INFORMATIQUE 2ème année

PLAN

I : Récursivité

1- Définitions récursives

Un objet d'un type donné est défini récursivement lorsque la définion fait appel à elle-même. Plus précisément, cette définition est constituée de deux parties :

la définition de base permettant d'initialiser un objet du type donné

la définition proprement récursive permettant de définir un objet du type donné à partir d'un autre objet du même type.

Cette notion est directement liée à la notion de définition par récurrence en mathématiques.

EXEMPLE 1 :

Les termes de la suite de Fibonacci sont définis récursivement par :

$F(0) = 1$	définition de base
$F(1) = 1$	deuxième définition de base
si $n \geq 2$, $F(n) = F(n-1) + F(n-2)$	définition récursive

EXEMPLE 2 :

De même pour la suite de Thue définie par :

$T(1) = 0$ définition de base

$$\begin{cases} \text{si } n \text{ pair} \geq 2,\ T(n) = 1 - T(\frac{n}{2}) \\ \text{si } n \text{ est impair} \geq 3,\ T(n) = T(\frac{n+1}{2}) \end{cases}$$ définition récurvise

On remarque que, si n est pair, $\frac{n}{2} < n$ et si n est impair supérieur ou égal à 3, $\frac{n+1}{2} < n$ de sorte que, dans chaque cas, la définition de T(n) fait appel à une valeur de T appliquée à un entier strictement plus petit. Il est alors facile de prouver par récurrence que la valeur de T(n) est définie de manière unique. L'exemple suivant est plus délicat.

EXEMPLE 3 :
La longueur de la suite de Collatz (cf cours de première année ALGO1.PDF) est définie par :

$C(1) = 0$ définition de base

$$\begin{cases} \text{si } n \text{ est pair} \geq 2,\ C(n) = 1 + C(\frac{n}{2}) \\ \text{si } n \text{ est impair} \geq 3,\ C(n) = 1 + C(3n+1) \end{cases}$$ définition récursive

Contrairement à l'exemple précédent, dans le cas n impair, la définition de C(n) fait appel à la valeur de C appliquée à $3n + 1$ qui est supérieur à n. Dans l'état actuel des connaissances mathématiques, on ignore si cette définition permet d'attribuer une valeur explicite à tout C(n). Cet exemple pose la question de la terminaison des définitions récursives. On prouve qu'il n'existe aucun procédé universel permettant de décider si une définition récursive s'arrête ou non. Ce problème est analogue au problème de terminaison des boucles tant que ... faire ...

En général, on prouve qu'une définition récursive se termine en mettant en évidence une fonction φ définie sur l'ensemble des objets définis récursivement, à valeurs entières, et qui vérifie les propriétés suivantes :

si x est un objet de base, alors $\varphi(x) = 0$
si x est un objet défini récursivement à partir des objets y, z, etc..., alors :
$\varphi(y) < \varphi(x)$, $\varphi(z) < \varphi(x)$, etc...

On peut alors montrer par récurrence sur la valeur $n = \varphi(x)$ que l'objet x est défini en un nombre fini d'étapes. Pour la suite de Fibonacci, il suffit de définir :

$\varphi(F(0)) = \varphi(F(1)) = 0$
$\forall\ n \geq 2,\ \varphi(F(n)) = n$

Pour la suite de Thue, on peut prendre une définition analogue. Mais pour la suite de Collatz, on ignore s'il est possible de mettre en évidence une telle fonction φ.

EXEMPLE 4 :
Les listes en informatique sont typiquement définies de façon récursive. Une liste est :

ou bien vide définition de base
ou bien constituée d'une liste suivie d'un élément définition récursive

EXEMPLE 5 :
Les courbes du dragon sont définies de façon itérative de la façon suivante. On considère une feuille de papier que l'on plie plusieurs fois :

une fois un pli

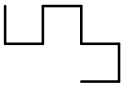
deux fois  trois plis

trois fois  sept plis

Lorsque l'on déplie les plis de la feuille à angle droit, on obtient, en laissant la partie inférieure de la feuille immobile :

une fois  G

deux fois  GGD

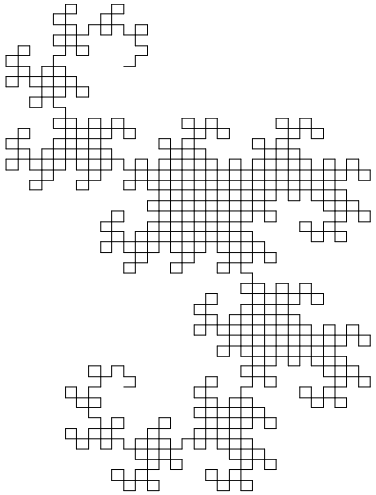
trois fois  GGDGGDD

Les lettres G ou D indique dans quelle direction le pli se fait (Gauche ou Droite) lorsque l'on part de le partie inférieure de la feuille. Les courbes obtenues s'appellent courbes du dragon. Si le procédé précédent est itératif, il permet de définir très simplement la suite S de lettres G ou D (ou mot) définissant une courbe du dragon de façon récursive. Notons S* le mot S à l'envers, et en intervertissant les lettres G et D. Par exemple, si S = GGD, alors S* = GDD. On a alors la définition récursive suivante des mots S formant une suite du dragon :

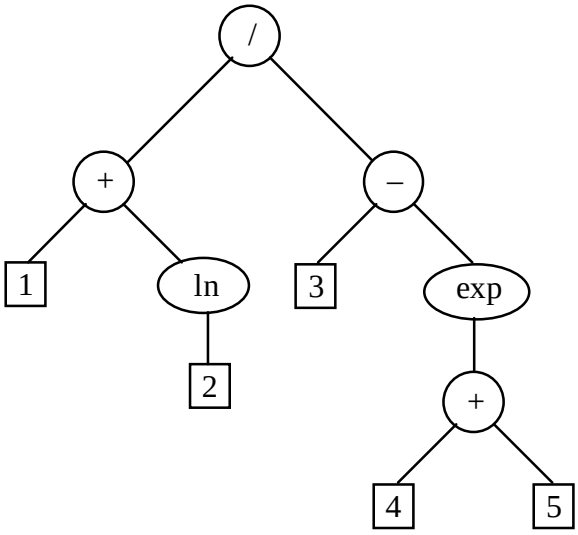
ou bien S = G	définition de base
ou bien S = $\sum G \sum^*$ où $\sum$ est une suite du dragon	définition récursive

Ainsi, les suites suivantes forment des suites du dragon (on a indiqué en bleu le G central) :

S_1 = G
S_2 = GGD
S_3 = GGDGGDD
S_4 = GGDGGDDGGGDDGDD
S_5 = GGDGGDDGGGDDGDDGGGDGGDDDGGDDGDD
S_6 = GGDGGDDGGGDDGDDGGGDGGDDDGGDDGDDG
GGDGGDDGGGDDGDDDGGDGGDDDGGDDGDD
etc ...



EXEMPLE 6 :
La notation postfixée.
Le calcul $\frac{1 + \ln(2)}{3 - \exp(4 + 5)}$ peut être représenté par une structure arborescente indiquant l'ordre des opérations à mener :



On peut également définir cette suite d'opérations à mener par une liste contenant les nombres suivis des fonctions ou opérateurs binaires appliqués sur ces nombres. On parle de notation postixée. Cette notation permet d'éviter l'usage de parenthèses :

[1, 2, ln, +, 3, 4, 5, +, exp, –, /]

De nombreux compilateurs (chargés de traduire les instructions de l'utilisateurs en commandes exécutables par la machine) traduisent une chaîne de caractères décrivant un calcul tapée par l'utilisateur (telle que "(1 + ln(2))/(3 - exp(4 + 5))" en un liste postfixée. C'est ensuite lors de l'exécution proprement dite du programme que cette liste est numériquement évaluée.

Comment définir une liste postfixée syntaxiquement correcte ? Les éléments de cette liste sont constitués de trois types d'objets, les valeurs numériques (1, 2, 3, ...), les fonctions (exp, ln), les opérateurs (+, – , ...). On dispose des trois règles de construction suivantes. La première est la définition de base, les deux autres sont récursives :

1) Une liste postfixée de base est constituée d'une unique valeur numérique [*val*]. Le résultat du calcul portant sur cette liste est simplement la valeur *val*.

