INFORMATIQUE 2ème année

PLAN

I : Récursivité

1- Définitions récursives

Un objet d'un type donné est défini récursivement lorsque la définition 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 :

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

Dans la suite de ce chapitre, cette définition récursive sera qualifiée de naïve, car nous verrons qu'elle n'est pas très efficace.

EXEMPLE 2 :

❑ La **suite de Thue**, définie pour $n \geq 0$, est telle que son n-ème terme $T(n)$ est la somme modulo 2 des bits de la décomposition binaire de n. En raisonnant sur le bit des unités, qui vaut 0 si n est pair et 1 si n est impair, on voit que :

$T(0) = 0$ définition de base

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

On remarque que, si n est pair strictement positif, $\frac{n}{2} < n$ et si n est impair, $\frac{n-1}{2} < n$ 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 par la définition récursive. L'exemple suivant est plus délicat.

EXEMPLE 3 :

❑ La longueur de la **suite de Collatz** (voir le chapitre L1/ALGO1.PDF pour la définition de cette suite) vérifie :

$C(1) = 0$ 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 récursive 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 d'objets y, z, etc..., alors :

$\varphi(y) < \varphi(x)$, $\varphi(z) < \varphi(x)$, etc. (et en particulier $\varphi(x) > 0$)

Montrons alors par récurrence sur la valeur $n = \varphi(x)$ que l'objet x est défini en un nombre fini d'étapes. Si $n = 0$, x est un objet de base et est défini. Supposons l'hypothèse de récurrence vérifiée jusqu'à un rang n et montrons-la pour tout objet x tel que $\varphi(x) = n + 1$. x n'est pas un objet de base et est défini récursivement à partir d'objets y, z, etc. Chacun de ces objets vérifie $\varphi(y) < \varphi(x)$, $\varphi(z) < \varphi(x)$, etc. Ils vérifient donc $\varphi(y) \leq n$, $\varphi(z) \leq n$, etc. et, d'après l'hypothèse de récurrence, ils sont définis en un nombre fini d'étapes. Il en est donc de même de $\varphi(x)$.

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

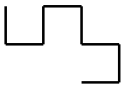
deux fois  trois plis

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

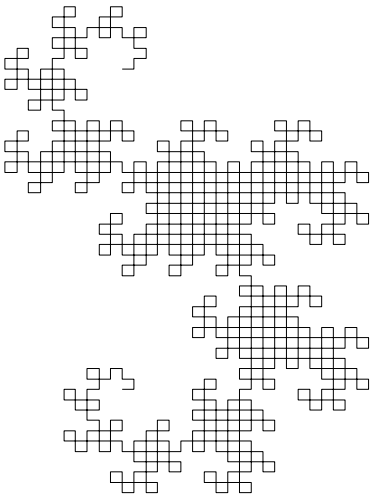
trois fois  GGDGGDD

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 est également possible de définir très simplement la suite S de lettres G ou D (ou mot) 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 = ∑G∑* où ∑ est une suite du dragon	définition récursive

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

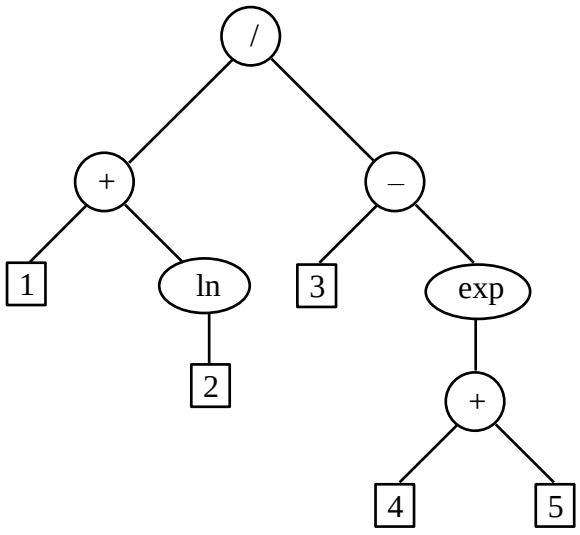
S_1 = G
S_2 = G**G**D
S_3 = GGD**G**GDD
S_4 = GGDGGDD**G**GGDDGDD
S_5 = GGDGGDDGGGDDGDD**G**GGDGGDDDGGDDGDD
S_6 = GGDGGDDGGGDDGDDGGGDGGDDDGGDDGDD**G**
GGDGGDDGGGDDGDDDGGDGGDDDGGDDGDD
etc ...



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'utilisateur 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*), où *val* est la valeur attribuée à [*liste*].

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*(*val*1, *val*2), où *val*1 et *val*2 sont les valeurs attribuées respectivement à [*liste*1] et [*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 $\det(A) = \sum_{i=1}^{n} (-1)^{i+1} a_{i1} \det((a_{kl}), k \neq i, l \neq 1)$ définition récursive

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 aléatoire d'un ensemble fini E, on peut procéder comme suit :

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

définition récursive : Sinon, placer en tête un élément *x* de E choisi au hasard (E est généralement muni d'une loi uniforme dans ce contexte) et permuter E \ {*x*}.

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. Dans ce langage, les fins d'instructions conditionnelles ou itératives sont repérées par indentation des paragraphes mais nous avons préféré les confirmer par un commentaire, rendant les programmes plus lisibles pour un lecteur non familier de ce langage.

EXEMPLE 1 :

❑ **La suite de Fibonacci**.

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

On voit qu'il suffit de traduire la définition récursive. Malheureusement, dans le cas présent, le temps de calcul récursif devient très important dès que *n* vaut quelques dizaines (exécuter le

programme précédent sur un ordinateur et 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. Ci-dessous, n//2 désigne le quotient entier de la division euclidienne de *n* par 2.

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

Voici les premières valeurs de la suite, *n* variant de 0 à 39 (la dernière valeur 40 est exclue) :

```
[T(n) for n in range(40)]
```

[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)
    elif n%2==0:
        return(1+C(n//2))
    else:
        return(1+C(3*n+1))
    # fin if
```

Voici les premières valeurs de la suite :

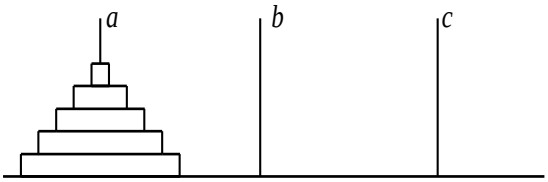
```
[C(n) for n in range(1,41)]
```

[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]

Comme expliqué plus haut, on ignore actuellement si l'algorithme se termine pour tout entier *n*.

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. Les disques sont déplacés un par un. 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 la fonction suivante, les paramètres indiquant le nom de la pile de départ, celui de la pile d'arrivée et celui de la pile auxiliaire :

```
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)
    # fin if
```

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 :

❑ 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. On ajoute quelques commentaires précédés d'un # :

```
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 de la liste 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)
            # fin for
        # fin for
        return(M)
    # fin if
```

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 6 :

❑ 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 lenLigne1 + 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=list(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)
                # fin for
            # fin if
        # fin for
    # fin if
    return(ListeN)
```

Pour créer la liste des tableaux de Young d'une longueur *n* donnée, il suffit d'appeler la fonction précédente avec les paramètres 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])
        # fin for
        print(' ')
    # fin for
```

EXEMPLE 7 :
❑ 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 8 :
❑ 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 nombres 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. Les programmes qui suivent sont écrits en Python. Dans ce langage, l'indice d'une liste commence à 0.

```
def FusionRecursive(A,B):
    if A==[]:           # la liste A est vide
        return(B)
    elif B==[]:         # la liste B est vide
        return(A)
```

```
    else:               # les deux listes sont non vides
        # on retire le dernier élément de A et on le stocke dans a
        a=A.pop()
        # idem pour b
        b=B.pop()
        if a<b:
            # on remet l'élément a en fin de liste A
            A.append(a)
            C=FusionRecursive(A,B)
            # on ajoute l'élément b en fin de la liste fusionnée C
            C.append(b)
            return(C)
        else:
            B.append(b)
            C=FusionRecursive(A,B)
            C.append(a)
            return(C)
        # fin if
    # fin if
```

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

```
def FusionIterative(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
        # fin if
    # fin while
    # 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
    # fin while
    while (j<lB):
        C.append(B[j])
        j=j+1
    # fin while
    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 :
❑ Prouvons 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.

On reconnaît ci-dessus un raisonnement par récurrence classique.

EXEMPLE 2 :
❑ Montrons 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.

EXEMPLE 3 :
❑ 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 n'a varié.
- 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 binaires) et *liste*2 en possède n_2 (et donc $n_2 - 1$ opérateurs binaires), 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.

II : Piles

1- Définition

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

L.empile(x) ou L.push(x)	qui empile l'élément *x* en queue de liste
x ← L.pop() ou x ← L.depile()	qui retire le dernier élément de la liste et le stocke dans une variable *x*. Seul le dernier élément d'une pile est directement accessible.

Ces deux méthodes se font en un temps borné, indépendant du nombre d'éléments contenus dans la pile. Ce nombre d'éléments est souvent qualifié de **hauteur** de la pile, alors qu'on parle plutôt de longueur d'une liste, comme si on visualisait une pile comme étant verticale, et une liste comme horizontale. Le dernier élément de la pile est alors celui situé au sommet de la pile.

On dispose enfin d'une fonction à valeur booléenne permettant de tester si la pile est vide. Nous représenterons la pile vide par [ ]. Une condition du type L=[ ] permet de tester si la liste est vide. Selon le langage utilisé, la méthode pop appliquée à une liste vide peut ou bien entraîner une erreur d'exécution, ou bien voir attribuer à *x* une valeur spécifique nommée NULL. Dans le premier cas, il est indispensable de tester si la pile L est non vide avant d'exécuter pop, à moins d'être sûr que la pile contienne au moins un élément. Dans le second cas, un test x=NULL permet de tester si la pile est vide.

Les piles servent à stocker des objets à traiter ultérieurement, le dernier objet 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 :

$y \leftarrow f(x)$

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 $y \leftarrow f(x)$ 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==0:
        return(0)
    elif n%2==0:
        return(T(n//2))
    else:
        return(1-T((n-1)//2))
    # fin if
```

Voici les empilements (gérés automatiquement par la machine) qui ont lieu lors de la demande de calcul de T(24). Chaque décalage de l'indentation vers la droite représente un empilement, chaque décalage vers la gauche un dépilement :

24 est pair

appel de T(12)
    12 est pair
    appel de T(6)
        6 est pair
        appel de T(3)
            3 est impair
            appel de T(1)
                1 est impair
                appel de T(0)
                    0 est valeur de base de T
                    renvoi de 0 comme valeur de T(0)
                renvoi de 1 – T(0) = 1 comme résultat de T(1)
            renvoi de 1 – T(1) = 0 comme résultat de T(3)
        renvoi de T(3) = 0 comme résultat de T(6)
    renvoi de T(6) = 0 comme résultat de T(12)
renvoi de T(12) = 0 comme résultat de T(24)
le résultat est 0

On voit que l'exécution, transparente pour l'utilisateur, est assez complexe. Elle est cependant efficace dans le cas présent. Montrons que, pour $n > 0$, si p est l'entier tel que $2^{p-1} \leq n < 2^p$, 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 avec un appel récursif à la valeur T(0) = 0. 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 $2^{p-2} \leq \frac{n}{2} < 2^{p-1}$, 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 $2^{p-2} \leq \frac{n-1}{2} < 2^{p-1}$, et on mène le même raisonnement que ci-dessus.

Comme on a $p - 1 \leq \frac{\ln(n)}{\ln(2)} < p$, p est un O(ln(n)). En supposant que le temps de calcul des divisions par 2, additions et soustractions, est borné[1], il existe une constante K, tel que le temps de calcul de T(n) soit majoré par pK donc est aussi un 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 Fibonacci :

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

[1] Ce n'est pas tout à fait exact. Diviser un nombre n par 2 par exemple demande un temps de calcul qui varie proportionnellement à son nombre de chiffres p, donc qui est un O(ln(n)). Le temps de calcul de T(n) est donc plutôt majoré par une quantité du type K(ln(1) + ln(2) + ln(3) + ... + ln(p)) = O($p$ln(p)) = O(ln(n)ln(ln(n)). Cependant, la croissance de ln(ln(n)) est très faible, et confondre ici O(ln(n)ln(ln(n))) avec O(ln(n)) est un abus acceptable.

