou le n^{e} élément par une procédure auxiliaire qui utilise la méthode `sort` de Python.

Complexité :

Notons $T(m)$ le temps nécessaire pour trouver le n^{e} élément d'une liste de cardinal m. On considèrera que si $m \leqslant k$, $T(m) = a = cste$).
On :

$$\begin{aligned} T(m) = \alpha m &\quad \text{(temps de sépartion ne } m/k \text{ sous-ensembles)} \\ + \frac{m}{k}T(k) &\quad \text{(médianes de chacun de ces sous-ensembles)} \\ + T\left(\frac{m}{k}\right) &\quad \text{(temps pour la médiane des médianes)} \\ + \beta m &\quad \text{(temps de séparation en deux sous-ensembles)} \\ + T\big(\min(j, m-j)\big) & \end{aligned}$$

Puisque $T(k) = a$, que $T\big(\min(j, m-j)\big) \leqslant T\left(\frac{m}{2}\right)$ et que $T\left(\frac{m}{k}\right) \leqslant T\left(\frac{m}{3}\right)$ car $k \geqslant 3$ on obtient :

$$T(m) \leqslant \left(\alpha + \beta + \frac{a}{k}\right) m + T\left(\frac{m}{2}\right) + T\left(\frac{m}{3}\right).$$

Montrons alors par récurrence qu'il existe une constante C telle que $T(m) \leqslant Cm$ pour tout m.

– Initialisation : on peut toujours trouver c telle que $T(m) \leqslant cm$ pour les premières valeurs de m, et toute valeur $C \geqslant c$ convient aussi.

– Ensuite :

$$\begin{aligned} T(m) &\leqslant \left(\alpha + \beta + \frac{a}{k}\right) m + C\frac{m}{k} + C\frac{m}{2} \\ &\leqslant \left(\alpha + \beta + \frac{a}{k} + \frac{C}{k} + \frac{C}{2}\right) m \\ &\leqslant \left(\alpha + \beta + \frac{a}{k} + \frac{5C}{6}\right) m \end{aligned}$$

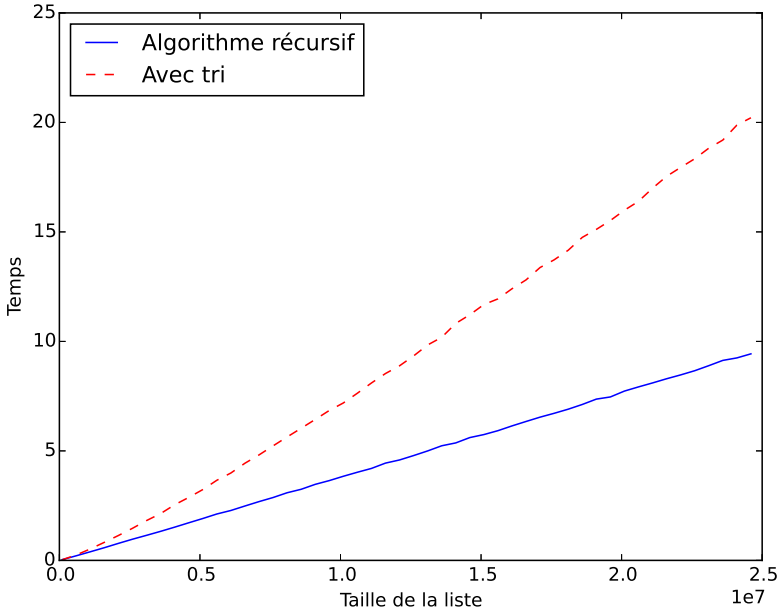
et il suffit donc de prendre $C \geqslant 6\left(\alpha + \beta + \frac{a}{k}\right)$ pour obtenir $T(m) \leqslant Cm$.

La complexité de l'algorithme est donc bien un $\mathcal{O}(m)$.

Comme nous n'aurons certainement pas le temps de faire tous les programmes de ce chapitre, je vous donne la solution en Python ; j'y ai inclus un graphique pour comparer le temps d'exécution avec celui obtenu à l'aide d'un simple programme de tri (qui est en $\mathcal{O}(m \log m)$ au minimum, comme on le verra dans un prochain chapitre).

```
import random
from time import clock
import matplotlib.pyplot as plt

def nieme_element1(liste,n):
    # n-ième élément d'une liste en utilisant le tri
    liste.sort()
    return liste[n-1]

def nieme_element(liste, n):
    k = 5
    m = len(liste)
    if m <= k:
        return nieme_element1(liste, n)
    else:
        liste_medianes = []
        q, r = divmod(m, k)
        for i in range(0, q, k):
            li = liste[i:i+k]
```

```
            liste_medianes.append(nieme_element1(li, k//2 + 1))
        if r != 0:
            li = liste[q*k:]
            liste_medianes.append(nieme_element1(li, r//2 + 1))
        x = nieme_element(liste_medianes, len(liste_medianes)//2 + 1)
        liste1 = []
        liste2 = []
        alternative = False
        for i in range(0,m):
            elt = liste[i]
            if elt < x:
                liste1.append(elt)
            elif elt > x:
                liste2.append(elt)
            else:
                if alternative:
                    liste1.append(elt)
                else:
                    liste2.append(elt)
                alternative = not alternative
        j = len(liste1)
        if j >= n:
            return nieme_element(liste1, n)
        else:
            return nieme_element(liste2, n-j)

nb_elements = 10000000
nmax = 10000000
liste_complete = [random.randint(0, nmax) for i in range(nb_elements)]
N = []
T1 = []
T2 = []
for n in range(100000, nb_elements, 200000):
    # print(n, sep=' ') juste pour suivre la progression
    liste = liste_complete[0:n]
    debut = clock()
    x = nieme_element(liste, n//2 + 1)
    T1.append(clock() - debut)
    debut = clock()
    y = nieme_element1(liste, n//2 + 1)
    T2.append(clock() - debut)
    if x != y:
        print('oups! programme faux!')
    N.append(n)

plt.plot(N, T1, color='blue', label='Algorithme récursif')
plt.plot(N, T2, color='red', linestyle='--', label='Avec tri')
plt.legend(loc = 'upper left')
plt.xlabel('Taille de la liste')
plt.ylabel('Temps')

plt.show()
```


Algorithme récursif
Avec tri
Temps
Taille de la liste
1e7
0
5
10
15
20
25
0.0
0.5
1.0
1.5
2.0
2.5