2) Si on dispose d'une liste postfixée [*liste*], alors [*liste*, *func*], où *func* est une fonction, est une liste postixée. Le résultat du calcul portant sur cette liste est *func*(*val*).

3) Si on dispose de deux listes postfixées [*liste*1] et [*liste*2], alors [*liste*1, *liste*2, *oper*], où *oper* désigne un opérateur binaire, est une liste postixée. Le résultat du calcul portant sur cette liste est *oper*(*liste*1, *liste*2).

EXEMPLE 7 :

Les déterminants peuvent être définis définis de manière récursive. Soit A une matrice $n \times n$ de terme général a_{ij}. Alors :

si $n = 1$ alors $\det(A) = a_{11}$ définition de base

sinon $\det(A) = \sum_{i=1}^{n} (-1)^{i+1} a_{i1} \det((a_{kl}), k \neq i, l \neq 1)$

Ce n'est rien d'autre qu'une définition du déterminant par développement par rapport à la première colonne. Elle est cependant numériquement inefficace, sa mise en oeuvre donnant un temps d'exécution en O(n!).

EXEMPLE 8 :

Pour définir une permutation d'un ensemble E = {1, ..., *n*}, on peut procéder comme suit :

Si E est un singleton {*x*}, la seule permution est Id définition de base

Sinon, placer en tête un élément *x* de E et permuter E – {*x*} définition récursive

2- Programmation récursive

Les langages de programmation évolués permettent de définir des fonctions ou des procédures récursives. Leur définition utilise un appel à elles-mêmes. Les algorithmes récursifs se distinguent des algorithmes itératifs, ces derniers faisant appel à des instructions de boucles for ou while. Nous donnerons des exemples de conversion de programme récursif en programme itératif dans la partie II. Les exemples sont donnés en Python.

EXEMPLE 1 :

La suite de Fibonacci.

```
def F(n):
    if n<=1:
        return(1)
    else:
        return(F(n-1) + F(n-2))
```

On voit qu'il suffit de traduire la définition récursive. Malheureusement, dans le cas présent, le temps de calcul devient très important dès que *n* vaut quelques dizaines (comparer par exemple le temps de calcul de F(20), F(30), F(40)...) et nous verrons pourquoi dans un prochain paragraphe.

EXEMPLE 2 :

La suite de Thue : contrairement à la suite de Fibonacci, l'utilisation de la récursivité est ici particulièrement efficace.

```
def T(n):
    if n==1:
        return(0)
    else:
        if n%2==0:
            return(1-T(n//2))
        else:
            return(T((n+1)//2))
```

Voici les premières valeurs de la suite :

```
[T(n) for n in range(1,41)]
```

[0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0]

EXEMPLE 3 :

La longueur de la suite de Collatz :

```
def C(n):
    if n==1:
        return(0)
    else:
        if n%2==0:
            return(1+C(n//2))
        else:
            return(1+C(3*n+1))
```

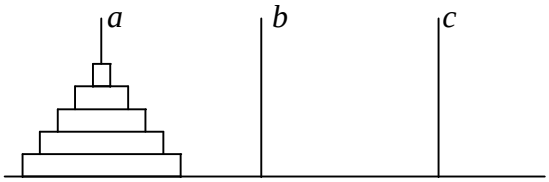
Voici les premières valeurs de la suite :

```
[C(n) for n in range(1,41)]
```

[0, 1, 7, 2, 5, 8, 16, 3, 19, 6, 14, 9, 9, 17, 17, 4, 12, 20, 20, 7, 7, 15, 15, 10, 23, 10, 111, 18, 18, 18, 106, 5, 26, 13, 13, 21, 21, 21, 34, 8]

EXEMPLE 4 :

Le problème des tours de Hanoï consiste à déplacer *n* disques empilés par ordre décroissant sur une pile, en direction d'une autre pile. On peut se servir pour cela d'une troisième pile, mais la règle est de ne pas empiler un disque sur un disque de taille inférieure.



La programmation récursive de ce problème est très simple :

si *n* = 1, déplacer le disque *a* vers le disque *b*
sinon:
déplacer les *n*–1 premiers disques de *a* vers *c* (appel récursif)
déplacer le dernier disque de *a* vers *b*
déplacer les *n*–1 disques de *c* vers *a* (appel récursif)

ce qui donne en Python :

```
def Hanoi(n,a,b,c):
    if n == 1:
        print("deplacer un disque de " + a + " vers " + b)
    else:
        Hanoi(n-1,a,c,b)
        Hanoi(1,a,b,c)
        Hanoi(n-1,c,b,a)
```

Voici un exemple d'exécution de l'algorithme précédent :

```
Hanoi(4,"A","B","C")
```
deplacer un disque de A vers C
deplacer un disque de A vers B
deplacer un disque de C vers B
deplacer un disque de A vers C
deplacer un disque de B vers A
deplacer un disque de B vers C
deplacer un disque de A vers C
deplacer un disque de A vers B
deplacer un disque de C vers B
deplacer un disque de C vers A
deplacer un disque de B vers A
deplacer un disque de C vers B
deplacer un disque de A vers C
deplacer un disque de A vers B
deplacer un disque de C vers B

EXEMPLE 5 :

Soient n et p deux entiers positifs ou nuls et soit A_{np} **le nombre de chemins** allant de (0,0) à (n,p) en utilisant uniquement les déplacements selon les vecteurs (1,0), (0,1) ou (2,1). Les A_{np} peuvent être définis récursivement de la façon suivante :

$\forall\, n,\ A_{n0} = 1$ définition de base (on ne peut utiliser que le vecteur (1,0) n fois)

$\forall\, p,\ A_{0p} = 1$ définition de base (on ne peut utiliser que le vecteur (0,1) p fois)

$\forall\, n,\ A_{1p} = p + 1$ définition de base (on utilise une fois le vecteur (1,0) et p fois le vecteur (0,1) et il y a $p + 1$ choix possibles de positionner le vecteur (1,0) parmi ces $p + 1$ vecteurs)

$\forall\, n \geq 2,\ \forall\, p \geq 1,\ A_{np} = A_{n,p-1} + A_{n-2,p-1} + A_{n-1,p}$ définition récursive (l'établir en envisageant les trois cas possibles pour le dernier déplacement)

La suite A_{nn} constitue la suite A006139 de l'encyclopédie des suites d'entiers. Le lecteur est invité à prouver que :

$$\pi = \sum_{n=0}^{\infty} \frac{(-1)^n\, 2^{2n+3}}{(n+1)\, A_{nn}\, A_{n+1,n+1}} = 4 - 1 + \frac{1}{6} - \frac{1}{34} + \frac{16}{3145} - \frac{4}{4551} + \frac{1}{6601} - \frac{1}{38341} + \ldots$$

(la marge est trop petite pour que la démonstration puisse y tenir).

Voici un algorithme récursif calculant A_{np} (comme pour la suite de Fibonacci, il est simple à mettre en oeuvre, mais peu efficace) sous forme d'un nombre flottant.

```
def A(n,p):
    if n==0:
        return(1.0)
    elif p==0:
        return(1.0)
    elif n==1:
        return(p+1)
    else:
        return(A(n,p-1)+A(n-2,p-1)+A(n-1,p))
```

On peut ensuite définir une fonction S de paramètre N calculant $\sum_{n=\mathrm{N}}^{\infty} \frac{(-1)^n\, 2^{2n+3}}{(n+1)\, A_{nn}\, A_{n+1,n+1}}$:

```
def S(N):
    s = 0.0
    for n in range(N+1):
        s = s+(-1)**n*2**(2*n+3)/((n+1)*A(n,n)*A(n+1,n+1))
```

Voici les premières valeurs de S :

```
for i in range(10):
    print(S(i))
```

```
4.0
3.0
3.16666666667
3.13725490196
3.14234234234
3.14146341463
3.14161490683
3.14158882509
```

3.14159331188
3.14159254045

EXEMPLE 6 :
On définit la **fonction** *f* de $\mathbb{N}$ dans $\mathbb{N}^*$ de la façon suivante :

$f(0) = 1$

et $\forall\, n,\ f(2n+1) = f(n),\ f(2n+2) = f(n) + f(n+1)$

de sorte que les valeurs de *f* sont, pour $n \geq 1$:

1, 2, 1, 3, 2, 3, 1, 4, 3, 5, 2, 5, 3, 4, 1, 5, 4, 7, 3, 8 ...
↑↑ ↑↑
n *n*+1 2*n*+1 2*n*+2

Les valeurs successives de $\frac{f(n)}{f(n+1)}$ sont :

$\frac{1}{2}, 2, \frac{1}{3}, \frac{3}{2}, \frac{2}{3}, 3, \frac{1}{4}, \frac{4}{3}, \frac{3}{5}, \frac{5}{2}, \frac{2}{5}, \frac{5}{3}, \frac{3}{4}, 4, \frac{1}{5}, \frac{5}{4}, \frac{4}{7}, \frac{7}{3}, \frac{3}{8}$ ...