```
        return(F(n-1)+F(n-2))
    # fin if
```

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 0 comme valeur de F(0)
renvoi de F(1) + F(0) = 1 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) = 2 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 0 comme valeur de F(0)
renvoi de F(1) + F(0) = 1 comme valeur de F(2)
renvoi de F(3) + F(2) = 3 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 0 comme valeur de F(0)
renvoi de F(1) + F(0) = 1 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) = 2 comme valeur de F(3)

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

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 $n = 1$, et si $n \geq 2$, on a, d'après la définition récursive de F(n) :

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

à savoir A(n – 1) additions pour calculer F(n – 1), A(n – 2) additions pour calculer F(n – 2), et l'addition finale des valeurs de F(n – 1) et F(n – 2) pour calculer F(n). Mais on constate alors que la suite $(A(n) + 1)_{n\geq 0}$ vérifie exactement la définition de la suite $(F(n + 1))_{n\geq 0}$. On a donc, pour tout n :

$A(n) + 1 = F(n + 1)$

$= O((\frac{1+\sqrt{5}}{2})^{n+1})$ (cf les exercices du chapitre L1/SUITES.PDF)

$= O((\frac{1+\sqrt{5}}{2})^{n})$

La croissance du temps de calcul en fonction de n est exponentiel, ce qui est numériquement inapplicable. Pour $n = 100$, il faut effectuer plusieurs centaines de milliards de milliards d'additions. En fait, l'algorithme ne fait rien d'autre qu'ajouter des 0 et des 1 en $F(n + 1) – 1$ opérations.
On peut remédier à cet inconvénient en mémorisant chaque valeur F(n) dès qu'elle est calculée une fois pour éviter de la calculer une deuxième fois. Cette technique de programmation s'appelle **mémoïsation** (sic), et certains langages de programmation la prévoient, avec plus ou moins de facilité. Par exemple, avec Maple, il suffit d'ajouter une option remember au début du corps de la définition récursive de la fonction pour que la mémoïsation soit automatiquement mise en oeuvre. Dès lors, Maple teste d'abord si F(n) n'aurait pas déjà été mémorisé avant de lancer un appel récursif de fonctions pour son calcul.

Pour un calcul ponctuel de F(n), il est encore plus efficace d'utiliser un algorithme itératif dont le temps de calcul est en O(n) et qui ne nécessite pas de retenir de manière définitive les valeurs calculées de la suite de Fibonacci :

```
A ← 0                          # 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)
    elif n%2==0:
        return(1+C(n//2))
```

```
    else:
        return(1+C(3*n+1))
    # fin if
```

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 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.
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.

EXEMPLE 4 :
❑ Considérons la fonction factorielle définie comme suit :

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

```
    # fin if
```

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 exemples de telles conversions, sans chercher à prétendre à l'exhaustivité de la question.

a) Récursivité terminale

Supposons qu'on dispose de deux ensembles E et E', d'une partie non vide B de E (dont les éléments seront les éléments de base de la définition récursive), d'une fonction $f : B \to E'$, et d'une fonction $g : E \setminus B \to E$. Définissons la fonction récursive $F : E \to E'$ par :

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

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

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

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

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

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

```
tant que x n'est pas dans B faire
        x ← g(x)
finfaire
retourner(f(x));
```

Certains compilateurs sont capables de reconnaître qu'une récursivité est terminale et de transformer d'eux-même l'algorithme récursif en algorithme itératif, évitant ainsi tout débordement de la pile d'exécution.

Remarquons que, s'il n'existe aucun n tel que $g^n(x)$ appartienne à B, alors la définition récursive ne permet pas de définir $F(x)$, la pile des appels récursifs croissant indéfiniment, et l'algorithme itératif précédent ne termine pas non plus, la boucle tant que se poursuivant indéfiniment.

EXEMPLE : LA DICHOTOMIE

❑ La recherche par dichotomie d'un élément dans un tableau trié relève d'une récursivité terminale portant sur des intervalles.
On dispose d'un tableau T d'éléments triés par ordre croissant, par exemple des entiers relatifs. On cherche s'il possède le nombre 0 entre les rang a et b, et à quel rang. A défaut, la recherche s'arrêtera sur le premier élément strictement positif ou le dernier élément strictement négatif. On part du couple (a, b), avec $a \leq b$.

Si $a = b$, la recherche est terminée.

Sinon, on a $a < b$. On prend l'élément au centre de cet intervalle $(a + b)/2$, désignant ici le quotient de la division euclidienne de $a + b$ par 2. On remarque que $a \leq (a + b)/2 < b$, car si $a + b$ est pair, alors $2a < a + b < 2b \Rightarrow a < (a + b)/2 < b$, et si $a + b$ est impair, alors $(a + b)/2 = (a + b - 1)/2$ et $2a \leq a + b - 1 < 2b \Rightarrow a \leq (a + b)/2 < b$. On remplace l'intervalle de recherche par $[a, (a + b)/2]$ ou $[(a + b)/2 + 1, b]$ selon le signe de $T[(a + b)/2]$. Si on note $F(a, b)$ le rang final donné par l'algorithme de recherche, on a :

x	couple (a, b)
B	$\{(a, b), a = b\}$
$f(x)$	a
$F(x)$	rang r élément de $[\![a, b]\!]$ tel que : $T[r] = 0$ ou ($r = a$ et $T[a] > 0$) ou ($r = b$ et $T[b] < 0$) ou ($r < b$ et $T[r] < 0$ et $T[r + 1] > 0$) ou ($r > a$ et $T[r - 1] < 0$ et $T[r] > 0$)
$g(x)$	couple $((a + b)/2 + 1, b)$ si $T[(a + b)/2] < 0$, et $(a, (a + b)/2)$ sinon

Si x appartient à B, on a évidemment $F(x) = f(x)$, puisque le rang r élément de $[\![a, b]\!]$ avec $a = b$ vaut a et vérifie $T[a] = 0$ ou ($r = a$ et $T[a] > 0$) ou ($r = b$ et $T[b] < 0$).

Sinon, vérifions que $F(x) = F(g(x))$. On procède par récurrence sur le nombre n d'itérations pour que $g^n(x)$ appartienne à B. On suppose que $r = F(g(x))$ satisfait les prédicats voulus et on montre que ce même r satisfait les prédicats attendus pour $F(x)$.
❑ Si $T[(a + b)/2] < 0$, alors, selon l'hypothèse de récurrence, $F((a + b)/2 + 1, b)$ donnera un rang r tel que :

$(a + b)/2 + 1 \leq r \leq b$ et
$T[r] = 0$ ou ($r = (a + b)/2 + 1$ et $T[(a + b)/2 + 1] > 0$) ou ($r = b$ et $T[b] < 0$)
ou ($r < b$ et $T[r] < 0$ et $T[r + 1] > 0$) ou ($r > (a + b)/2 + 1$ et $T[r - 1] < 0$ et $T[r] > 0$)

Tous les cas satisfont les conditions de $F(x)$, sauf peut-être le cas $r = (a + b)/2 + 1$, mais alors on a $r > a$ et $T[r - 1] < 0$ et $T[r] > 0$, ce qui convient également.

❑ Si $T[(a + b)/2] \geq 0$, alors, selon l'hypothèse de récurrence, $F(a, (a + b)/2)$ donnera un rang r tel que :

$a \leq r \leq (a + b)/2$ et
$T[r] = 0$ ou ($r = a$ et $T[a] > 0$) ou ($r = (a + b)/2$ et $T[(a + b)/2] < 0$)
ou ($r < (a + b)/2$ et $T[r] < 0$ et $T[r + 1] > 0$) ou ($r > a$ et $T[r - 1] < 0$ et $T[r + 1] > 0$)

Tous les cas satisfont les conditions de $F(x)$, sauf peut-être le cas $r = (a + b)/2$, mais ce cas est impossible puisqu'il signifierait à la foit $T[r] \geq 0$ et $T[r] < 0$.

Dans le cas où x n'appartient pas à B, on a donc bien $F(x) = F(g(x))$. Enfin la récursivité se termine bien car, si x n'appartient pas à B, la longueur de l'intervalle que définit $g(x)$ est strictement inférieure à celui que définit x. En effet :

$b - ((a + b)/2 + 1) < b - (a + b)/2 < b - a$

et $(a + b)/2 - a < b - a$

La conversion de la définition récursive $F(x) = \begin{cases} f(x) \text{ si } x \in B \\ F(g(x)) \text{ sinon} \end{cases}$ donne l'algorithme itératif suivant :

```
tant que b ≠ a faire
        si T[(a + b)/2] < 0 alors  a ← (a + b)/2 + 1
                                sinon  b  ← (a + b)/2
        finsi
finfaire
# Le résultat final est a
```

b) Variante I

Une variante un peu plus compliquée consiste à considérer deux ensembles E et E', une partie B non vide de E, une fonction f : B → E', une fonction g : E \ B → E et une fonction h : E' → E'. Définissons la fonction récursive F : E → E' par :

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

En procédant comme dans le a), il n'est pas très difficile de montrer par récurrence sur n que, si n est le plus petit entier tel que $g^n(x)$ soit élément de B, alors $F(x) = h^n(f(g^n(x)))$, ce qui donne l'algorithme itératif suivant pour calculer $F(x)$:

```
n ← 0
tant que x n'est pas dans B faire
        # invariant de boucle : si x0 est la valeur initiale de x, alors
        # la valeur courante de x est x = g^n(x0)
        x ← g(x)
        n ← n+1
finfaire
# on termine la boucle au premier n tel que g^n(x0) appartienne à B.
y ← f(x)
# y = f(g^n(x0)) avec n plus petit entier tel que g^n(x0) appartienne à B.
pour i variant de 1 à n faire
        y ← h(y)
finfaire
# y = h^n(f(g^n(x0))) avec n plus petit entier tel que g^n(x0)
# appartienne à B. Donc y = F(x0).
```

Mais nous pouvons également voir que nous sommes dans un cas particulier d'une fonction récursive terminale Ψ : E × $\mathbb{N}$ → E'. Pour tout x de E et tout entier naturel m, posons :

$\Psi(x, m) = h^{n+m}(f(g^n(x))$

où n est le plus petit entier tel que $g^n(x)$ appartienne à B. En particulier, on a :

$F(x) = h^n(f(g^n(x)) = \Psi(x, 0)$

Ψ peut être définie récursivement comme suit :

si $x \in B$, $\Psi(x, m) = h^m(f(x))$ définition de base
sinon, $\Psi(x, m) = \Psi(g(x), m + 1)$ définition récursive

En effet, dans le second cas, si n est le plus petit entier tel que $g^n(x)$ appartienne à B, alors $n - 1$ est le plus petit entier tel que $g^{n-1}(g(x))$ appartienne à B et l'on a aussi bien :

$\Psi(x, m) = h^{n+m}(f(g^n(x))$

que $\Psi(g(x), m + 1) = h^{(n-1)+(m+1)}(f(g^{n-1}(g(x))) = h^{n+m}(f(g^n(x)) = \Psi(x, m)$

On est alors dans le cas d'une récursion terminale en prenant :

$\{(x, m), x \in B, m \in \mathbb{N}\}$ ensemble des cas de base de la fonction Ψ

$\psi(x, m) = h^m(f(x))$ définition de base de la fonction Ψ

$\gamma(x, m) = (g(x), m + 1)$

$$\Psi(x, m) = \begin{cases} \psi(x, m) \text{ si } x \in B \\ \Psi(\gamma(x, m)) \text{ sinon} \end{cases}$$

La programmation itérative de Ψ donnée par le a) est :

```
tant que x n'est pas dans B faire
        (x, m) ← gamma(x, m)
finfaire
retourner(psi(x,m));
```

ou encore :

```
# on part avec x et m comme paramètres. Si on initialise m à 0, on calcule F(x).
tant que x n'est pas dans B faire
        x ← g(x)
        m ← m+1
finfaire
y ← f(x)
pour i variant de 1 à m faire
        y ← h(y)
finfaire
# le résultat cherché est dans y.
```

EXEMPLE : *LA LONGUEUR DE LA SUITE DE COLLATZ*

❑ Vérifions que la transformation récursif-itératif donne bien l'algorithme itératif attendu dans le cas de la longueur de la suite de Collatz :

$C(1) = 0$ 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

On a ici :

$B = \{1\}$

$f(1) = 0$

$$g(n) = \begin{cases} \frac{n}{2} \text{ si } n \text{ est pair} \\ 3n + 1 \text{ si } n \text{ est impair} \end{cases}$$

$h(y) = y + 1$

de sorte que :

si $x \in B$, $C(x) = f(x)$

sinon, C(x) = h(C(g(x)))

La traduction itérative de la définition récursive décrite dans ce paragraphe donne :

```
n ← 0
tant que x ≠ 1 faire
        si x est pair alors
                x ← x/2
        sinon
                x ← 3x + 1
        finsi
        n ← n+1
finfaire
y ← 0
pour i variant de 1 à n faire
        y ← y + 1
finfaire
```

qu'on peut abréger en :

```
n ← 0
tant que x ≠ 1 faire
        si x est pair alors
                x ← x/2
        sinon
                x ← 3x + 1
        finsi
        n ← n+1
finfaire
# le résultat final est dans n
```

EXEMPLE : *PROGRAMMATION ITERATIVE DE LA SUITE DE FIBONACCI*

❑ Vérifions que nous pouvons retrouver la programmation itérative de la suite de Fibonacci à partir de sa définition récursive naïve. Prenons E = $\mathbb{N}^*$, E' = $\mathbb{N}^2$. Nous notons ici Fib(n) le terme général de la suite de Fibonacci. Prenons :

B = {1}
f(1) = (1, 0)
$\forall\ n \geq 1$, F(n) = (Fib(n), Fib(n – 1))
g(n) = n – 1
h(x, y) = (x + y,x)

La fonction F vérifie bien F(n) = $\begin{cases} f(n) \text{ si } n \in B \\ h(F(g(n))) \text{ sinon} \end{cases}$.

En effet :

Si n = 1 alors F(n) = (Fib(1), Fib(0)) = (1, 0) = f(n)
sinon :
h(F(g(n))) = h(F(n – 1)) = h(Fib(n – 1), Fib(n – 2))
= (Fib(n – 1) + Fib(n – 2), Fib(n – 1))
= (Fib(n), Fib(n – 1))
= F(n)

L'algorithme itératif issu de la description récursive est (en prenant $n \geq 1$ comme paramètre, et k comme variable locale incrémentée dans la boucle) :

```
k ← 0
tant que n ≠ 1 faire
        n ← n-1
        k ← k+1
finfaire
(y, z) ← (1, 0)
pour i variant de 1 à k faire
        (y, z) ← (y + z, y)
finfaire
# La valeur finale de (y, z) est (Fib(n), Fib(n-1))
```

Mais, comme on le voit, la première boucle est inutile puisque, $k + n$ gardant une valeur constante au cours de cette boucle, la valeur finale de k sera égale à la valeur initiale de n diminuée de 1. L'algorithme se réduit donc à :

```
(y, z) ← (1, 0)
pour i variant de 1 à n-1 faire
        (y, z) ← (y + z, y)
finfaire
```

Il suffit alors de décrire plus en détail l'affectation de (y, z) :

```
z ← 0
y ← 1
Pour i de 1 à n-1 faire
        temp ← y
         y ← y+z
        z ← y
finfaire
# La valeur de Fib(n) se trouve dans y
```

et l'on reconnaît l'algorithme itératif usuel de la suite de Fibonacci.

c) Variante II

Considérons maintenant deux ensembles E et E', une partie non vide B de E, une fonction $f : B \to E$, une fonction $g : E \setminus B \to E$, une fonction $h : E \to E'$ et une loi interne sur E', notée $*$, que nous supposerons commutative et associative pour ne pas abuser de parenthèses, et possédant un élément neutre noté e. Soit $F : E \to E'$ définie récursivement par :

si $x \in B$, $F(x) = f(x)$ définition de base sinon, $F(x) = h(x) * F(g(x))$

La fonction F est un cas particulier d'une fonction $\Psi : E \times E' \to E'$ récursive terminale définie comme suit :

L'ensemble des cas de base de Ψ est $B \times E' = \{(x, a), x \in B, a \in E'\}$

$\psi(x, a) = a * f(x)$

$\gamma(x, a) = (g(x), a * h(x))$

$$\Psi(x, a) = \begin{bmatrix} a * f(x) \text{ si } x \in B \\ \Psi(g(x), a * h(x)) \end{bmatrix} = \begin{bmatrix} \psi(x, a) \text{ si } x \in B \\ \Psi(\gamma(x, a)) \text{ sinon} \end{bmatrix}$$

Vérifions par récurrence sur n, plus petit entier tel que $g^n(x)$ appartienne à B, que $\Psi(x, a) = a * F(x)$.
Si $n = 0$, x est élément de B et $\Psi(x, a) = a * f(x) = a * F(x)$.
Supposons la relation vérifiée au rang $n - 1$ et vérifions-la au rang n. Si x est tel que n est le plus petit entier tel que $g^n(x)$ appartienne à B, alors $n - 1$ est le plus petit entier tel que $g^{n-1}(g(x))$ appartienne à B, donc on peut appliquer l'hypothèse de récurrence au rang $n - 1$ sur $g(x)$. On a :

$\Psi(x, a) = \Psi(\gamma(x, a))$
$= \Psi(g(x), a * h(x))$
$= a * h(x) * F(g(x))$ d'après l'hypothèse de récurrence au rang $n - 1$
$= a * F(x)$ par définition de F

et la relation est vérifiée au rang n.

En particulier, $F(x) = \Psi(x, e)$.

Ψ est une fonction définie par une récursivité teminale, et la traduction récursif-itératif du a) appliquée à $\Psi(x, a)$ donne :

```
# Initialiser a par la valeur e si on souhaite calculer F(x)
tant que x n'est pas dans B faire
        (x, a) ← gamma(x, a)
finfaire
retourner(psi(x, a));
```

ou encore :

```
# Initialiser a par la valeur e si on souhaite calculer F(x)
tant que x n'est pas dans B faire
        a ← a * h(x)
        x ← g(x)
finfaire
retourner(a * f(x));
```

La variable a s'appelle un **accumulateur**.

On peut rajouter des commentaires prouvant directement la validité de l'algorithme, indépendamment de ce qui précède :

```
# Initialiser a par la valeur e si on souhaite calculer F(x)
tant que x n'est pas dans B faire
        # Invariant de boucle : si on a fait k boucles, et si (x0, a0) est la valeur initiale du
        # couple (x, a), on a
        # a = a0 * h(x0) * h(g(x0)) * ... * h(g^(k-1)(x0))
        # x = g^k(x0)
        a ← a * h(x)
        # a = a0 * h(x0) * h(g(x0)) * ... * h(g^(k-1)(x0)) * h(g^k(x0))
        x ← g(x)
        # x = g^(k+1)(x0)
        # On a terminé une boucle, on incrémente k de 1.
finfaire
# On quitte la boucle si x appartient à B. Le nombre de boucles est le plus petit n tel que g^n(x0)
# appartienne à B. On a :
```

```
# a = a0 * h(x0) * h(g(x0)) * ... * h(g^(n-1)(x0))
# x = g^n(x0)
retourner(a * f(x));
# On a calculé a0 * h(x0) * h(g(x0)) * ... * h(g^(n-1)(x0)) * f(g^n(x0))
```

Vérifions en effet que $\Psi(x, a) = a * \mathop{*}_{k=0}^{n-1} h(g^k(x)) * f(g^n(x))$, où n est le plus petit entier tel que $g^n(x)$ soit élément de B. Raisonnons par récurrence sur n.

Pour $n = 0$, la quantité $\mathop{*}_{k=0}^{n-1} h(g^k(x))$ est vide, donc égale à e, donc :

$$a * \mathop{*}_{k=0}^{n-1} h(g^k(x)) * f(g^n(x)) = a * f(x) = \psi(x, a) = \Psi(x, a)$$

Supposons la relation vraie au rang $n - 1$ et vérifions-la au rang n. Posons $y = g(x)$. $n - 1$ est le plus petit entier tel que $g^{n-1}(y)$ appartienne à B

$$a * \mathop{*}_{k=0}^{n-1} h(g^k(x)) * f(g^n(x)) = a * h(x) * \mathop{*}_{k=1}^{n-1} h(g^{k-1}(y)) * f(g^{n-1}(y))$$

$$= a * h(x) * \mathop{*}_{k=0}^{n-2} h(g^k(y)) * f(g^{n-1}(y))$$

en changeant d'indice dans $\mathop{*}_{k=1}^{n-1}$

$$= \Psi(y, a * h(x)) \quad \text{d'après l'hypothèse de récurrence}$$

$$= \Psi(g(x), a * h(x))$$

$$= \Psi(x, a) \quad \text{d'après la définition récursive de } \Psi$$

En particulier, $F(x) = \mathop{*}_{k=0}^{n-1} h(g^k(x)) * f(g^n(x))$, où n est le plus petit entier tel que $g^n(x)$ soit élément de B.

EXEMPLE : *LA FACTORIELLE*

❑ Considérons la factorielle définie récursivement par :

$F(0) = 1$

$\forall\ n \geq 1, F(n) = n\ F(n - 1)$

On a ici :

la loi $*$ est le produit usuel

$e = 1$

$B = \{0\}$

$f(0) = 1$

$\forall\ n \geq 1, g(n) = n - 1$

$\forall\ n \geq 1, h(n) = n$

L'algorithme itératif décrit précédemment s'écrit :

```
a ← 1
tant que n ≠ 0 faire
        a ← a * n
        n ← n-1
finfaire
retourner(a);
```

a stocke par ordre décroissant le produit des entiers, depuis *n* jusqu'à 1, et donne bien comme valeur finale la factorielle désirée.

EXEMPLE : *L'EXPONENTIATION RAPIDE*

❑ Soient *z* et *n* des nombres entiers. Ci-dessous, *n*/2 désigne le quotient de la division euclidienne de *n* par 2. On peut définir le nombre x^n récursivement par :

$z^n = 1$	si $n = 0$	définition de base
$z^n = (z^2)^{n/2}$	si *n* est pair	
$z^n = z \times (z^2)^{(n-1)/2}$	si *n* est impair	

Posons donc :

$*$ le produit usuel

$e = 1$

$x = (z, n)$

$F(x) = z^n$

$B = \{(z, 0)\}$

$f(x) = 1$

$g(x) = (z^2, n/2)$on rappelle que *n*/2 désigne ici le quotient entier,

égal à $\frac{n-1}{2}$ si *n* est impair.

$$h(x) = \begin{cases} 1 \text{ si } n \text{ est pair} \\ z \text{ si } n \text{ est impair} \end{cases}$$

F est bien défini récursivement comme suit :

$$F(x) = \begin{cases} f(x) \text{ si } x \in B \\ h(x) * F(g(x)) \text{ sinon} \end{cases}$$

La traduction itérative donne :

```
a ← 1
tant que n ≠ 0 faire
        a ← a * h(z, n)
        (z, n) ← (z², n/2)
finfaire
retourner(a);
```

ou encore, en détaillant davantage :

```
a ← 1
tant que n ≠ 0 faire
        si n est impair alors a ← a * z finsi
        z ← z²
        n ← n/2          # division entière
finfaire
retourner(a);
```

On reconnaît exactement l'algorithme donné dans le cours de première année L1/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 : *CALCUL RAPIDE DES 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, on vérifiera par récurrence sur n que :

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

Le calcul de F_n peut donc être obtenu par celui de $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$, et l'algorithme d'exponentiation rapide appliqué sur la puissance de matrice 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'algorithme itératif donné plus haut, et pire encore, $O((\frac{1+\sqrt{5}}{2})^n)$ pour la procédure récursive naïve initiale.

Avec les notations de l'exemple précédent, 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 $A^0 = I_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ 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, sous une forme itérative :

```
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 + a*q
                          q ← a*temp + b*q
        finsi
        temp ← a                            # calcul de A ← A²
        a ← a² + 2*a*b
        b ← temp² + b²
        n ← n/2                             # division entière de n par 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
        # fin if
        temp=a
        a=a**2 + 2*a*b
        b=temp**2 + b**2
        n=n//2
    # fin while
    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())
```

d) Recursivité non terminale

On donne ici un exemple de récursivité non terminale, qu'on transformera en un algorithme itératif muni d'une pile. Soient E et E' deux ensembles, B une partie non vide de E, et des fonctions $f : E \to E'$, $g : E \setminus B \to E$, $h : E \times E' \to E'$. On définit récursivement F par :

$$F(x) = \begin{cases} f(x) \text{ si } x \in B \\ h(x, F(g(x))) \text{ sinon} \end{cases}$$

Si n est le plus petit entier tel que $g^n(x)$ appartienne à B, alors :

$$F(x) = h(x, h(g(x), h(g^2(x), ...(h(g^{n-1}(x), f(g^n(x)))))...))$$

avec n fois la lettre h, comme on le vérifie par récurrence sur n.

Pour dérécursifier, on peut mémoriser les éléments x, $g(x)$, ..., $g^n(x)$ en les stockant successivement dans une pile. L'algorithme itératif est le suivant, la pile étant représentée par le symbole L :

```
fonction F(x)
    L ← [ ]
    tantque (x n'est pas dans B) faire
        # Invariant de boucle. Notons x0 la valeur initiale de x.
        # Si on a effectué k boucles, alors L contient les
        # éléments [x0, g(x0), g^2(x0), ..., g^(k-1)(x0)] et x = g^k(x0)
        L.empile(x)
        # L contient les éléments [x0, g(x0), g^2(x0), ..., g^(k-1)(x0), g^k(x0)]
        x ← g(x)
        # x = g^(k+1)(x0).
        # On a terminé une boucle, on incrémente k de 1.
    finfaire
    # Si n est le plus petit entier tel que g^n(x0) soit élément de B, alors
    # x = g^n(x0) et L = [x0, g(x0), g^2(x0), ..., g^(n-1)(x0)].
    y ← f(x);
    # y = f(g^n(x0))
    tant que L ≠ [ ] faire
```

```
                # Invariant de boucle : si on a effectué k boucles, alors
                # L contient [x0, g(x0), ..., g^{n-1-k}(x0)] et
                # y = h(g^{n-k}(x0), ...(h(g^{n-1}(x0), f(g^n(x0)))))...))
                y ← h(L.depile(), y)
                # On dépile l'élément g^{n-1-k}(x0). L contient [x0, g(x0), ..., g^{n-2-k}(x0)] et
                # y = h(g^{n-1-k}(x0), h(g^{n-k}(x0), ...(h(g^{n-1}(x0), f(g^n(x0)))))...))
                # On a terminé une boucle. k est incrémenté de 1
        finfaire
        # On quitte la boucle quand L est vide, donc quand k=n.
        # donc y = h(x0, h(g(x0), ...(h(g^{n-1}(x0), f(g^n(x0)))))...)) = F(x0)
        retourner(y)
```

La complexité en temps de calcul est un O(n) et celle en espace est O(n) (correspondant à la place en mémoire occupée par la pile).
On peut se dispenser de la pile ou de tout tableau de stockage en calculant d'abord le plus petit n tel que $g^n(x)$ appartienne à B, puis en effectuant une boucle pour k variant de 1 à n qui change la valeur de y, mais cela nécessite pour chaque k la connaissance de $g^{n-1-k}(x)$, ce qui demande un temps de calcul en O($n - 1 - k$) faute d'avoir mémorisé cette valeur. L'algorithme obtenu aura alors une complexité en temps de calcul en O(n^2) et une complexité en espace en O(1).

EXEMPLE :
La méthode utilisée ici doit pouvoir redonner les cas a), b), c) avec un moyen de supprimer la pile.
❑ a) On prend $h(u, v) = v$. L'algorithme avec pile s'écrit alors :

```
fonction F(x)
        L ← [ ]
        tantque (x n'est pas dans B) faire
                L.empile(x)
                x ← g(x)
        finfaire
        y ← f(x);
        tant que L ≠ [ ] faire
                y ← y
        finfaire
        retourner(y)
```

et l'on voit que la deuxième boucle est inutile, et donc que L est également inutile, d'où :

```
fonction F(x)
        tantque (x n'est pas dans B) faire
                x ← g(x)
        finfaire
        y ← f(x);
        retourner(y)
```

qui est bien l'algorithme itératif du a).

❑ b) On prend $h(u, v) = h(v)$, ce qui donne :

```
fonction F(x)
        L ← [ ]
        tantque (x n'est pas dans B) faire
                L.empile(x)
                x ← g(x)
        finfaire
        y ← f(x);
        tant que L ≠ [ ] faire
                y ← h(y)
```

```
        finfaire
        retourner(y)
```

On voit que les éléments contenus dans L n'interviennent pas. Seul compte la hauteur de L qui indique combien de fois on doit itérer la deuxième boucle. On peut donc se borner à utiliser une variable entière qui stocke cette hauteur :