On montre que tous les rationnels positifs apparaissent une fois et une seule dans cette liste, de sorte que l'application $n \rightarrow \frac{f(n)}{f(n+1)}$ forme une bijection de $\mathbb{N}$ dans $\mathbb{Q}^{+}*$. On demande d'écrire une procédure de paramètre *n* permettant de calculer *f*(*n*).

```
def f(n):
    if  n==0:
        return(1)
    elif n%2==1:
        return(f((n-1)//2))
    else:
        return(f(n//2-1)+f(n//2))

[f(n) for n in range(1,20)]
```

[1, 2, 1, 3, 2, 3, 1, 4, 3, 5, 2, 5, 3, 4, 1, 5, 4, 7, 3]

EXEMPLE 7 :
Programmons un algorithme pour dresser la liste de toutes les **permutations d'un ensemble E**. E est représenté par une liste de mots d'une lettre. Le résultat de la procédure est une liste dont les éléments sont les permutations en question. On procède récursivement de la façon suivante : pour chaque élément *x* de E, on crée la liste des permutations des éléments de E autres que *x*, et on ajoute *x* en tête de chacune de ces permutations. Les permutations sont représentées sous forme de chaînes de caractères :

```
def perm(E):
    n = len(E)
    if n==1:
        return([E[0]])
    else:
        M = [ ]
        # M sera la liste des permutations de E
        for i in range(n):
            # copie de E dans C
            C = E[:]
            # suppression de l'élément d'indice i
            del C[i:i+1]
```

```
            # création des permutations de C
            L = perm(C)
            # ajout de E[i] en début de chaque élément de L
            for mot in L:
                M.append(E[i]+mot)
        return(M)
```

Voici un exemple d'exécution :

```
S=perm(['a','b','c','d'])
S
```

['abcd', 'abdc', 'acbd', 'acdb', 'adbc', 'adcb', 'bacd', 'badc', 'bcad', 'bcda', 'bdac', 'bdca',
'cabd', 'cadb', 'cbad', 'cbda', 'cdab', 'cdba', 'dabc', 'dacb', 'dbac', 'dbca', 'dcab', 'dcba']

EXEMPLE 8 :
Les **tableaux de Young** sont des structures intervenant dans plusieurs problèmes d'algèbre ou de combinatoire. Ils sont constitués de lignes successives ayant un nombre décroissant d'éléments. On numérote les cases du tableau par ordre croissant. Voici les sept tableaux de Young constitués de $n = 5$ éléments, chaque ligne étant représentée par la liste des éléments qu'elle contient :

[1]
[2]
[3]
[4]
[5]

[1, 2]
[3]
[4]
[5]

[1, 2]
[3, 4]
[5]

[1, 2, 3]
[4]
[5]

[1, 2, 3]
[4, 5]

[1, 2, 3, 4]
[5]

[1, 2, 3, 4, 5]

Chaque ligne étant une liste, un tableau de Young peut être représenté par la liste de ses lignes, donc par une liste de listes :
[[1], [2], [3], [4], [5]]

[[1, 2], [3], [4], [5]]

[[1, 2], [3, 4], [5]]

[[1, 2, 3], [4], [5]]

[[1, 2, 3], [4, 5]]

[[1, 2, 3, 4], [5]]

[[1, 2, 3, 4, 5]]

Pour un nombre d'éléments *n* donnés, l'ensemble de tous les tableaux de Young peut être représentés par la liste des tableaux. On obtient une liste de listes de listes :

[[[1], [2], [3], [4], [5]] , [[1, 2], [3], [4], [5]] , [[1, 2], [3, 4], [5]] , [[1, 2, 3], [4], [5]] , [[1, 2, 3], [4, 5]] , [[1, 2, 3, 4], [5]] , [[1, 2, 3, 4, 5]]]

On souhaite écrire une procédure Young de paramètre *n* créant cette liste. On voit que, une fois la première ligne d'un tableau créée, de longueur lenLigne1, il s'agit de la compléter par un tableau de Young dont la longueur des lignes ne dépasse pas la longueur de la première ligne et dont les éléments sont numérotés de lnLigne1 + 1 à *n*. Il convient donc d'écrire une procédure intermédiaire calcYoung ayant trois paramètres :

ideb : indice de début de numérotation inclus

ifin : indice de fin de numérotation inclus

lmax : longueur des lignes à ne pas dépasser.

Si ideb = ifin, le tableau de Young n'a qu'un seul élément et vaut [ideb]. Sinon, on crée sa première ligne de longueur lenLigne1 variant de 1 au minimum de lmax et de ifin – ideb + 1. Pour chacune de ces premières lignes, on crée récursivement la liste des tableaux de Young dont les indices varient entre ideb + lenLigne1 jusque ifin, et dont la longueur ne dépasse pas lenLigne1. On ajoute chacun des tableaux de Young ainsi créé à la fin de la première ligne créée (la concaténation des listes s'obtenant par un simple +), formant ainsi à chaque fois un nouveau tableau de Young qu'on stocke au fur à mesure dans une liste ListeN au moyen d'une commande append. Si l'appel récursif renvoie une liste vide de tableaux de Young, on se borne à stocker la seule première ligne comme nouveau tableau de Young :

```
def calcYoung(ideb,ifin,lmax):
    if ideb==ifin:
        return([[[ideb]]])
    else:
        ListeN=[ ]
        for lenLigne1 in range(1,min([ifin-ideb+2,lmax+1])):
            Ligne1=range(ideb,ideb+lenLigne1)
            ListeTYR=calcYoung(ideb+lenLigne1,ifin,lenLigne1)
            lenLTYR=len(ListeTYR)
            if lenLTYR==0:
                ListeN.append([Ligne1])
            else:
                for i in range(lenLTYR):
                    TYN=[Ligne1]+ListeTYR[i]
                    ListeN.append(TYN)
    return(ListeN)
```

Pour créer la liste des tableaux de Young d'une longueur *n* donnée, il suffit d'appeler la fonction précédente avec les paramètre 1, *n* et *n* :

```
def Young(n):
    return(calcYoung(1,n,n))
```

Si LTY est une telle liste de tableaux de Young, on pourra afficher les tableaux au moyen de la procédure suivante :

```
def afficheYoung(LTY):
    n=len(LTY)
    for i in range(n):
        m=len(LTY[i])
        for j in range(m):
            print(LTY[i][j])
        print(' ')
```

EXEMPLE 9 :
Il existe des cas de récursivité indirecte, où une fonction *f* appelle une fonction *g*, qui appelle une fonction *h*, qui appelle la fonction *f*.