```
fonction F(x)
        m ← 0
        tantque (x n'est pas dans B) faire
                m ← m+1
                x ← g(x)
        finfaire
        y ← f(x);
        pour i de 1 à m faire
                y ← h(y)
        finfaire
        retourner(y)
```

et l'on retrouve l'algorithme itératif du b).

❑ c) Prenons $h(u, v)$ sous la forme $h(u) * v$. On a alors :

```
fonction F(x)
        L ← [ ]
        tantque (x n'est pas dans B) faire
                L.empile(x)
                x ← g(x)
        finfaire
        y ← f(x);
        tant que L ≠ [ ] faire
                y ← h(L.depile()) * y
        finfaire
        retourner(y)
```

On voit que, L stockant les valeurs $g^k(x)$, $0 \leq k < n$ où n est le plus petit entier tel que $g^n(x)$ appartienne à B, la variable y stocke le produit $*$ de $f(g^n(x))$ et des $h(g^k(x))$. On peut donc utiliser un accumulateur qui calcule le produit des $h(g^k(x))$ dès la première boucle, permettant de retrouver l'algorithme itératif tel que nous l'avons vu auparavant.

III : Tri de tableaux

On cherche à trier un tableau par ordre croissant. Les éléments du tableau appartiennent à un ensemble muni d'un ordre total noté ≤ (i.e. on peut comparer entre eux deux éléments quelconque de cet ensemble). 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
```

Dans la boucle, la variable m est en permanence égale à T[j]. On pourrait donc s'en passer. Nous l'avons laissée 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 croît comme le carré de la longueur du tableau et 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=T.size
    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
            # fin if
        # fin for
        T[j]=T[i]
        T[i]=m
    # fin for
```

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
        # fin while
    # fin for
```

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 on est 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 pour insérer la valeur T[i] dans le tableau trié [T[0], ..., T[i–1]], 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'affectations 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[n/2 – 1]] et [T[n/2],...,T[n – 1]], où n/2 désigne ici le quotient entier de n par 2. 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). La fusion ici décrite est analogue à la fonction `FusionIterative` décrite plus haut, que nous appellerons ici simplement `Fusion` :

```
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))
    # fin if
```

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, chacun des deux sous-tableaux possède $\frac{n}{2}$ éléments et le temps de calcul pour trier chacun de ces deux sous-tableaux est $t(\frac{n}{2})$. En outre, la fusion finale demande un nombre d'opérations proportionnel à n puisqu'on remplit le tableau final trié élément par élément. On a donc une relation du type :

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

Supposons que $n = 2^p$ et posons $u(p) = t(2^p)$. 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 :

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

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 ou égaux 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 ici de se donner une valeur pivot *x*, puis de déplacer les éléments inférieurs ou égaux à *x* vers les indices les plus faibles, et les éléments strictement supérieurs à *x* vers les indices les plus élevés. Puis on trie récursivement les deux sous-tableaux. 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 a un coût en temps de calcul. On se bornera à prendre comme pivot la dernière valeur du tableau, ce qui est légitime si le tableau est initialement rangé dans un ordre aléatoire. S'il est partiellement trié, il vaudrait mieux prendre comme pivot l'élément situé au centre du tableau (qui ne devrait pas être trop éloigné de la valeur médiane). Il suffira alors de modifier l'algorithme en permutant préalablement ce pivot avec le dernier élément du tableau.

Voici le programme, en Python, de paramètres le tableau T et deux indices *p* et *q*. Le programme 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 tri_rapide(T,p,q):
    # Si p ≥ q, on ne traite pas le tableau.
    if p<q:
        x=T[q]
        i=p
        j=q-1
        while i<j:
        # Invariant de boucle :
        # T[p],...,T[i-1] ≤ x < T[j+1],...,T[q-1].
        # Initialement, le membre de gauche et celui
        # de droite sont vides.
            if T[i]>x:
                # On permute T[i] et T[j]
                temp=T[i]
                T[i]=T[j]
                T[j]=temp
                # On a :
                # T[p],...,T[i-1] ≤ x < T[j],...,T[q-1].
                j=j-1
                # On a :
                # T[p],...,T[i-1] ≤ x < T[j+1],...,T[q-1].
            else:
                # On a ici T[i] ≤ x.
                i=i+1
                # On a :
                # T[p],...,T[i-1] ≤ x < T[j+1],...,T[q-1].
            # fin if
            # Dans tous les cas :
```

```
            # T[p],...,T[i-1] ≤ x < T[j+1],...,T[q-1].
        # fin while
        # On sort de la boucle avec i=j.
        # On termine en traitant l'élément T[i].
        # Si T[i]>x, on a :
        # T[p],...,T[i-1] ≤ x < T[i],T[i+1]=T[j+1],...,T[q-1].
        # Sinon, on exécute l'instruction suivante :
        if T[i]<=x:
            i=i+1
            # On a i=j+1 et :
            # T[p],...,T[i-1] ≤ x < T[j+1]=T[i],...,T[q-1]
        # fin if
        # Dans tous les cas :
        # T[p],...,T[i-1] ≤ x < T[j+1]=T[i],...,T[q-1]
        # On déplace le pivot T[q] = x
        # en l'échangeant avec T[i].
        temp=T[i]
        T[i]=T[q]
        T[q]=temp
        # On a :
        # T[p],...,T[i-1] ≤ T[i] < T[i+1],...,T[q]
        tri_rapide(T,p,i-1)
        tri_rapide(T,i+1,q)
    # fin if
```

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

L'algorithme se termine nécessairement puisque chaque appel récursif se fait sur des tableaux de taille strictement inférieure.

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 vide, l'autre de taille n – 1 par exemple. Si cette répartition malheureuse se reproduit récursivement trop souvent (c'est le cas si le tableau est intialement trié par ordre strictement décroissant), 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 moyen 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, qu'on supposera distincts pour

simplifier le raisonnement, la probabilité de choisir un tel pivot est $\frac{1}{n+1}$. Le nombre moyen de comparaisons à faire est alors, en appliquant le théorème de l'espérance totale (voir le chapitre L2/PROBA2.PDF) :

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

Le premier n dans la somme de droite correspond aux n comparaisons que l'on fera entre les éléments n premiers éléments du tableau avec le pivot x 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 à 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.
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 $\sum_{k=1}^{n} \frac{1}{k} \sim \ln(n)$ (voir le chapitre L2/SERIES.PDF). Donc :

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

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

Annexe : la résolution du problème des partis par Pascal

Une des premières apparitions explicites de la notion de récursivité date de la résolution par Blaise Pascal d'un problème datant du XIVème, le problème des "partis" (sic). Ce problème fut en effet à nouveau posé par le Chevalier de Méré à Pascal en 1652, et donna lieu à une correspondance entre Pascal et Fermat en 1654. On y voit apparaître également la notion implicite d'espérance mathématique.

L'auteur anonyme du XIVème considère deux joueurs d'échecs (qui est considéré curieusement comme un jeu de hasard équiprobable). Le premier joueur qui gagne trois parties remporte deux ducats. Alors que l'un des joueurs a gagné deux parties et l'autre aucune, le jeu s'interrompt et on demande comment faire le partage des deux ducats. Nous passons sur les diverses solutions proposées pendant trois siècles pour arriver à Pascal (pour qui la mise de chaque joueur est de 32 pistoles) :

> *Voici à peu près comment je fais pour savoir la valeur de chacune des parties, quand deux joueurs jouent par exemple, en trois parties, et chacun a mis 32 pistoles au jeu [c'est-à-dire que le premier qui a gagné trois parties remporte toute la mise, soit 64 pistoles] :*
> *Posons que le premier ait deux et l'autre une ; ils jouent maintenant une partie, dont le sort est tel que, si le premier la gagne, il gagne tout l'argent qui est au jeu, savoir 64 pistoles ; et si l'autre la gagne, ils sont deux parties à deux parties, et par conséquent, s'ils veulent se séparer, il faut qu'ils retirent chacun leur mise, savoir 32 pistoles.*
> *Considérez donc, Monsieur, que si le premier gagne, il lui appartient 64 ; s'il perd, il lui appartient 32. Donc, s'ils veulent ne point hasarder cette partie et se séparer sans la jouer, le premier doit dire : "je suis sûr d'avoir 32 pistoles, car la perte même me les donne ; mais pour les 32 autres, peut-être je les aurai, peut-être vous les aurez ; le hasard est égal ; partageons donc ces 32 pistoles par la moitié et me donnez, outre cela, mes 32 qui me sont sûres." Il aura donc 48 pistoles et l'autre 16.*

Si nous notons G(*a*, *b*) la somme que doit recevoir le joueur qui a gagné *a* parties et en a perdu *b*, Pascal énonce que G(3, 1) = 64, G(2, 2) = 32 et :

$$G(2, 1) = G(2, 2) + \frac{G(3, 1) - G(2, 2)}{2} = 48 = \frac{G(3, 1) + G(2, 2)}{2}$$

Passons au cas suivant :

> *Posons maintenant que le premier ait deux parties et l'autre point, et ils commencent à jouer une partie. Le sort de cette partie est tel que, si le premier la gagne, il tire tout l'argent, 64 pistoles : si l'autre la gagne, les voilà revenus au cas précédent, auquel le premier aura deux parties et l'autre une.*
> *Or, nous avons déjà montré qu'en ce cas il appartient à celui qui a les deux parties 48 pistoles : donc, s'ils veulent ne point jouer cette partie, il doit dire ainsi : "Si je la gagne, je gagnerai tout, qui est 64 ; si je la perds, il m'appartiendra légitimement 48 : donc donnez-moi les 48 qui me sont certaines au cas même que je perde et partageons les 16 autres par la moitié, puisqu'il y a autant de hasard que vous les gagniez comme moi." Ainsi, il aura 48 et 8, qui font 56 pistoles.*

On a maintenant, avec G(2, 1) = 48 ≤ 64 = G(3,0) :

$$G(2, 0) = G(2, 1) + \frac{G(3, 0) - G(2, 1)}{2} = 56 = \frac{G(3, 0) + G(2, 1)}{2}$$

donnant ainsi la solution au problème historique. Pascal poursuit le raisonnement en envisageant un troisième cas :

> *Posons enfin que le premier n'ait qu'une partie et l'autre point. Vous voyez, Monsieur, que, s'ils commencent une partie nouvelle, le sort en est tel que, si le premier la gagne, il aura deux parties à point, et partant, par le cas précédent, il lui appartient 56 ; s'il la perd, ils sont partie à partie : donc il lui appartient 32 pistoles. Donc il doit dire : "Si vous voulez ne la pas jouer, donnez-moi 32 pistoles qui me sont sûres, et partageons le reste des 56 par la moitié. De 56 ôtez 32, reste 24 ; partagez donc 24 par la moitié, prenez-en 12, et moi 12, qui avec 32, font 44."*

Ainsi, avec $G(1, 1) = 32 \leq 56 = G(2, 0)$:

$$G(1, 0) = G(1, 1) + \frac{G(2, 0) - G(1, 1)}{2} = 44 = \frac{G(2, 0) + G(1, 1)}{2}$$

Pascal n'est pas le premier a tenir ce type de raisonnement récursif. La démarche suivie figure déjà dans un texte anonyme du XVème, texte cependant resté confidentiel, les découvertes mathématiques relevant encore beaucoup du secret à l'époque, contrairement à ce qui se passe au XVIIème.

Il ne fait guère de doute que Pascal serait capable de généraliser le problème en exigeant que le vainqueur gagne n parties au lieu de 3 pour emporter la totalité M des mises. On a alors :

$\forall\ b < n,\ G(n, b) = M$	cas de base
$\forall\ a < n,\ G(a, n) = 0$	autre cas de base
$\forall\ a,\ G(a, a) = \frac{M}{2}$	autre cas de base
$\forall\ a < n, \forall\ b < n,\ G(a, b) = \frac{G(a + 1, b) + G(a, b + 1)}{2}$	définition récursive

Le lecteur pourra s'amuser à programmer la fonction G.

Exercices

1- Enoncés

Exo.1) Soit (F(n)) la suite de Fibonacci.

a) Prouver que, pour tout n et tout m, $F(n + m) = F(n + 1)F(m) + F(n)F(m - 1)$ (*)

b) On considère l'algorithme suivant de calcul rapide de F(n) donné dans le cours, n étant le paramètre, le résultat final étant p :

```
a ← 1
b ← 0
p ← 0
q ← 1
tant que n ≠ 0 faire
        si n impair alors temp ← p
                          p ← (a + b)*p + a*q
                          q ← a*temp + b*q
        finsi
        temp ← a
        a ← a^2 + 2*a*b
        b ← temp^2 + b^2
        n ← n/2
finfaire
```

Notons n_0 la valeur initiale de n. Montrer directement la validité de cet algorithme en vérifiant qu'un invariant de boucle est :

$$\exists\, k, \exists\, r, n_0 = 2^k\, n + r, 0 \le r < 2^k, a = F(2^k), b = F(2^k - 1), p = F(r), q = F(r - 1)$$

Exo.2) a) Ecrire une procédure inv de paramètre n entier, et dont le résultat est obtenu en écrivant les chiffres décimaux de n en sens inverse. Par exemple, inv(291) a pour résultat l'entier 192.
b) On définit la **hauteur palindromique** $h(n)$ d'un nombre n de la façon suivante :

$$h(n) = \begin{cases} 0 \text{ si } n = \text{inv}(n) \\ 1 + h(n + \text{inv}(n)) \text{ sinon} \end{cases}$$

Ecrire une procédure non récursive de paramètre n entier qui calcule sa hauteur palindromique.
On ignore actuellement si la fonction h est définie pour tout n. Tester en particulier n = 89, 196, 9008299. Le record actuel du nombre de plus haute hauteur palindromique est 11860603078919299 90. Pour 196, en 2011, on a itéré un milliard de fois pour atteindre un nombre de 400 millions de chiffres, sans succés.

Exo.3) Dans chacun des cas a), b), c) suivant, on considère une fonction F définie sur $\mathbb{N}^*$. p désigne un entier supérieur ou égal à 1 :

a) $\begin{cases} F(1) = 1 \\ F(2p) = F(p) \\ F(2p + 1) = 1 + F(p) \end{cases}$ b) $\begin{cases} F(1) = 0 \\ F(2p) = F(p) + 1 \\ F(2p + 1) = F(p) \end{cases}$ c) $\begin{cases} F(1) = 0 \\ F(2p) = F(p) + 1 \\ F(2p + 1) = 0 \end{cases}$

Que calculent ces fonctions, en relation avec la décomposition binaire des entiers ?
Dans chacun des cas, écrire un algorithme itératif calculant F(n).

Exo.4) : un affichage récursif. On considère la procédure récursive suivante, de paramètre un entier n (une procédure est une suite d'instructions exécutant une tâche, mais qui ne renvoie aucune valeur particulière. Les tâches sont ici des affichages à l'écran) :