EXEMPLE 10 :
On considère deux listes A et B de nombres, triées par ordre croissant. On souhaite fusionner ces deux listes en une seule, également triée par ordre croissant. On suppose qu'on ne peut accéder aux deux listes que par leur dernier élément (ce sont des *piles* de nombre qu'on peut dépiler ou empiler. Voir plus bas la notion de piles). On peut procéder récursivement comme suit : l'appel récursif fusionne les deux piles privées de leur plus grand élément, qu'on ajoute après. Attention, cette fonction modifie les contenus des piles A et B. Si on souhaite les conserver, il convient d'en faire une copie auparavant.

```
def Fusion(A,B):
    if A==[]:
        return(B)
    elif B==[]:
        return(A)
    else:
        # retire le dernier élément de A et le stocke dans a
        a=A.pop()
        # idem pour b
        b=B.pop()
        if a<b:
            A.append(a)
            C=Fusion(A,B)
            C.append(b)
            return(C)
        else:
            B.append(b)
            C=Fusion(A,B)
            C.append(a)
            return(C)
```

Si on suppose que A et B sont des listes qu'on peut parcourir depuis leur premier élément, on peut procéder itérativement. On stocke dans une liste C la fusion des deux listes A et B :

```
def Fusion(A,B):
    C=[]
    i=0            # indice de parcours de A
    j=0            # indice de parcours de B
    lA=len(A)      # longueur de A
    lB=len(B)      # longueur de B
    while (i<lA) and (j<lB):
        if A[i]<B[j]:
            C.append(A[i])
            i=i+1
        else:
            C.append(B[j])
            j=j+1
    # L'une des deux listes a été entièrement parcourue.
    # On complète alors C avec les éléments restants dans la liste
non vide
    while (i<lA):
        C.append(A[i])
        i=i+1
    while (j<len(B)):
        C.append(B[j])
        j=j+1
    return(C)
```

3- Preuves récursives

On prouve qu'une propriété est vérifiée pour tout objet défini récursivement :

en vérifiant cette propriété pour le cas de base

en prouvant que, si elle est vraie pour les composantes d'un objet défini récursivement, elle est vraie pour l'objet lui-même.

La démarche ci-dessus suffit car il est alors facile de montrer qu'un objet vérifie la propriété par récurrence sur le nombre de fois où la définition a été utilisée pour définir cet objet.

EXEMPLE 1 :

Prouver que, dans les suites du dragon, le nombre de lettres est de la forme $2^n - 1$

Cette propriété est vraie pour la définition de base avec $n = 1$.

Supposons qu'une suite du dragon $\sum$ contienne $2^n - 1$ lettres. Alors la suite du dragon $S = \sum G \sum^*$ contiendra $2^n - 1 + 1 + 2^n - 1 = 2^{n+1} - 1$ lettres et la propriété est vraie pour S.

La propriété est alors vraie pour toute suite du dragon.

EXEMPLE 2 :

Prouvons que, dans une liste de calcul en notation postfixée, le nombre d'opérateurs binaires est égal au nombre de valeurs numériques diminué de 1.

- La propriété est trivialement vérifiée pour le cas de base, où [*liste*] = [*val*]
- Si elle est vraie pour [*liste*], elle est vraie pour [*liste*, *func*] puisque ni le nombre de valeurs numériques ni le nombre d'opérateurs binaires.
- Si elle est vraie pour deux listes [*liste*1, *liste*2], alors elle est vraie pour [*liste*1, *liste*2, *oper*]. Car si *liste*1 possède n_1 valeurs numériques (et donc $n_1 - 1$ opérateurs) et *liste*2 en possède n_2 (et donc $n_2 - 1$ opérateurs), alors [*liste*1, *liste*2, *oper*] possède $n_1 + n_2$ valeurs numériques et $(n_1 - 1) + (n_2 - 1) + 1 = n_1 + n_2 - 1$ opérateurs. La propriété est donc vérifiée pour [*liste*1, *liste*2, *oper*].

La propriété ayant été vérifiée sur le cas de base et son héridité vérifiée sur chaque définition récursive, elle est vraie pour toute liste ainsi définie.

On peut aussi procéder à des récurrences classiques :

EXEMPLE 3 :
Montrer que le nombre de déplacements de n disques dans le problème des tours de Hanoï est $2^n - 1$.
Cette propriété est vraie si $n = 1$ (il y a un seul déplacement).
Supposons qu'elle soit vraie au rang $n - 1$. Alors le nombre de déplacements de n disques est (en suivant l'algorithme et en appliquant deux fois l'hypothèse de récurrence pour déplacer les tas de $n - 1$ disques : $(2^{n-1} - 1) + 1 + (2^{n-1} - 1) = 2^n - 1$.
La propriété a ainsi été prouvée par récurrence.

II : Piles

1- Définition

Une pile L est une liste dotée des deux méthodes suivantes :

`L.push(x)`	qui ajoute l'élément *x* en queue de liste
`x=L.pop()`	qui retire le dernier élément de la liste et le stocke dans une variable *x*

La méthode pop appliquée à une liste vide entraîne généralement une erreur d'exécution. Il est donc recommandé de tester la longueur de L (appelée hauteur de la pile) avant d'exécuter pop, à moins d'être sûr que la pile ne contienne un élément.

Les piles servent à stocker des objets à traiter ultérieurement, le dernier dossier empilé étant le premier traité (*Last in, first out* ou LIFO disent les anglais).

En Python, une pile peut être simulée par une liste. La méthode pop existe. Quand à push, on l'obtient par la commande :

```
L.append(x)
```

EXEMPLE
Voici comment une pile sert à traiter les listes de calcul postfixées. Les éléments de la liste sont lus de gauche à droite.

Si l'élément est une valeur *val*, celui-ci est empilé

Si l'élement est une fonction *func*, on dépile la pile, on applique *func* à l'élément dépilé, et on empile le résultat.

Si l'élément est un opérateur binaire *oper*, on dépile les deux derniers éléments de la pile, on leur applique *oper*, et on empile le résultat.
A la fin de la lecture, le résultat est l'élément unique de la pile.

Considérons par exemple la liste [1, 2, ln, +, 3, 4, 5, +, exp, –, /]. Voici l'évolution du calcul (les éléments de la pile sont empilés ou dépilés par la droite)

Initialement	pile vide [ ]
On lit 1	pile = [1]
On lit 2	pile = [1, 2]
On lit ln	pile = [1]

	calcul de ln(2) = 0.6931471805599 pile = [1, 0.6931471805599]
On lit +	pile = [ ] calcul de 1 + 0.6931471805599 = 1.6931471805599 pile = [1.6931471805599]
On lit 3	pile = [1.6931471805599, 3]
On lit 4	pile = [1.6931471805599, 3, 4]
On lit 5	pile = [1.6931471805599, 3, 4, 5]
On lit +	pile = [1.6931471805599, 3] calcul de 4 + 5 = 9 pile = [1.6931471805599, 3, 9]
On lit exp	pile = [1.6931471805599, 3] calcul de exp(9) = 8103.083927576 pile = [1.6931471805599, 3, 8103.083927576]
On lit –	pile = [1.6931471805599] calcul de 3 – 8103.083927576 = – 8100.083927576 pile = [1.6931471805599, – 81003083927576]
On lit /	pile = [ ] calcul de 1.6931471805599/(– 81003083927576) pile = [– 0.0002090283502861]

Les piles jouent également un rôle crucial dans la récursivité.

2- Appels de procédure

Une pile est utilisée lors de l'exécution d'un programme, afin de gérer les appels de procédures ou de fonctions. Supposons que l'on soit en train d'exécuter un programme et qu'on tombe sur une instruction du type :

`y ← f(x)`

où *x* est une variable préalablement assignée et *f* une fonction. La machine doit déterminer la valeur de *f*(*x*) avant de l'affecter à *y*. Pour cela, elle empile l'état de la machine (registres du processeur, valeurs des variables locales, adresse de retour de l'instruction `y ← f(x)` qu'elle devra retrouver à l'issue du calcul de *f*(*x*)), puis on affecte à l'adresse de la prochaine instruction à exécuter celle où débute le calcul de *f*, la valeur du paramètre *x* lui est transmise et l'exécution de *f* commence. Lorsqu'on arrive à la fin de la procédure calculant *f*(*x*), on dépile la pile pour revenir à l'état antérieur, et le programme principal se poursuit en affectant la valeur de *f*(*x*) à *y*. Bien entendu, si l'exécution du calcul de *f*(*x*) fait lui-même appel à une fonction *g*, un deuxième empilement de données à lieu au moment de l'appel de *g*. Dans le cas d'une fonction définie récursivement, les empilements vont se succéder jusqu'à ce qu'on atteigne une définition de base de la fonction *f*.

EXEMPLE 1 :
Considérons la suite de Thue définie par le programme récursif suivant :

```
def T(n):
    if  n==1:
        return(0)
    else:
        if n%2==0:
            return(1-T(n//2))
        else:
            return(T((n+1)//2))
```

Voici les empilements (gérés automatiquement par la machine) qui ont lieu lors de la demande de calcul de T(25). Chaque décalage de l'indentation vers la droite représente un empilement, chaque décalage vers la gauche un dépilement :

25 est impair
appel de T(13)
13 est impair
appel de T(7)
7 est impair
appel de T(4)
4 est pair
appel de T(2)
2 est pair
appel de T(1)
1 est valeur de base de T
renvoi de 0 comme valeur de T(1)
renvoi de 1 – T(1) = 1 comme résultat de T(2)
renvoi de 1 – T(2) = 1 comme résultat de T(4)
renvoi de T(4) = 1 comme résultat de T(7)
renvoi de T(7) = 1 comme résultat de T(7)
renvoi de T(13) = 1 comme résultat de T(25)
le résultat est 1

On voit que l'exécution, transparente pour l'utilisateur, est assez complexe. Elle est cependant efficace dans le cas présent. Montrons que, si p est l'entier tel que $2^{p-1} < n \leq 2^p$, alors le nombre d'appels récursifs pour le calcul de T(n) est p. Cette propriété est vraie pour $n = 1 = 2^0$ et pour lequel la valeur de T(1) est donnée immédiatement. Soit $n > 1$ et supposons la propriété vraie pour tout entier strictement inférieur à n.

Si n est pair, le calcul de T(n) fait appel à T($\frac{n}{2}$) or $2^{p-2} < \frac{n}{2} \leq 2^{p-1}$, donc, d'après l'hypothèse de récurrence, le nombre d'appels récursifs pour calculer T($\frac{n}{2}$) est $p - 1$. Le nombre d'appels pour calculer T(n) est donc p et la propriété est vérifiée.

Si n est impair, le calcul de T(n) fait appel à T($\frac{n+1}{2}$) or $2^{p-2} < \frac{n+1}{2} \leq 2^{p-1}$, et on mène le même raisonnement que ci-dessus.

On peut donc estimer le temps de calcul à O(p) = O(ln(n)), ce qui est considéré comme très performant. En gros, le temps de calcul est proportionnel au nombre de chiffres de n.

EXEMPLE 2 :
Considérons maintenant la suite de Fibonnaci :

```
def F(n):
```

```
if  n<=1:
    return(1)
else:
    return(F(n-1) + F(n-2))
```

Suivons le calcul de F(5). Nous avons mis des couleurs pour mieux se repérer dans la suite des empilements :

5 est plus grand que 1
appel de F(4)
4 est plus grand que 1
appel de F(3)
3 est plus grand que 1
appel de F(2)
2 est plus grand que 1
appel de F(1)
1 est valeur de base
renvoi de 1 comme valeur de F(1)
appel de F(0)
0 est valeur de base
renvoi de 1 comme valeur de F(0)
renvoi de F(1) + F(0) = 2 comme valeur de F(2)
appel de F(1)
1 est valeur de base
renvoi de 1 comme valeur de F(1)
renvoi de F(2) + F(1) = 3 comme valeur de F(3)
appel de F(2)
2 est plus grand que 1
appel de F(1)
1 est valeur de base
renvoi de 1 comme valeur de F(1)
appel de F(0)
0 est valeur de base
renvoi de 1 comme valeur de F(0)
renvoi de F(1) + F(0) = 2 comme valeur de F(2)
renvoi de F(3) + F(2) = 5 comme valeur de F(4)
appel de F(3)
3 est plus grand que 1
appel de F(2)
2 est plus grand que 1
appel de F(1)
1 est valeur de base
renvoi de 1 comme valeur de F(1)
appel de F(0)
0 est valeur de base
renvoi de 1 comme valeur de F(0)
renvoi de F(1) + F(0) = 2 comme valeur de F(2)
appel de F(1)
1 est valeur de base
renvoi de 1 comme valeur de F(1)
renvoi de F(2) + F(1) = 3 comme valeur de F(3)

renvoi de F(4) + F(3) = 8 comme valeur de F(5)
le résultat est 8

On constate que le nombre d'appels est beaucoup plus important dans le cas présent et surtout que le logiciel ne s'aperçoit pas qu'il calcule plusieurs fois la même valeur. Estimons le nombre d'additions effectuées pour calculer F(n). Soit A(n) ce nombre. Il est nul si n = 0 ou 1, et si $n \geq 2$, on a, d'après la définition de F(n) :

$A(n) = A(n-1) + A(n-2) + 1$

Mais on constate alors que la suite (A(n) + 1) vérifie exactement la définition de la suite de Fibonnaci. On a donc, pour tout n :

$A(n) + 1 = F(n) = O(\left(\frac{1+\sqrt{5}}{2}\right)^n)$ (cf le chapitre SUITES.PDF de première année)

La croissance du temps de calcul en fonction de n est exponentiel, ce qui est numériquement inapplicable. Pour n = 100, il faut effectuer environ 800 milliards de milliards d'additions. En fait, l'algorithme ne fait rien d'autre qu'ajouter 1 + 1 + 1 + ... + 1 avec F(n) – 1 additions.

Il est alors préférable d'utiliser un algorithme itératif dont le temps de calcul est en O(n) :

```
A ← 1                                  # A = F(0)
B ← 1                                  # B = F(1)
pour I ← 1 à n–1 faire C ← B           # C = F(I) = B, A = F(I–1)
                        B ← A+B        # C = F(I), B = F(I+1), A = F(I–1)
                        A ← C          # A = F(I), B = F(I+1)
                        finfaire
```

Le résultat cherché F(n) est dans B.

EXEMPLE 3 :
Considérons le calcul de la longueur de la suite de Collatz

```
def C(n):
    if n==1:
        return(0)
    else:
        if n%2==0:
            return(1+C(n//2))
        else:
            return(1+C(3*n+1))
```

Appliquons cette procédure à n = 3

3 est impair
appel de C(10)
10 est pair
appel de C(5)
5 est impair
appel de C(16)
16 est pair
appel de C(8)
8 est pair
appel de C(4)
4 est pair
appel de C(2)
2 est pair

appel de C(1)
1 est valeur de base
renvoi de 0 comme valeur de C(1)
renvoi de 1 comme valeur de C(2)
renvoi de 2 comme valeur de C(4)
renvoi de 3 comme valeur de C(8)
renvoi de 4 comme valeur de C(16)
renvoi de 5 comme valeur de C(5)
renvoi de 6 comme valeur de C(10)
renvoi de 7 comme valeur de C(3)
le résultat est 7

Cependant, nous avons indiqué qu'en général, on ignore si le calcul de C(*n*) se termine pour tout entier *n*. Il est possible qu'il existe un entier pour lequel le calcul ne se termine pas. Dans ce cas, la pile va croître indéfiniment, et dans un ordinateur réel où la taille de la pile est finie, cela produira une erreur d'exécution par débordement de pile (stack overflow).
Cette erreur peut également se produire pour un algorithme qui, en théorie, se termine, mais nécessite une taille de pile plus grande que celle prévue sur la machine. Ce phénomène ne se produit pas pour les algorithmes itératifs, qui n'utilisent pas de pile.
Enfin, cette erreur de débordement de pile se produit nécessairement pour un algorithme récursif mal conçu qui s'appelle indéfiniment, le cas de base ayant été mal prévu.

EXEMPLE 4 :
Considérons la fonction factorielle définie comme suit :

```
def fact(n):
    if n==0:
        return(1)
    else:
        return(n*f(n-1))
```

Si on appelle cette fonction avec la valeur $n = -1$, on va appeler fact(–2) qui appelle fact(–3) qui appelle fact(–4), ... entraînant un débordement de pile. Dans les logiciels professionnels, le concepteur de l'algorithme doit prévoir un test vérifiant que *n* est bien dans le domaine pour lequel il est prévu.

3- Conversion récursif-itératif

Les algorithmes récursifs utilisant par nature une pile, ce que ne font pas nécessairement les algorithmes itératifs, on souhaite parfois convertir un algorithme récursif en algorithme itératif. Nous donnons ci-dessous quelques éléments permettant de procéder à cette conversion, sans chercher à prétendre à l'exhaustivité de la question.