```
procedure F(n)
        # si n<0, on ne fait rien
        si n >=0 alors
                F(n-2):
                print(n):
                F(n-1):
        finsi
```

Dire ce qu'affichera l'appel F(4) et expliquer pourquoi.

Exo.5) Pour tout entiers n et p, on note $\sigma(n, p)$ le nombre de surjections de $[\![1, n]\!]$ sur $[\![1, p]\!]$. Se reporter à l'annexe I du chapitre L1/DENOMBRE.PDF pour déterminer un algorithme récursif ou itératif permettant de calculer $\sigma(n, p)$. Prouver que l'algorithme choisi se termine et estimer sa complexité en temps de calcul.

Exo.6) Les **nombres de Bernoulli** b_n sont définis de la façon suivante :

$$b_0 = 1$$

$$\forall\, n \ge 1, b_n = -\frac{1}{n+1} \sum_{k=0}^{n-1} \binom{n+1}{k} b_k$$

Ecrire une fonction de paramètre n permettant de calculer b_n.

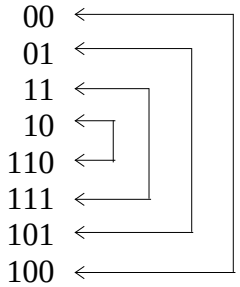
Exo.7) (énoncé de Michel Lafond, paru dans le Bulletin Vert de l'APMEP, n°487 (mars-avril 2010), problème 487-3) On joue au jeu suivant. L'un des partenaires choisit un nombre entier N élément de

[[1, 299]] . L'autre joueur doit trouver N en ne posant que des questions du type : "N est-il strictement plus grand que *x* ?", *x* étant un entier quelconque de son choix. La réponse par le partenaire ne peut être que OUI ou NON. On a droit à un maximum de 12 questions avant de proposer une réponse. Mais on n'a droit qu'à un maximum de 3 réponses NON. Autrement dit, si on a pour la troisième fois une réponse NON, on est obligé de proposer une réponse au coup suivant, et le jeu est alors terminé. Avec ces règles, il est possible de gagner à coup sûr, mais la stratégie est unique. En particulier quelle doit être la première question ?

Exo.8) Le code de gray. Voici ci-dessous deux manières de coder les entiers avec une suite de 0 ou de 1, le code binaire et le code de Gray :

décimal	*binaire*	*Gray*
0	0	0
1	1	1
2	10	11
3	11	10
4	100	110
5	101	111
6	110	101
7	111	100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000
16	10000	11000
17	10001	11001
...	...	...

On peut définir le code de Gray récursivement de la façon suivante : pour tout entier $m \geq 0$ et tout k tel que $0 \leq k < 2^m$, le code de Gray de $2^m + k$ (qui possède $m + 1$ chiffres en binaire) est constitué d'un 1 suivi des m chiffres constituant le code de Gray de $2^m - 1 - k$ (éventuellement complété par des 0 à gauche si ce code possède moins de m chiffres). Cela signifie que la suite des codes de Gray des nombres de 2^m jusqu'à $2^{m+1} - 1$ s'obtient en inversant l'ordre de la suite des codes de Gray des nombres de 0 à $2^m - 1$, et en les faisant précéder d'un 1. Par exemple, pour $m = 2$:



a) Ecrire des fonctions inttogray, graytoint convertissant respectivement un entier en code de Gray, et un code de Gray en entier. Les codes de Gray seront simplement représentés par les nombres décimaux correspondants ; ainsi le code 110 sera désigné au moyen de l'entier décimal cent

dix. Les procédures auront donc toutes comme paramètre un entier (entier constitué uniquement de chiffres décimaux 0 ou 1 pour la fonction graytoint) et leur résultat sera un entier (entier constitué uniquement de chiffres décimaux 0 ou 1 pour la fonction inttogray). Aucun tableau ne doit être utilisé dans ces procédures.

b) Montrer que, si n possède pour code binaire $...b_ib_{i-1}...b_0$ et pour code de Gray $...g_ig_{i-1}...g_0$, alors $b_i \equiv \sum_{k \geq i} g_k \mod 2$, et $g_i \equiv b_i + b_{i+1} \mod 2$. (On peut raisonner par récurrence sur n). En déduire deux procédures bintogray et graytobin de conversion directe d'un code binaire au code de Gray et réciproquement, (également sans utiliser de tableau).

c) Le code de Gray possède la propriété remarquable suivante. Le code de l'entier $n + 1$ diffère du code de l'entier n sur un bit. Montrer cette propriété en utiliser le codage binaire de n et les relations du b) entre code binaire et code de Gray. En déduire une procédure directe successeur de paramètre un code de Gray et dont le résultat est le code de Gray du nombre qui suit. Ainsi successeur(111) donne le résultat 101.

Exo.9) La **suite de Stern** est définie de la façon suivante :

$s(0) = 0 \qquad s(1) = 1$ définition de base

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

Ses premières valeurs sont 0, 1, 1, 2, 1, 3, 2, 3, 1, 4, 3, etc.

La définition récursive relève de divisions par 2, comme pour la suite de Thue. Cela signifie-t-il qu'un algorithme récursif basé sur cette définition aura une complexité en temps de calcul en O(ln(n)), proportionnel au nombre de chiffres de n, comme pour la suite de Thue ? Ou bien cet algorithme est-il notablement ralenti par les deux appels récursifs lorsque n est impair, conduisant à une complexité exponentielle comme pour la programmation récursive naïve de la suite de Fibonacci ? On se propose de répondre à cette question. Le nombre R(n) d'appels récursifs pour calculer $s(n)$ vérifie :

$R(0) = R(1) = 0$

$$\forall\ n \geq 2,\ R(n) = \begin{cases} 1 + R(\frac{n}{2}) \text{ si } n \text{ est pair} \\ 2 + R(\frac{n+1}{2}) + R(\frac{n-1}{2}) \text{ si } n \text{ est impair} \end{cases}$$

En effet, si n est pair, le nombre d'appels récursifs pour calculer $s(n)$ est égal à 1 (appel de $s(\frac{n}{2})$) auquel on ajoute $R(\frac{n}{2})$ (nombre d'appels récursifs pour calculer $s(\frac{n}{2})$). On raisonne de même dans le cas où n est impair. Le lecteur est invité à :

écrire un programme calculant $s(n)$

écrire un programme calculant R(n)

écrire un programme qui calcule $M_n = \text{Max}\ \{R(k),\ 2^n \leq k < 2^{n+1}\}$ et l'indice k pour lequel ce maximum est atteint, et à décomposer ce k en binaire.

Ci-dessous, on donne les valeurs de R(k), $k \geq 1$, rangées ligne par ligne, la n-ème ligne donnant les valeurs R(k), $2^n \leq k < 2^{n+1}$.

$n = 0 \qquad 0$

$n = 1$ 1, 3
$n = 2$ 2, 6, 4, 7
$n = 3$ 3, 10, 7, 12, 5, 13, 8, 12
$n = 4$ 4, 15, 11, 19, 8, 21, 13, 19, 6, 20, 14, 23, 9, 22, 13, 18

Opérant ainsi pour plusieurs valeurs de n, il devrait comprendre d'où sort la fonction $f(n)$ définie dans le b) ci-dessous.

a) Pour tout $n \geq 0$, calculer $R(2^n)$, $R(2^n + 1)$ et $R(2^{n+1} - 1)$.

b) Pour tout $n \geq 1$, on pose $f(n) = \frac{5 \times 2^n + (-1)^n}{3}$. Montrer que $f(n)$ est un entier. Quelle est sa parité ? Combien sa décomposition binaire possède-t-elle de bits ?

Les premières valeurs de $f(n)$ sont 3, 7, 13, 27, 53, etc.

c) Montrer que la suite $(s(f(n))$ est la suite de Fibonacci (à un décalage d'indice près), et qu'il en est de même de la suite $(\frac{R(f(n)) + 3}{2})$. Conclure que $R(n)$ n'est pas un $O(\ln(n))$.

d) Montrer que, pour tout $n \geq 1$, $R(f(n)) = \text{Max } \{R(k), 2^n \leq k < 2^{n+1}\}$

e) Conclure que $R(n)$ est un $O(n^\alpha)$ avec $\alpha = \frac{\ln(\frac{1 + \sqrt{5}}{2})}{\ln(2)} \approx 0{,}694$.

Exo.10) Calcul de π par dénombrement de mots : Soit v_n le nombre de mots de n lettres dans un alphabet de n lettres, chaque lettre n'apparaissant pas plus de deux fois. On convient que $v_0 = 1$. Les premières valeurs de la suite $(v_n)_{n\geq 0}$ sont :

1, 1, 4, 24, 204, 2220, 29520, 463680, 8401680, 172504080, 3958113600

a) Montrer que $v_n = n! \sum_{k\geq 0} \frac{1}{2^k} \binom{2k}{k} \binom{n}{2k}$.

b) Déterminer l'expression de la fonction génératrice $H(x) = \sum_{n=0}^{\infty} \frac{v_n}{n!} x^n$.

c) En déduire que la suite (v_n) vérifie la relation de récurrence :

$$\forall\ n \geq 1,\ v_{n+1} = (2n + 1)v_n + n^2\ v_{n-1}.$$

On pourra chercher à exprimer $H'(x)$ en fonction de $H(x)$.

d) La relation permet un calcul des v_n, soit par un algorithme itératif, soit par un algorithme récursif (à condition, dans ce dernier cas, d'utiliser une méthode de memoïsation, faute de quoi le nombre d'appels récursifs croît exponentiellement avec n). Ecrire un tel algorithme.

Pour tout n, on pose $u_n = \frac{v_n}{n!}$, égal au rapport du nombre de mots de n lettres dans un alphabet de n lettres, chaque lettre n'apparaissant pas plus de deux fois, sur le nombre de mots de n lettres dans un alphabet de n lettres, chaque lettre n'apparaissant pas plus d'une fois. Calculer une valeur approchée de $4 \times \sum_{n=0}^{10} \frac{(-1)^n}{(n + 1)u_n u_{n+1}}$. Conjecturer la valeur de $\sum_{n=0}^{\infty} \frac{(-1)^n}{(n + 1)u_n u_{n+1}}$.

Exo.11) Considérons la suite F définie récursivement sur $\mathbb{N}^*$ par[2] :

[2] Chris Groer, The Mathematics of Survival : From Antiquity to the Playground, 110:9, *Amer. Math. Monthly* (novembre 2003), 812-825.

$F(1) = 1$ définition de base

$\forall\ p \geq 1,\ F(2p) = F(2p-1) = 2p + 1 - 2F(p)$ définition récursive

Les premières valeurs de F sont :

1, 1, 3, 3, 1, 1, 3, 3, 9, 9, 11, 11, 9, 9, 11

Donner un algorithme itératif avec pile permettant de calculer F(p), en utilisant la traduction récursif-itératif du II-3-d).

Exo.12) 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 au centre du tableau si 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 $\frac{5+6}{2} = 5{,}5$. En triant le tableau pour obtenir la médiane, 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 pour les tris les plus rapides.

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 l'algorithme du 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 générale, l'algorithme nommé **Quickselect** de paramètre k compris entre 1 et la longueur du tableau détermine le k-ème élément du tableau par ordre de tri de la manière suivante :

```
Fonction Quickselect(Tableau T ayant n éléments, rang k de l'élément cherché)
    Choisir un pivot x élément de T
    Comparer les éléments du tableau à ce pivot pour les séparer en deux sous-tableaux Tinf et Tsup,
les éléments de Tinf étant strictement inférieurs à x et ceux de Tsup étant strictement supérieurs à x.
    Si le sous-tableau Tinf possède k éléments ou plus, alors appliquer Quickselect(Tinf, k)
    Sinon
        Si le sous-tableau Tsup possède n-k+1 éléments ou plus, alors
            appliquer Quickselect(Tsup, k-n+longueur(Tsup_i))
        Sinon
            le résultat est le pivot x
        Finsi
    Finsi
```

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

a) Ecrire dans le langage de votre choix une fonction Quickselect.

b) On suppose les éléments du tableau distincts, et on suppose que les données du tableau sont rangées au hasard, toutes les permutations possibles étant équiprobables. Montrer que le nombre moyen de comparaisons C(n, k) entre éléments du tableau lors de l'exécution de Quickselect(Tableau T ayant n éléments, rang k de l'élément cherché) vérifie la relation de récurrence suivante :

$$C(n, k) = n + \frac{1}{n}\left(\sum_{m=k+1}^{n} C(m-1, k) + \sum_{m=1}^{k-1} C(n-m, k-m)\right)$$

c) Montrer, par récurrence sur n, que, pour tout k, $C(n, k) \leq 4n$.

2- Solutions

Sol.1) a) Utilisons la relation matricielle donnée dans le cours : $\begin{pmatrix}1&1\\1&0\end{pmatrix}^n = \begin{pmatrix}F_{n+1}&F_n\\F_n&F_{n-1}\end{pmatrix}$ et remplaçons-y n par $n + m$. On obtient :

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

d'où le résultat.

b) Dans l'invariant :

$\exists\, k, \exists\, r, n_0 = 2^k\, n + r, 0 \le r < 2^k, a = F(2^k), b = F(2^k - 1), p = F(r), q = F(r - 1)$

k indique combien de fois la boucle a été exécutée. r est le reste de la division euclidienne de n_0 par 2^k.

La relation est vraie au départ en prenant $k = 0$, et donc $r = 0$, $a = F(1) = 1$, $b = F(0) = 0 = p$, $q = F(-1) = 1$. Supposons qu'elle soit vraie à l'issue de la k-ème boucle et exécutons cette boucle une fois de plus en supposant n non nul.

- Si n est impair, alors p et q prennent les valeurs :

  $p = (F(2^k) + F(2^k - 1))F(r) + F(2^k)F(r - 1)$
  $= F(2^k + 1)F(r) + F(2^k)F(r - 1)$
  $= F(2^k + r)$ en appliquant (*)
  $q = F(2^k)F(r) + F(2^k - 1)F(r - 1)$
  $= F(2^k + r - 1)$ en appliquant (*)

  or n impair signifie qu'il existe n' tel que $n = 2n' + 1$ et donc que :

  $n_0 = 2^k\, n + r = 2^{k+1}n' + 2^k + r = 2^{k+1}n' + r'$ en posant $r' = 2^k + r$

  donc $p = F(r')$
  $q = F(r' - 1)$

- Si n est pair, alors il existe n' tel que $n = 2n'$ et donc :

  $n_0 = 2^k\, n + r = 2^{k+1}n' + r = 2^{k+1}n' + r'$ en posant $r' = r$

  mais p et q ne changent pas de valeurs donc :
  donc :

  $p = F(r')$
  $q = F(r' - 1)$

On remarque que, dans les deux cas, $0 \le r' \le 2^k + r < 2^k + 2^k = 2^{k+1}$, $p = F(r')$, $q = F(r')$.

a et b prennent ensuite les valeurs :

$a = F(2^k)^2 + 2F(2^k)F(2^k - 1) = F(2^k)(F(2^k) + 2F(2^k - 1))$
$= F(2^k)(F(2^k + 1) + F(2^k - 1))$
$= F(2^k + 1)F(2^k) + F(2^k)F(2^k - 1)$
$= F(2^k + 2^k)$ en appliquant (*)
$= F(2^{k+1})$
$b = F(2^k)^2 + F(2^k - 1)^2 = F(2^k)^2 + (F(2^k + 1) - F(2^k))F(2^k - 1)$
$= F(2^k + 1)F(2^k - 1) + F(2^k)(F(2^k) - F(2^k - 1))$
$= F(2^k + 1)F(2^k - 1) + F(2^k)F(2^k - 2)$
$= F(2^k + 2^k - 1)$ en appliquant (*)
$= F(2^{k+1} - 1)$

Finalement, $n_0 = 2^{k+1}n' + r'$, $0 \le r' < 2^{k+1}$, $a = F(2^{k+1})$, $b = F(2^{k+1} - 1)$, $p = F(r')$, $q = F(r' - 1)$. Comme on attribue ensuite à n la valeur n', et donc à r la valeur r', l'invariant de boucle est bien vérifié.

On itère jusqu'à ce que $n = 0$, mais dans ce cas, $r = n_0$ et $p = F(r) = F(n_0)$ comme annoncé.

Sol.2) a) On donne ci-dessous le calcul de inv(*n*, *b*) en base de numération *b*. Le résultat se trouve dans *m* :

```
m ← 0
tant que n ≠ 0 faire
        # invariant de boucle : à l'issue de la k-ème boucle, m est constitué des k chiffres
        # les plus à droite du n initial, écrits en sens inverse. Le n actuel est le n initial
        # privé de ces k chiffres.
        m ← b*m + (n mod b)
        # n mod b est le chiffre le plus à droite du n actuel, donc le (k+1)-ème chiffre le plus
        # à droite du n initial. On a concaténé à droite de m ce (k+1)-ème chiffre.
        n ← n/b                    # division entière
        # on supprime le chiffre le plus à droite du n actuel.
finfaire
```

b) La hauteur palindromique de *n* est calculée ci-dessous dans la variable *h*. Il vaut mieux donner un calcul itératif plutôt que récursif, le nombre d'appels de fonction dans le dernier cas risquant de dépasser la capacité de la pile de récursivité :

```
h ← 0
m ← inv(n,b)
tant que m ≠ n faire
        # Invariant de boucle : m est le palindrome de n.
        # h est le nombre d'itérations effectuées.
        # La somme de h et de la hauteur palindromique du n courant est égale
        # à la hauteur palindromique du n initial
        n ← m+n
        m ← inv(n,b)
        h ← h+1
        # Comme n a pris la valeur n+m, sa hauteur palindromique a diminué de 1
        # alors que la valeur de h a augmenté de 1.
        # Donc la somme de h et de la hauteur palindromique du n courant est bien
        # toujours égale à la hauteur palindromique du n initial.
finfaire
# On termine la boucle quand n est un palindrome, donc a une hauteur palindromique
# nulle. h est donc alors égal à la hauteur palindromique du n initial.
```

On peut aussi obtenir l'algorithme précédent en utilisant la conversion récursif-itératif telle qu'elle est décrite en II-3-b), en calquant ce qu'on a fait pour la longueur de la suite de Collatz.
Un nombre pour lequel *h* n'est pas défini s'appelle **nombre de Lychrel**. On pourra lire sur ce sujet : Jean-Paul Delahaye, *Déconcertantes conjectures*, Pour La Science, n°367, mai 2008, 92-97.
ou consulter le site www.p196.org.

Sol.3) a) F(*n*) est égal au nombre de 1 dans la décomposition binaire de *n*. Cela se montre par récurrence :

❑ C'est vrai pour $n = 1$, pour lequel $f(n) = 1$.

❑ Supposons que cela soit vrai pour tous les entiers jusqu'à $n - 1$ et montrons-le pour *n* :

si *n* est pair, de la forme $2p$, alors la décomposition binaire de *n* s'obtient à partir de celle de *p* en décalant les chiffres d'un rang et en rajoutant un 0 comme chiffre des unités. Le nombre de 1 dans les deux décompositions est donc le même, conformément à la formule $F(2p) = F(p)$.

si *n* est impair, de la forme $2p + 1$, alors la décomposition binaire de *n* s'obtient à partir de celle de *p* en décalant les chiffres d'un rang et en rajoutant un 1 comme chiffre des unités. Le nombre de 1 dans la décomposition de *n* est donc supérieur d'une unité à celle de *p*, conformément à la formule $F(2p + 1) = F(p) + 1$.

b) Par un raisonnement analogue, on montrera que la fonction F calcule maintenant le nombre de 0 dans la décomposition binaire de *n*.
c) De même, on montrera que F calcule ici le nombre de 0 successifs situé dans la partie droite du développement binaire de *n*, ou bien de manière équivalente, la plus grande puissance de 2 qui divise *n*.
Les fonctions F définies en a) et b) relèvent d'une définition récursive décrite dans le II-3-c).

si $x \in B$, $F(x) = f(x)$
sinon, $F(x) = h(x) * F(g(x))$

avec $*$ égal à la somme, le neutre *e* égal à 0, l'ensemble de base B égal à {1}, $g(n) = n/2$ (quotient entier de *n* par 2). Seules diffèrent les définitions de *f* et *h* Appliquons la traduction récursif-itératif qui y est décrite. On reconnaîtra alors l'algorithme itératif usuel correspondant à chacune des fonctions.

a) On prend $f(1) = 1$, $h(n) = \begin{cases} 0 \text{ si } n \text{ est pair} \\ 1 \text{ si } n \text{ est impair} \end{cases} = n \bmod 2$

```
a ← 0
tant que n ≠ 1
        si n est impair alors a ← a + 1 finsi
        n ← n/2
finfaire
retourner(a+1);
```

qu'on peut aussi écrire sous la forme suivante (avec une boucle de plus) :

```
a ← 0
tant que n ≠ 0
        a ← a + (n mod 2)
        n ← n/2
finfaire
retourner(a);
```

b) On prend $f(1) = 0$, $h(n) = \begin{cases} 1 \text{ si } n \text{ est pair} \\ 0 \text{ si } n \text{ est impair} \end{cases}$

```
a ← 0
tant que n ≠ 1
        si n est pair alors a ← a + 1 finsi
        n ← n/2
finfaire
retourner(a);
```

c) Ce cas relève également du II-3-c), mais de manière moins visible. On prend ici $*$ égal à la somme, B = {1}, $f(1) = 0$, $g(n) = \begin{cases} n/2 \text{ si } n \text{ est pair} \\ 1 \text{ si } n \text{ est impair} \end{cases}$, $h(n) = \begin{cases} 1 \text{ si } n \text{ est pair} \\ 0 \text{ si } n \text{ est impair} \end{cases}$.

Pour *n* impair, on aura bien :

$h(n) + F(g(n)) = 0 + F(1) = f(1) = 0 = F(n)$

et si *n* est pair :

$h(n) + F(g(n)) = 1 + F(n/2) = F(n)$

La traduction récursif-itératif donne :

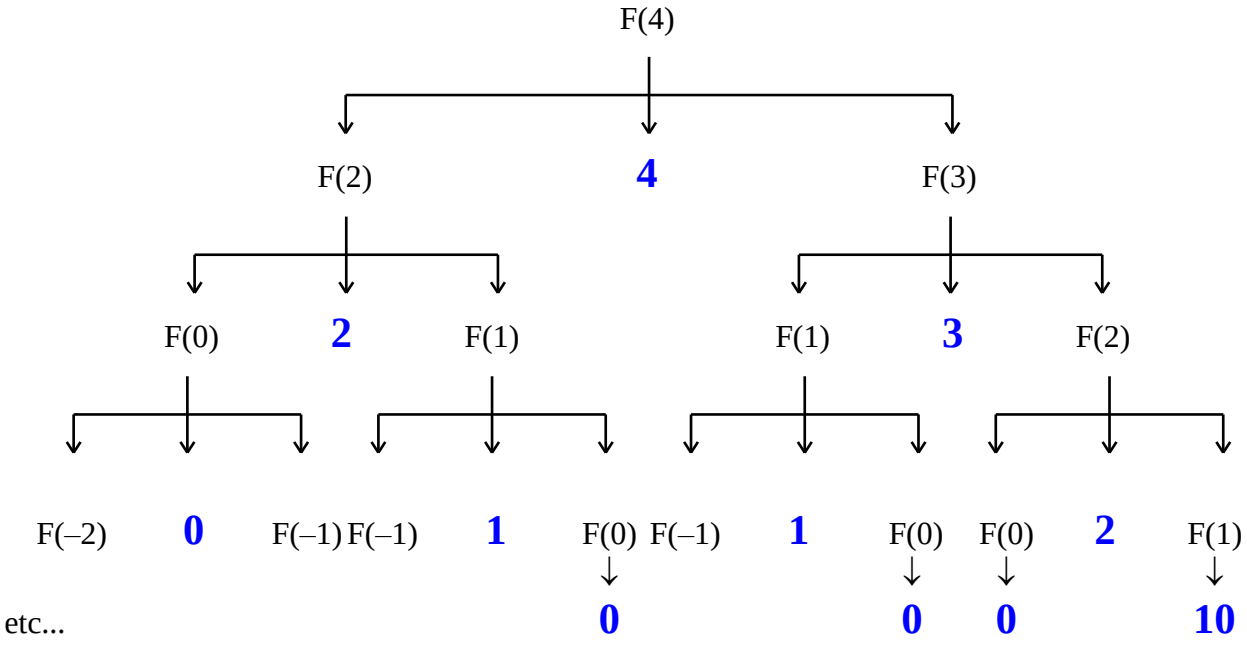
```
a ← 0
tant que n ≠ 1faire
        si n est pair alors
                a ← a * 1
                n ← n/2
        sinon
                n ← 1
```

```
            finsi
        finfaire
        retourner(a);
```

La boucle s'arrête dès que *n* prend une valeur impaire (*n* prenant transitoirement la valeur 1 avant l'arrêt de la boucle), de sorte qu'on peut écrire également le programme précédent sous la forme plus simple suivante :

```
a ← 0
tant que n est pair faire
        a ← a * 1
        n ← n/2
finfaire
retourner(a);
```

Sol.4) Lors de l'exécution de F(4), il s'affichera 0, 2, 1, 0, 4, 1, 0, 3, 0, 2, 1, 0. En effet, l'appel de F(4) se traduit par l'appel de F(2), l'affichage de 4, puis l'appel de F(3), donc 4 apparaîtra après tous les affichages de F(2) et avant tous les affichages de F(3). On procède de même pour F(2) et F(3). On a alors l'arborescence suivante, parcourue récursivement dans l'ordre gauche-centre-droit :



Sol.5) L'annexe I du chapitre L1/DENOMBRE.PDF donne les relations suivantes :

$\sigma(0, 0) = 1$

$\sigma(n, 0) = 0$ si $n > 0$

$\sigma(n, p) = 0$ si $n < p$

$\sigma(n, p) = p \times (\sigma(n-1, p-1) + \sigma(n-1, p))$ si $1 \leq p \leq n$

Ces relations permettent d'écrire l'algorithme récursif suivant (en Python) :