Commençons par un premier type de récursivité. Soit f et φ deux fonctions prédéfinies sur un ensemble E, et B un ensemble de valeurs données, inclus dans E. Définissons la fonction récursive F par :

si $x \in$ B, $F(x) = f(x)$	cas de base
sinon, $F(x) = F(\varphi(x))$	définition récursive

On parle de récursivité terminale car la définition récursive de F se termine par l'application de F elle-même. On supposera que, pour tout x de E, il existe n tel que $\varphi \circ \varphi \circ \ldots \circ \varphi(x) = \varphi^n(x)$ appartient à

B pour garantir que l'algorithme se termine pour tout x. Si n est le plus petit entier tel que $\varphi^n(x)$ appartient à B, alors $F(x) = f(\varphi^n(x))$. Cela se montre par récurrence sur n.

Si $n = 0$, alors x est dans B et $F(x) = f(x)$

Supposons la propriété vérifiée pour $n - 1$ et soit x tel que n soit le plus petit entier tel que $\varphi^n(x)$ appartient à B. Alors $F(x) = F(y)$ avec $y = \varphi(x)$. Mais le plus petit entier m tel que $\varphi^m(y)$ soit dans B est $m = n - 1$. On peut donc appliquer l'hypothèse de récurrence et :

$F(x) = F(y) = f(\varphi^{n-1}(y)) = f(\varphi^n(x))$

Pour calculer $F(x)$, on applique donc f sur le premier itéré par φ de x qui est dans B. D'où un algorithme itératif de F :

```
tant que (x n'est pas dans B) x = phi(x);
retourner(f(x));
```

EXEMPLE 1 *: L'EXPONENTIATION RAPIDE*

Dans l'exemple ci-dessous, $n//2$ désigne le quotient de la division euclidienne de n par 2. a et T étant deux réels et n un entier naturel, on peut définir le nombre $T \times a^n$ récursivement par :

$Ta^n = T$	si $n = 0$	définition de base
$= T\ (a^2)^{n//2}$	si n est pair	
$= Ta\ (a^2)^{n//2}$	si n est impair	

Posons donc :

x	(a, n, T)
$F(x)$	a^nT
$\varphi(a, n, T)$	si n pair alors $(a^2, n/2, T)$ sinon $(a^2, n/2, aT)$
$f(x)$	T
B	$\{(a, 0, T)\}$

$x \in$ B est équivalent à $n = 0$.

F est bien défini récursivement comme suit :

$F(x) = f(x)$ si $x \in$ B

$= F(\varphi(x))$ sinon

autrement dit :

Si (n == 0) alors $F(a, n, T) = T$

sinon si n (pair) alors $F(a, n, T) = F(a^2, n/2, T)$

sinon $F(a, n, T) = F(a^2, n/2, a * T)$

La traduction itérative devient :

```
tant que n ≠ 0 faire
        si n impair alors T ← a * T finsi
        a ←a²
        n ← n/2
        finfaire
retourner(T)
```

On reconnaît exactement l'algorithme donné dans le cours de première année (ALGO1.PDF) en annexe de la multiplication égyptienne. Le temps de calcul est un $O(\ln(n))$, plus rapide que le calcul du produit par a, n fois de suite.

EXEMPLE 2 : *LES NOMBRES DE FIBONACCI*

L'algorithme d'exponentiation rapide donné ci-dessus donne également un moyen très efficace de calculer la suite de Fibonacci, partant de $F_0 = 0$ et $F_1 = 1$. En effet, par récurrence sur n :

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

Le calcul de F_n peut être obtenu par celui de $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$, et l'algorithme d'exponentiation rapide permet d'effectuer ce calcul en un temps $O(\ln(n))$ (proportionnel au nombre de chiffres de n) au lieu de $O(n)$ pour l'agorithme itératif donné plus haut, et pire encore, $O((\frac{1+\sqrt{5}}{2})^n)$ pour la procédure récursive naïve initial.

Partons donc de $T = I_2$, et $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix}$. On peut considérer T comme égal à A^0 à condition de poser $F_{-1} = 1$. Il suffit alors de donner les matrices par les composantes de leur deuxième colonne, à savoir $\begin{pmatrix} a \\ b \end{pmatrix}$ pour représenter $A = \begin{pmatrix} a+b & a \\ a & b \end{pmatrix}$ et $\begin{pmatrix} p \\ q \end{pmatrix}$ pour représenter $T = \begin{pmatrix} p+q & p \\ p & q \end{pmatrix}$. L'algorithme d'exponentiation rapide appliqué dans ce cas donne :

```
a ← 1                    # initialisation de A
b ← 0
p ← 0                    # initialisation de T
q ← 1
tant que n ≠ 0 faire
    si n impair alors temp ← p          # calcul de T ← A × T
                      p ← (a + b)p + aq
                      q ← a temp + bq
    finsi
    temp ← a                            # calcul de A ← A²
    a ← a² + 2ab
    b ← temp² + b²
    n ← n/2
finfaire
```

Le résultat F_n est dans p.

Voici le programme en Python :

```
def fibrapide(n):
    a=1
    b=0
    p=0
    q=1
    while n<>0:
        if  n%2==1:
            temp=p
            p=(a+b)*p + a*q
            q=a*temp + b*q
        temp=a
        a=a**2 + 2*a*b
        b=temp**2 + b**2
        n=n//2
    return(p)
```

On pourra comparer les temps de calcul pour un n de l'ordre de quelques centaines de mille. L'algorithme précédent s'applique en une fraction de seconde, alors que l'itération donnée au II-2 prend plusieurs secondes :

```
import datetime
n = 200000
topdebut = datetime.datetime.today()
r = fibrapide(n)
topfin = datetime.datetime.today()
duree = topfin - topdebut
print(duree.total_seconds())
```

III : Tri de tableaux

On cherche à trier un tableau numérique par ordre croissant. De nombreux algorithmes existent. Nous en présentons quelques-uns. Les indices du tableau varient entre 0 et $n - 1$, où n est le nombre d'éléments du tableau.

1- Le tri par sélection

Il consiste à rechercher le plus petit élément du tableau [T[0], ..., T[n–1]], et à le permuter avec l'élément de tête. On itère le procédé en triant [T[1], ..., T[n–1]] de la même façon. La description précédente est récursive, mais on peut donner directement un algorithme itératif. On suppose $n \geq 2$.

L'invariant de la boucle portant sur i est :

$T[0] \leq T[1] \leq ... \leq T[i-1] \leq \text{Min}(T[i], ..., T[n-1])$

Pour la valeur initiale $i = 0$, la propriété précédente est vide. Supposons-la vraie au début d'une boucle $i \geq 0$. Les instructions qui suivent déterminent la valeur m de Min(T[i], ..., T[n–1]) ainsi que l'indice $j \geq i$ tel que $T[j] = m$. Une fois j ainsi déterminé, on a donc :

$T[0] \leq T[1] \leq ... \leq T[i-1] \leq T[j] = \text{Min}(T[i], ..., T[n-1])$

Il suffit alors de permuter les valeurs de T[i] et T[j] pour obtenir :

$T[0] \leq T[1] \leq ... \leq T[i-1] \leq T[i] = \text{Min}(T[i], ..., T[n-1]) \leq \text{Min}(T[i+1], ..., T[n-1])$

qui est bien l'invariant de boucle au rang i suivant. On termine l'algorithme avec la dernière boucle pour $i = n - 2$, ce qui donnera comme valeur finale de l'invariant :

$T[0] \leq T[1] \leq ... \leq T[n-3] \leq T[n-2] \leq \text{Min}(T[n-1]) = T[n-1]$

```
pour i de 0 à n–2 faire
        m ← T[i]                          # on cherche Min(T[i], ... T[n–1])
        j ← i
        pour k de i+1 à n–1 faire         # m = T[j] = Min(T[i], ..., T[k–1])
                si T[k] < m alors m ← T[k]
                                  j ← k
                finsi                     # m = T[j] = Min(T[i], ..., T[k])
        finfaire                          # m = T[j] = Min(T[i], ..., T[n–1])
        T[j] ← T[i]                       # on échange T[i] et T[j]
        T[i] ← m
finfaire
```

La variable m étant en permanence égale à T[j], on pourrait s'en passer. Nous l'avons laissé car l'algorithme nous paraît plus compréhensible ainsi.