```
def nbsurj(n,p):
    if n==0 and p==0:
        return(1)
    elif p==0 or n<p:
        return(0)
    else:
        return(p*(nbsurj(n-1,p-1)+nbsurj(n-1,p)))
    # fin if
```

Pour prouver que l'algorithme précédent se termine pour tout couple (*n*, *p*), posons :

$\varphi(0, 0) = \varphi(n, 0) = 0$

$$\varphi(n, p) = \begin{cases} 0 \text{ si } n < p \\ n \text{ si } 1 \le p \le n \end{cases}$$

Pour tout couple $x = (n, p)$, $\varphi(x) = 0$ pour les cas de base. Sinon, pour $1 \le p \le n$, x est défini à partir des couples $y = (n - 1, p - 1)$ et $z = (n - 1, p)$. Comme $\varphi(y) < \varphi(x)$ et $\varphi(z) < \varphi(x)$, φ décroît strictement à chaque appel récursif et cela assure que nbsurj(n, p) est calculé en un nombre fini d'étapes.

Nous allons évaluer le temps de calcul en dénombrant le nombre $R(n, p)$ d'appels récursifs effectués pour calculer $\sigma(n, p)$. On a :

$$R(n, p) = \begin{cases} 0 \text{ si } p = 0 \text{ ou si } n < p \\ 2 + R(n - 1, p - 1) + R(n - 1, p) \text{ sinon} \end{cases}$$

Le 2 du membre de droite provient de l'appel de $\sigma(n - 1, p - 1)$ et de $\sigma(n - 1, p)$, chacun d'eux faisant respectivement appel à $R(n - 1,p - 1)$ et $R(n - 1, p)$ autres fonctions σ.

Montrons, par récurrence sur n, que, pour tout $p \le n$, $R(n, p) = 2\binom{n+1}{p} - 2$. La relation est vérifiée pour $n = p = 0$. Pour $n = 1$, on a

$$R(1, 0) = 0 = 2\binom{2}{0} - 2$$

et $$R(1, 1) = 2 + R(0, 0) + R(0, 1) = 2 = 2\binom{2}{1} - 2$$

Supposons la relation vraie au rang $n - 1$. On a alors, au rang n :

$$R(n, 0) = 0 = 2\binom{n+1}{0} - 2$$

$$\forall\, p \in [\![1, n]\!],\ R(n, p) = 2 + 2\binom{n}{p-1} - 2 + 2\binom{n}{p} - 2$$

$$= 2\left(\binom{n}{p-1} + \binom{n}{p}\right) - 2$$

$$= 2\binom{n+1}{p} - 2$$

qui est bien le résultat attendu.

Pour n donné et p variant de 1 à n, la valeur de $R(n, p)$ est maximale pour $p = \left\lfloor \frac{n+1}{2} \right\rfloor$. Pour n pair, égal à $2m$, ce maximum est :

$$R(2m, m) = 2\binom{2m+1}{m} - 2$$

$$\sim 2\binom{2m+1}{m} \qquad \text{quand } m \text{ tend vers l'infini}$$

$$\sim \frac{2(2m+1)}{m+1}\frac{(2m)!}{m!m!}$$

$$\sim 4\frac{(2m)^{2m} e^{-2m}\sqrt{4\pi m}}{(m^m e^{-m}\sqrt{2\pi m})^2} \qquad \text{d'après la formule de Stirling } n! \sim n^n e^{-n}\sqrt{2\pi n}.$$

Voir les chapitres L2/SERIES.PDF
ou L2/SUITESF.PDF.

$$\sim 4\frac{2^{2m}}{\sqrt{\pi m}} = O\left(\frac{2^n}{\sqrt{n}}\right)$$

On pourra vérifier qu'on obtient un résultat identique pour n impair.

Cela montre que le calcul de R(n, p) peut faire intervenir une croissance quasi exponentielle avec un facteur 2^n. L'algorithme récursif est donc peu efficace, sauf si on applique une technique de mémoïsation.

Si on sait a priori qu'on utilisera les valeurs $\sigma(n, p)$ pour $1 \le p \le n \le N$ avec N connu, il est plus efficace d'utiliser un programme itératif qui stocke les $\sigma(n, p)$ dans un tableau T[n, p] de N^2 éléments :

```
pour p ← 1 à N faire
        si p=1 alors T[1,p] ← 1 sinon T[1,p] ← 0 finsi
finfaire
pour n ← 2 à N faire
        T[n,1] ← 1
        pour p ← 2 à N faire
                si p ≤ n alors T[n,p] ← p * (T[n-1,p-1] + T[n-1,p])
                sinon T[n,p] ← 0
                finsi
        finfaire
finfaire
```

Si on ne souhaite pas mémoriser toutes les valeurs de $\sigma(n, p)$, $1 \le p \le n \le N$, on peut se contenter d'un tableau unidimensionnel stockant les valeurs de $\sigma(n, p)$ pour le dernier n en cours de calcul :

```
pour p ← 1 à N faire
        # On stocke les valeurs de σ(1, p)
        si p=1 alors T[p] ← 1 sinon T[p] ← 0 finsi
finfaire
pour n ← 2 à N faire
        # Invariant de boucle : T stocke les valeurs σ(n-1, p).
        temp1 ← 1
        # temp1 stocke la valeur de T[1] = 1 = σ(n-1, 1)
        pour p ← 2 à n faire
                # Invariant de sous-boucle :
                # temp1 stocke la valeur T[p-1] = σ(n-1, p-1).
                # T[k] pour 1≤k<p stocke les valeurs σ(n, k).
                # T[k] pour k≥p stocke les valeurs σ(n-1,k).
                # On va modifier le T[p] du rang n-1 en le T[p] du rang n
                temp2 ← T[p]
                # temp2 stocke T[p] = σ(n-1, p). On peut maintenant modifier T[p].
                T[p] ← p * (temp1 + T[p])
                # La valeur de T[p] est maintenant σ(n, p)
                temp1 ← temp2
                # temp1 a pris la valeur σ(n-1, p) du T[p] du rang n-1.
                # p sera incrémenté de 1 à boucle suivante
                # et temp1 aura alors la valeur T[p-1] au rang n-1
        finfaire
        # T stocke les valeurs σ(n, p). n sera incrémenté de 1 à la boucle suivante
        # et T stockera alors les valeurs de σ(n-1, p) au début de cette prochaine boucle.
finfaire
```

A la sortie, le tableau T[p] stocke les valeurs de $\sigma(N, p)$. L'algorithme de calcul est en $O(N^2)$ en raison des deux boules imbriquées.

On peut aussi écrire une procédure itérative qui calcule $\sigma(n, p) = \sum_{k=0}^{p} \binom{p}{k} (-1)^{p-k} k^n$ (formule qu'on trouve dans l'annexe I de L1/DENOMBRE.PDF), mais son inconvénient est de faire des différences de nombres qui peuvent être élevés.

Sol.6) On peut penser à utiliser une programmation récursive (ci-dessous en Python) :

```
def b(n):
    if n==0:
        return(1)
    else:
        S=0
        for k in range(n):    # k varie de 0 à n-1
            S=S-binomial(n+1,k)*b(k)
        # fin for
        return(S/(n+1))
    # fin if
```

en supposant que le langage utilisé possède la fonction binomial(n,p) (sinon la programmer à part). Mais les appels récursifs aux $b(k)$ pour $0 \le k \le n-1$ font rapidement exploser la pile des appels de fonctions si on n'utilise pas une technique de mémoïsation. On peut proposer une programmation itérative stockant les valeurs b_k dans un tableau de $n+1$ éléments :

```
T[0] ← 1
pour i variant de 1 à n faire
        # Invariant de boucle : T[0] = b0, ..., T[i-1] = bi –1
        S ← 0
        pour k variant de 0 à i-1 faire
                S ← S - binomial(i+1,k)*T[k]
        finfaire
        T[i] ← S/(i+1)
        # T[0] = b0, ..., T[i] = bi
finfaire
```

Le tableau final T contiendra les valeurs de b_0 à b_n. Le temps de calcul semble être en $O(n^2)$, provenant de la double boucle sur i et k, mais ceci à condition que le changement de valeur de S au sein de la double boucle se fasse en un temps O(1), uniformément borné pour tout i et tout k. Or ce changement de valeur fait intervenir le coefficient binomial $\binom{i+1}{k}$. Si on utilise l'expression du coefficient binomial au moyen des factorielles pour le calculer, le temps de calcul de ce coefficient binomial n'est pas un O(1), mais un $O(i)$, et le temps total de l'algorithme ne sera pas :

$$\sum_{i=1}^{n}\sum_{k=0}^{i-1} O(1) = \sum_{i=1}^{n} i\,O(1) = \frac{n(n+1)}{2}\,O(1) = O(n^2)$$

mais $$\sum_{i=1}^{n}\sum_{k=0}^{i-1} O(i) = \sum_{i=1}^{n} i\,O(i) = \sum_{i=1}^{n} i^2\,O(1) = \frac{n(n+1)(2n+1)}{6}\,O(1) = O(n^3)$$

Utiliser l'expression $\binom{i+1}{k} = \frac{(i+1)i...(i+2-k)}{k!}$ fait intervenir un temps de calcul en $O(k)$ car il y a k facteurs au numérateur et au dénominateur, mais ne change pas fondamentalement les choses. En effet :

$$\sum_{i=1}^{n}\sum_{k=0}^{i-1} O(k) = \sum_{i=1}^{n}\sum_{k=0}^{i-1} k\,O(1) = \sum_{i=1}^{n} \frac{i(i-1)}{2}\,O(1) = \frac{(n-1)n(n+1)}{6}\,O(1) = O(n^3)$$

Il vaut mieux utiliser la relation de récurrence $\binom{i+1}{k+1} = \frac{i+1-k}{k+1}\binom{i+1}{k}$ et profiter de la boucle sur k pour calculer ainsi $\binom{i+1}{k}$:

```
T[0] ← 1
```

```
pour i variant de 1 à n faire
        # Invariant de boucle : T[0] = b_0, ..., T[i-1] = b_{i-1}
        S ← 0
        coeffbin ← 1
        # coeffbin = binomial(i+1,0)
        pour k variant de 0 à i-1 faire
                # coeffbin = binomial(i+1,k)
                S ← S - coeffbin*T[k]
                coeffbin ← coeffbin*(i+1-k)/(k+1)
                # coeffbin = binomial(i+1,k+1)
        finfaire
        T[i] ← S/(i+1)
        # T[0] = b_0, ..., T[i] = b_i
finfaire
```

On obtient bien alors un algorithme dont la complexité en temps de calcul est $O(n^2)$.

Certains langages de programmation permettent d'exécuter l'algorithme précédent avec un calcul exact de fraction. Les premières valeurs de la suite b sont :

$$1, -\frac{1}{2}, \frac{1}{6}, 0, -\frac{1}{30}, 0, \frac{1}{42}, 0, -\frac{1}{30}, 0, \frac{5}{66}, \ldots$$

Les nombres de Bernoulli ont de très nombreuses applications. Nous nous bornerons à donner (sans preuve) le développement en série entière de la fonction tan et th, pour x élément de $]-\frac{\pi}{2}, \frac{\pi}{2}[$:

$$\tan(x) = \sum_{n=1}^{\infty} (-1)^{n+1} \frac{(16^n - 4^n) b_{2n}}{(2n)!} x^{2n-1}$$

$$\mathrm{th}(x) = \sum_{n=1}^{\infty} \frac{(16^n - 4^n) b_{2n}}{(2n)!} x^{2n-1}$$

Sol.7) Pour tout entier a et b tels que $a \geq b \geq 1$, soit $K = K(a, b)$ le nombre maximal pour lequel on puisse déterminer un nombre N appartenant à $[\![1, K]\!]$ en posant a questions avec au plus b réponses NON.

❑ On a :

$$K(a, 1) = a + 1$$

En effet, puisque la réponse NON exige une solution immédiate, les questions ne peuvent produire que des réponses OUI jusqu'au moment où l'on est sûr de connaître le nombre N. Chaque question ne peut donc servir qu'à séparer un unique élément des autres. Les questions sont donc successivement :

« N > 1 ? » « N > 2 ? » ... « N > a ? »

❑ On a $K(a, a) = 2^a$ car on pose a questions, chacune d'elle pouvant recevoir la réponse OUI ou NON, et on tombe dans un problème classique de dichotomie, la première question étant « N > 2^{a-1} ? ».

❑ Pour $K = K(a, b)$ avec a et b quelconque, on pose la première question, pour un certain c qu'on définira plus loin : « N > c ? »

Premier cas : la réponse est NON. Alors N figure parmi 1, ..., c. Il reste $a - 1$ questions mais on n'a plus droit qu'à $b - 1$ NON. La valeur la plus grande de c qu'on puisse choisir est donc égale à $K(a - 1, b - 1)$.

Deuxième cas : la réponse est OUI. Alors N figure parmi $c + 1$, ..., K, constitué de $K - c$ éléments. Il reste $a - 1$ questions et on a encore droit à b NON. A translation près de la numérotation des éléments, la valeur la plus grande possible de $K - c$ est donc égale à $K(a - 1, b)$.

Il en résulte que la valeur la plus grande possible de K vérifie la relation :

$$K = K(a, b) = K(a - 1, b) + K(a - 1, b - 1)$$

et la première question est « N > K(a – 1,b – 1) ? ». On peut alors calculer récursivement la valeur K(a, b). On trouve bien K(12, 3) = 299. La première question est « N > K(11, 2) = 67 ? ».

Sol.8) a) Pour la procédure inttogray, on utilise une procédure récursive traduisant la définition. On traite à part la valeur initiale 0. On cherche ensuite p, puissance de 2, et k, compris entre 0 et p – 1 tels que $n = p + k$. p n'est autre que le 2^m de l'énoncé, mais m n'est pas directement utilisé. Parallèlement, on calcule une puissance q de 10 ayant le même exposant m que celui de p en tant que puissance de 2. q permet de placer le 1 qui débute le code de Gray de n.