On peut estimer la complexité en temps de l'algorithme en estimant le nombre de comparaisons effectuées et le nombre de changement de valeurs d'un élément du tableau. La seule comparaison est le test T[k] < m et ce test est inclus dans une double boucle où i varie de 0 à $n - 2$, et pour chaque i, k

varie de $i + 1$ jusqu'à $n - 1$. Pour chaque i, le nombre d'itérations de k est $n - 1 - i$, donc le nombre total de comparaisons effectuées est :

$$\sum_{i=0}^{n-2} (n - 1 - i) = \sum_{p=1}^{n-1} p \qquad \text{en posant } p = n - 1 - i$$

$$= \frac{n(n-1)}{2} = O(n^2)$$

Les changements de valeurs d'éléments du tableau se produisent à la fin de chaque itération sur i, quand on échange T[i] et T[j], soit $2(n - 1) = O(n)$ changements.

Le nombre d'itérations est assez coûteux. Nous verrons ci-dessous des algorithmes plus efficaces. Par contre le nombre de changements de valeurs est raisonnable. Enfin, l'occupation en mémoire est réduite. En dehors du tableau, il n'y a que trois variables entières ainsi que la variable m dont nous avons indiqué qu'on pouvait s'en passer.

En Python, l'algorithme se traduit de la façon suivante, en utilisant le type array du module numpy.

```
import numpy as np
def triselect(T):
    n=size(T)
    for i in range(n-1):
        m=T[i]
        j=i
        for k in range(i+1,n):
            if T[k]<m:
                m=T[k]
                j=k
        T[j]=T[i]
        T[i]=m
```

Attention, c'est le tableau lui-même qui se trouve trié et non une copie. On a donc perdu l'ordre initial du tableau. Voici ci-dessous un exemple d'instructions permettant de définir un tableau T aléatoire, de le copier dans un tableau U, de trier T avec la procédure précédente, et de trier U avec la commande prédéfinie de tri de Python. On pourra comparer les résultats finaux.

```
T = np.random.random(15)
U = T.copy()
print(T)
triselect(T)
print(T)
print(U)
U.sort()
print(U)
```

2- Le tri par insertion

Supposons triée la partie T[0], ..., T[i–1]. Le tri par insertion consiste à insérer T[i] au bon endroit de sorte que les valeurs finales de T[0], ..., T[i] soient triées. Il suffit pour cela de comparer T[i] avec l'élément du tableau qui précède et, s'il est plus petit, de le permuter avec celui-ci. On itère cette démarche jusqu'à trouver un élément du tableau qui précède qui soit plus petit ou jusqu'à ce qu'on remonte jusqu'au début du tableau.
Pour obtenir un tableau entièrement trié, il suffit d'effectuer la démarche précédente pour i variant de 1 à $n - 1$.

```
def triinsert(T):
    n=T.size
    for i in range(1,n):
        j=i
        while (j>0) and (T[j]<T[j-1]):
            temp=T[j]
            T[j]=T[j-1]
            T[j-1]=temp
            j=j-1
```

Dans le meilleur des cas (le tableau est déjà trié), on se borne à tester T[i] avec T[i–1] pour i variant de 1 à n – 1 sans effectuer la moindre permutation. Le temps de calcul est alors en O(n). Par contre, si dans le pire des cas (le tableau est trié mais par ordre décroissant), pour chaque i, j va décroître de i jusqu'à 0. Comme dans le tri par sélection, le temps de calcul sera en O(n^2).
On peut diminuer le nombre de comparaisons à faire en procédant par dichotomie, mais dans le cas d'un tableau, la mise en place de T[i] au bon endroit nécessite de décaler les éléments suivants du tableau et on ne peut éviter que le nombre d'affectation de variables soit un O(n^2). Cet algorithme est donc efficace pour des tableaux déjà en partie triés, qui ne nécessiteront que quelques déplacements d'éléments mal placés.

3- Tri par fusion

C'est typiquement un tri dont l'algorithme est récursif. On commence par trier récursivement les deux sous-tableaux [T[0],...,T[$\frac{n}{2}$ – 1]] et [T[$\frac{n}{2}$],...,T[n – 1]]. Notons L_1 et L_2 les deux sous-tableaux triés.

On les fusionne ensuite pour obtenir le tableau trié complet. Pour cela, en partant de $i = j = k = 0$, on compare $L_1[i]$ et $L_2[j]$.

Si c'est $L_1[i]$ le plus petit, on le stocke dans T[k] et on augmente i et k de 1
Si c'est $L_2[j]$ le plus petit, on le stocke dans T[k] et on augmente j et k de 1

Il faut également veiller à ce que i et j ne dépasse pas les limites de leur tableau respectifs. Dans ce dernier cas, on ajoute en fin de tableau T le reliquat des listes L_1 et L_2 (dont au moins un des deux est vide). On utilise une fonction `Fusion` de deux listes triées, écrite plus haut :

```
def trifusion(T):
    n=len(T)
    if n<=1:
        return(T)
    else:
        L1=trifusion(T[0:n//2])
        L2=trifusion(T[n//2:n])
        return(Fusion(L1,L2))
```

Estimons grossièrement le temps de calcul $t(n)$ d'un tableau de n éléments. En négligeant le fait qu'il conviendrait de distinguer n pair et n impair, chaque sous-tableau L_1 et L_2 possède $\frac{n}{2}$ éléments et le temps de calcul pour trier chacun de ces deux sous-tableaux est $t(\frac{n}{2})$. Par ailleurs, la fusion finale demande un nombre d'opérations à peu près proportionnel à n puisqu'on remplit le tableau trié élément par élément. On a donc une relation du type :

$t(n) \leq 2t(\frac{n}{2}) + Kn$ où K est une constante

Posons $n = 2^p$ et $u(p) = t(2^p)$. La relation précédente s'écrit :

$$u(p) \leq 2u(p-1) + K\,2^p$$

Donc :

$$\frac{u(p)}{2^p} \leq \frac{u(p-1)}{2^{p-1}} + K$$

Pour p entier, on obtient par récurrence :

$$\frac{u(p)}{2^p} \leq (p-1)K + \frac{u(1)}{2} = O(p)$$

Nous admettrons que cette relation reste valide pour p non entier. On obtient alors :

$$t(n) = u(p) = O(p2^p) = O(n\log_2(n))$$

Ce temps de calcul est incomparable plus petit que $O(n^2)$. Pour $n = 1000$, $n^2 = 10^6$ alors que $n\log_2(n)$ vaut environ 10^4, soit 100 fois moins.

4- Tri rapide

Si, dans l'algorithme du tri par fusion les deux sous-tableaux sont tels que les éléments du premier sont inférieurs à ceux de second, alors la fusion se limite à une simple concaténation des éléments du premier tableau, suivis des éléments du second. L'idée est donc de créer les deux sous-tableaux en disposant à gauche les éléments inférieurs ou égaux à une valeur x, et à droite les éléments supérieurs à x. On trie ensuite les deux sous-tableaux récursivement. L'idéal serait de choisir x comme valeur médiane du tableau de façon à ce que les deux sous-tableaux soient à peu près de même taille. Mais la détermination de cette médiane n'est pas aisée. On se bornera à prendre pour x la valeur située au centre du tableau, ce qui présente l'avantage que, si le tableau est déjà partiellement trié, x ne devrait pas être trop éloigné de cette médiane.

Voici le programme, en Python. La procédure admet comme paramètre le tableau T ainsi que deux indices p et q. On trie la partie du tableau [T[p], T[p+1], ..., T[q]]. On a indiqué en bleu quelques commentaires permettant de prouver la validité de l'algorithme. Le lecteur est invité à les vérifier.

```
def trirapide(T,p,q):
    if p<q:
        x = T[(p+q)//2]
        i = p
        j = q
        while i<=j:
# Invariant de boucle : T[p], ..., T[i-1] ≤ x ≤ T[j+1], ..., T[q]
            while T[i]<x:
                i = i+1
            while T[j]>x:
                j = j-1
# On a maintenant T[p], ..., T[i-1] ≤ x ≤ T[j+1], ..., T[q],
# et  T[i] ≥ x ≥ T[j]
            if i<=j:
# On échange les valeurs de T[i] et T[j] si i ≤ j
                temp = T[i]
                T[i] = T[j]
                T[j] = temp
            i = i+1
            j = j-1
# On a à nouveau T[p], ..., T[i-1] ≤ x ≤ T[j+1], ..., T[q]
# On quitte la boucle quand i > j.
# Dans ce cas, les valeurs T[j+1], ..., T[i-1] sont toutes égales
à x.
# Il suffit donc de trier [T[p], ..., T[j]] et [T[i], ..., T[q]]
```

```
trirapide(T,p,j)
trirapide(T,i,q)
```

On obtient le tri du tableau complet en appelant trirapide(T,0,n-1) si n est le nombre d'éléments du tableau.

On peut prouver de la façon suivante que l'algorithme se termine. Dans la boucle tant que i ≤ j, on remarque que i est incrémenté ou que j est décrémenté. Si aucun des deux ne varie lors des boucles tant que T[i]<x et tant que T[j]>x, comme $i \leq j$, c'est dans l'instruction conditionnelle qui suit qu'ils voient leur valeur modifiée. Il en résulte que $i - j$ croît strictement et que i finira par dépasser j. En outre i croît au moins une fois, et j décroît au moins une fois pour la même raison. Il en résulte que l'intervalle final $[p, j]$ et l'intervalle final $[i, q]$ sont tous deux strictement inclus dans $[p, q]$. On applique donc une récursivité sur des intervalles de plus en plus petit, ce qui aboutit à coup sûr au cas de base, celui où l'intervalle a moins de un élément. Dans ce cas, aucune action n'est appliquée.

Le lecteur pourra réfléchir que le fait de remplacer le test $T[i] < x$ (respectivement $T[j] > x$) par le test $T[i] \leq x$ (respectivement $T[j] \geq x$) peut entraîner une erreur d'exécution. Il est indispensable de conserver des inégalités strictes. Pourquoi ?

Le temps d'exécution est en général, comme pour le tri par fusion, en $O(n\ln(n))$, à condition qu'à chaque appel de procédure, les deux sous-tableaux soient à peu près de même taille. Cette condition n'est cependant pas assurée et il peut exister des configurations conduisant à deux sous-tableaux, l'un de taille 1, l'autre de taille $n - 1$ par exemple. Si cette répartition malheureuse se reproduit récursivement, le temps de calcul sera en $O(n^2)$. Ces configurations sont cependant rares et l'algorithme de tri rapide est l'un des plus rapides qui soit. On pourra le tester sur des tableaux aléatoires de plusieurs milliers de données numériques. Le tri par sélection ou par insertion demanderont plusieurs dizaines de secondes alors que le tri par fusion ou le tri rapide ne prendront qu'une fraction de seconde.

Si les données du tableau sont initialement rangées au hasard, toutes les permutations possibles étant équiprobables, on peut montrer que l'espérance du temps de calcul est en $O(n\ln(n))$. Pour évaluer le temps de calcul, nous évaluons le nombre de comparaisons entre éléments du tableau. Soit C_n ce nombre de comparaisons si ce tableau comporte n éléments. Toutes les répartitions étant équiprobables, la probabilité que le pivot x choisi soit le k-ème élément du tableau trié ne dépend pas de k. Dans le cas d'un tableau de $n + 1$ éléments, la probabilité de choisir un tel pivot est $\frac{1}{n+1}$. Le nombre de comparaisons à faire est alors :

$$C_{n+1} = n + \frac{1}{n+1} \left(\sum_{k=1}^{n+1} C_{k-1} + C_{n+1-k}\right) \qquad \text{avec } C_0 = C_1 = 0$$

Le premier n dans la somme de droite correspond aux n comparaisons que l'on fera entre les éléments du tableau avec le pivot pour les disposer dans l'un des deux sous-tableaux. Si le pivot est le futur k-ème élément du tableau trié, l'un des sous-tableau possède $k - 1$ éléments et il faudra en moyenne C_{k-1} comparaisons pour le trier, l'autre aura $n + 1 - k$ éléments pour le trier. On pondère ces moyennes par la probabilité $\frac{1}{n+1}$ de choisir le k-ème pivot, et l'on somme sur toutes les valeurs possibles de k. (C'est une variante des probabilités totales, mais appliquées aux espérances). En effectuant le changement d'indice $k - 1 \to k$ dans la première somme et $n + 1 - k \to k$ dans la deuxième somme, on obtient :

$$C_{n+1} = n + \frac{2}{n+1} \sum_{k=1}^{n} C_k$$

ou $\quad (n+1)C_{n+1} = n(n+1) + 2\sum_{k=1}^{n} C_k$

Au rang n on a :

$$nC_n = (n-1)n + 2\sum_{k=1}^{n-1} C_k$$

En retranchant membre à membre, on obtient :

$(n+1)C_{n+1} - nC_n = 2n + 2C_n$

$\Leftrightarrow \quad (n+1)C_{n+1} = (n+2)C_n + 2n$

$\Leftrightarrow \quad \frac{C_{n+1}}{n+2} = \frac{C_n}{n+1} + \frac{2n}{(n+1)(n+2)}$

d'où, par récurrence :

$$\frac{C_n}{n+1} = \sum_{k=1}^{n-1} \frac{2k}{(k+1)(k+2)}$$

On a donc :

$$\frac{C_n}{n+1} = \sum_{k=1}^{n-1} \frac{4}{k+2} - \frac{2}{k+1} = 4\sum_{k=3}^{n+1} \frac{1}{k} - 2\sum_{k=2}^{n} \frac{1}{k} = \frac{4}{n+1} - 4 + 2\sum_{k=1}^{n} \frac{1}{k}$$

Mais par comparaison série-intégrale, on a $\sum_{k=1}^{n} \frac{1}{k} \sim \ln(n)$. Donc :

$$\frac{C_n}{n+1} \sim 2\ln(n)$$

et $\quad C_n \sim 2n\ln(n)$

5- Calcul de la médiane d'un tableau

Dans le cas d'un tableau ayant un nombre impair d'éléments, la médiane est le nombre x qui serait an centre du tableau ci-celui ci était trié. Par exemple, dans le tableau [1,6,5,3,8,6,7], la médiane est 6. Dans le cas où le tableau possède un nombre pair d'éléments, on prend la moyenne entre les deux éléments qui seraient centraux si le tableau était trié. Par exemple, dans le tableau [1,6,5,3,8,6,7,2], la médiane est 5,5. En procédant ainsi, la complexité en temps du calcul d'une médiane est la même que celle du tri d'un tableau, soit $O(n\ln(n))$ par exemple.

On peut cependant mettre au point un algorithme dont on peut montrer qu'en moyenne, il calcule la médiane en un temps en $O(n)$. Il suffit pour cela d'appliquer le tri rapide, mais de ne garder que le sous-tableau dont on sait qu'il contient la médiane, l'autre étant inutile. Le gain de temps obtenu en ne triant pas le sous-tableau inutile suffit à faire passer la complexité de $O(n\ln(n))$ à $O(n)$. De manière plus précise, l'algorithme récursif pour déterminer le k-ème élément du tableau par ordre de tri est le suivant :

Procèdure ChercheElem(Tableau T ayant n éléments, rang de l'élément cherché k)

    Choisir un pivot x

Comparer les autres éléments du tableau à ce pivot pour les séparer en deux sous-tableaux, celui T_i des éléments inférieurs strictement à x et celui T_s des éléments strictement supérieurs à x.

Si le premier sous-tableau possède $k - 1$ éléments, alors x est l'élément cherché

Sinon, si le premier sous-tableau possède plus de k éléments, appliquer ChercheElem(T_i, k)

sinon appliquer ChercheElem(T_s, $k - 1 -$ longueur(T_i))

Pour chercher la médiane, on prendra initialement $k = \lfloor \frac{n+1}{2} \rfloor$.

◆