```
def inttogray(n):
    if n==0:
        return(0)
    else:
        p=1
        q=1
        while 2*p<=n:
            p=2*p
            q=10*q
        # fin while
        # On sort de la boucle avec p≤n<2*p.
        k=n-p
        return(q+inttogray(p-k-1))
    # fin if
```

Pour la procédure graytoint, on recherche le rang du 1 de gauche. Pour cela, si le paramètre g est un entier de m + 1 chiffres 0 ou 1, commençant par le chiffre 1 et décrivant un code de Gray, on effectue une boucle permettant de stocker dans une variable p le nombre 2^m et dans une variable q le nombre 10^m. g – q est l'entier décrivant le code de Gray privé de son chiffre de gauche. Un appel récursif permet de convertir ce code en un entier r. L'entier de code g est $n = 2^m + k$ avec k tel que $r = 2^m - 1 - k$, ce qui donne $n = 2^m + k = 2p - r - 1$.

```
def graytoint(g):
    if g==0:
        return(0)
    else:
        temp=g
        p=1
        q=1
        while temp!=1:
            # Invariant de boucle :
            # si u est tel que p = 2^u, alors q = 10^u.
            # temp est obtenu à partir de g en
            # supprimant u chiffres à sa droite.
            p=2*p
            q=10*q
            temp=temp//10
        # fin while
        # On sort de la boucle while avec temp=1.
        # Donc si g possède m+1 chiffres,
        # temp est obtenu à partir de g
        # en supprimant m chiffres à sa droite.
```

```
        # Donc p = 2^m et q = 10^m.
        return(2*p-graytoint(g-q)-1)
    # fin if
```

b) Par récurrence sur n. La propriété énoncée est vraie pour les entiers $n = 0$ ou 1. Supposons la relation vraie pour tout $n < 2^p$. Soit $n = 2^p + k$, avec $0 \leq k < 2^p$. Le code binaire de n est 1 (pour le chiffre b_p), suivi du code binaire de k. Le code de Gray de n est 1 (pour le chiffre g_p) suivi du code de Gray de $2^p - k - 1$. Soient $g_{p-1}g_{p-2}...g_0$ les chiffres de ce code. Le code binaire de $2^p - k - 1$ admet pour chiffres, en utilisant l'hypothèse de récurrence, et en partant du chiffre de rang 0 :

$g_0 + g_1 + ... + g_{p-1} \bmod 2$
$g_1 + ... + g_{p-1} \bmod 2$
...
$g_{p-1} \bmod 2$

Le code binaire de k admet pour chiffres :

$1 + g_0 + g_1 + ... + g_{p-1} \bmod 2$
$1 + g_1 + ... + g_{p-1} \bmod 2$
...
$1 + g_{p-1} \bmod 2$

(En ajoutant les deux codes binaires, on obtient en binaire 11....1 avec p chiffres, qui est le code binaire de $2^p - 1$). Ce code binaire de k est bien (puisque $g_p = 1$) :

$g_0 + g_1 + ... + g_{p-1} + g_p \bmod 2$
$g_1 + ... + g_{p-1} + g_p \bmod 2$
...
$g_{p-1} + g_p \bmod 2$

et celui de n est :

$g_0 + g_1 + ... + g_{p-1} + g_p \bmod 2$
$g_1 + ... + g_{p-1} + g_p \bmod 2$
...
$g_{p-1} + g_p \bmod 2$
$g_p = 1$

La relation $g_i \equiv b_i + b_{i+1} \bmod 2$ se déduit immédiatement de $b_i \equiv g_i + ... + g_p \bmod 2$ et $b_{i+1} \equiv g_{i+1} + ... + g_p \bmod 2$.

Pour passer du code binaire au code de Gray, on lit le code binaire de droite à gauche, en mémorisant deux chiffres consécutifs b_i et b_{i+1}, de façon à en déduire le chiffre g_i du code de Gray correspondant. Un nombre q stocke les puissances de 10 successives permettant de disposer le chiffre trouvé à l'endroit voulu dans l'entier qui servira de résultat final.

```
def bintogray(b):
    g=0
    q=1
    while b!=0:
        # invariant de boucle :
        # i étant le nombre de boucles effectuées,
        # q = 10^i, et g est le code constitué
        # des bits g_{i-1}...g_0 et
        # b est le code binaire initial privé
        # des bits b_{i-1}...b_0.
        c=b%10                  # c est le bit b_i
        b=b//10                 # b se voit privé du bit b_i.
        c=(c + (b%10))%2        # c ≡ b_i + b_{i+1} mod 2 ≡ g_i
        g=g+c*q                 # g est le code constitué
                                # des bits g_i...g_0.
        q=q*10                  # q = 10^(i+1)
```

```
        # Une boucle étant effectuée, i augmente de 1.
    #fin while.
    return(g)
```

Inversement, le code de Gray étant g, on calcule la puissance de 10, notée q, dont l'exposant est le nombre de bits de g. On lit ensuite le code g de gauche à droite en ajoutant modulo 2 les uns après les autres les bits de g, obtenant ainsi les bits $b_i \equiv b_{i+1} + g_i \bmod 2$ cherchés. Parallèlement, q est divisé par 10 pour chaque chiffre du code de Gray lu. Les bits b_i sont multipliés par q et accumulés dans un nombre entier b qui servira de code binaire final.

```
def graytobin(g):
    if g==0:
        return(0)
    else:
        temp=g
        q=1
        while temp!=0:
            # invariant de boucle.
            # Si q = 10^m, temp est le code obtenu
            # en supprimant m bits à droite de g.
            q=10*q
            temp=temp//10
        #fin while
        # q = 10^j où j est le nombre de bits de g.
        # Le bit 1 le plus à gauche de g ou
        # du code binaire b correspond à
        # q/10 = 10^(j-1) en notation décimale.
        c=0
        b=0
        temp=g
        while q!=1:
            # invariant de boucle :
            # si k est le nombre de boucles déjà
            # effectuées et si i=j-1-k, alors
            # q = 10^(i+1), c = b_{i+1},
            # temp est égal à g privé de
            # ses k=j-1-i bits de gauche.
            # temp possède les i+1 bits g_i...g_0.
            # b est l'entier représentant le code binaire
            # b_{j-1}...b_{i+1}0...0 avec i+1 chiffres 0.
            q=q//10     # q=10^i indique la place où doit
                        # figurer b_i dans l'entier b.
            c=(c+temp//q)%2  # temp//q est égal à g_i,
                             # donc c prend la valeur
                             # b_{i+1}+g_i mod 2 ≡ b_i
            b=b+c*q          # b représente le code binaire
                             # b_{j-1}...b_i0...0
                             # avec i chiffres 0.
            temp=temp%q      # on enlève le bit de gauche à temp.
                             # temp est égal à g_{i-1}...g_0.
          # Une boucle étant effectuée, i décroît de 1.
        #fin while.
        # On quitte la boucle quand q=1, c=b_0, temp est vide.
        # et b est l'entier représentant b_{j-1}...b_0.
        return(b)
```

```
    #fin if
```

c) ❑ Supposons que le code de Gray de n possède un nombre pair de 1. Notons $b_i(n)$ le chiffre binaire de rang i de l'entier n (respectivement $g_i(n)$ pour le code de Gray). Alors $b_0(n)$, chiffre de droite du code binaire de n, qui est la somme modulo 2 des chiffres du code de Gray, est nul. Le code binaire de $n + 1$ est le même que celui de n, mais où b_0 est remplacé par 1. Il en résulte que $g_0(n + 1)$, chiffre de droite du code de Gray de $n + 1$, vaut $1 + b_1(n + 1) \bmod 2 \equiv 1 + b_1(n)$ alors que $g_0(n) = b_1(n)$. Donc $g_0(n + 1) \equiv g_0(n) + 1 \bmod 2$. Il suffit donc de changer le chiffre de droite du code de Gray de n. Les autres chiffres du code de Gray sont inchangés, puisqu'ils font intervenir les chiffres binaires b_i, $i \geq 1$, qui sont identiques entre n et $n + 1$.

❑ Supposons maintenant que le code de Gray de n possède un nombre impair de 1. Alors $b_0(n) = 1$. Soit p l'entier tel que $b_{p+1}(n) = 0$ et $b_p(n) = b_{p-1}(n) = ... = b_1(n) = b_0(n) = 1$. Alors, en ajoutant 1 à n, il y a une propagation de la retenue et on obtiendra pour code binaire de $n + 1$ les chiffres suivants :

$b_i(n + 1) = b_i(n)$ si $i \geq p + 2$

$b_{p+1}(n + 1) = 1$

$b_p(n + 1) = b_{p-1}(n + 1) = ... = b_1(n + 1) = b_0(n + 1) = 0$

En ce qui concerne le code de Gray de n, on obtiendra à partir des bits $b_i(n)$:

$g_{p+1}(n) = b_{p+2}(n)$

$g_p(n) = 1$

$g_{p-1}(n) = ... = g_0(n) = 0$

alors que celui de $n + 1$ donne :

$g_{p+1}(n + 1) \equiv 1 + b_{p+2}(n) \bmod 2$

$g_p(n + 1) = 1$

$g_{p-1}(n + 1) = ... = g_0(n + 1) = 0$

On voit donc qu'il s'agit de rechercher le premier 1 rencontré à partir de la droite dans le code de Gray de n, puis de changer le chiffre à gauche de ce 1.

Une procédure regroupant les deux cas consiste à parcourir le code de Gray de droite à gauche, à repérer le rang du premier 1 et à faire la somme de tous les chiffres. Selon la parité de cette somme, on change le chiffre de droite ou bien le chiffre à gauche du premier 1. Notons c le chiffre en cours de lecture, s la somme des chiffres, p la puissance de 10 correspondant au premier 1 trouvé. Trouve est une variable booléenne indiquant s'il est nécessaire de continuer à chercher ce 1.

```
def successeur(g):
    temp=g
    p=1
    s=0
    trouve=False
    while temp!=0:
        # invariant de boucle :
        # si on a exécuté i boucles, temp est
        # égal à g privé des i bits de
        # droite g_0, ... g_{i-1}.
        # s est égal à la somme g_0+...+g_{i-1}.
        # trouve est un booléen indiquant si
        # l'un des bits g_0,...,g_{i-1} vaut 1.
        # Si tel est le cas et si g_k est le
        # premier bit valant 1, p est égal à 10^k.
        # Sinon p=10^i.
        c=temp%10          # c est égal au bit g_i
        temp=temp//10      # temp est égal à g privé
                           # de i+1 bits de droite.
        s=s+c              # s = g_0 + ... + g_i
```

```
        if not trouve:     # p=10^i
            if c==1:
                trouve=True  # trouve indique que
                             # l'un des bits g_0,...,g_i
                             # vaut 1 et p indique
                             # que c'est g_i.
            else:
                p=10*p       # p=10^(i+1)
        #fin if
        # On a terminé une boucle, i augmente de 1.
    #fin while
    # s est égal à la somme de tous les bits de g.
    # p indique où se trouve le premier 1
    # à partir de la droite  parmi les bits de g.
    if s%2==0:             # on teste la parité de s.
        # s est pair
        if g%2==0:         # g%2 est égal au bit g_0.
            return(g+1)    # si g_0 = 0, on le remplace par 1.
        else:
            return(g-1)    # si g_0 = 1, on le remplace par 0.
        #fin if
    elif (g//(10*p))%10==0: # s est impair.
                            # (g//(10*p))%10 est le bit
                            # situé à gauche du premier
                            # 1 rencontré, et il vaut ici 0.
        return(g+10*p)      # on remplace ce bit par 1.
    else:
        return(g-10*p)      # si ce bit valait 1,
                            # on le remplace par 0.
    #fin if
```

Sol.9) a) D'après la relation de récurrence définissant R, on a :

$R(2^0) = 0$ et $R(2^n) = 1 + R(2^{n-1})$, donnant une suite arithmétique, donc $R(2^n) = n$.

$R(2^0 + 1) = 1$ et $R(2^n + 1) = 2 + R(2^{n-1} + 1) + R(2^{n-1})$

$= 2 + R(2^{n-1} + 1) + n - 1$

$= R(2^{n-1} + 1) + n + 1$

donc $R(2^n + 1) = \dfrac{(n+1)(n+2)}{2}$ par récurrence

$R(2^1 - 1) = 0$ et $R(2^{n+1} - 1) = 2 + R(2^n) + R(2^n - 1)$

$= 2 + n + R(2^n - 1)$

donc $R(2^{n+1} - 1) = \dfrac{n(n+5)}{2}$ par récurrence

b) On a :

$5 \times 2^n + (-1)^n \equiv 5 \times (-1)^n + (-1)^n \bmod 3$

$\equiv 6 \times (-1)^n \bmod 3$

$\equiv 0 \bmod 3$

donc 3 divise $5 \times 2^n + (-1)^n$ et $f(n)$ est bien un entier.

On a aussi $5 \times 2^n + (-1)^n \equiv 1 \bmod 2$, donc le numérateur de $f(n)$ est impair et la division par 3 ne change pas sa parité.

Par ailleurs :

$$\frac{5\times 2^n+(-1)^n}{3}\geq\frac{5\times 2^n-1}{3}\geq\frac{5\times 2^n-2\times 2^n}{3}=2^n$$

et $$\frac{5\times 2^n+(-1)^n}{3}\leq\frac{5\times 2^n+1}{3}<\frac{5\times 2^n+2^n}{3}=2^{n+1}$$

donc $f(n)\in[\![\,2^n,\,2^{n+1}\,[\![$, donc la décomposition binaire de $f(n)$ possède $n+1$ bits.

c) Si n est pair, supérieur ou égal à 2, on a :

$$f(n)=\frac{5\times 2^n+1}{3}$$

$$\frac{f(n)-1}{2}=\frac{5\times 2^{n-1}-1}{3}=f(n-1)$$ car $n-1$ est impair

$$\frac{f(n)+1}{2}=\frac{5\times 2^{n-1}+2}{3}=2\times\frac{5\times 2^{n-2}+1}{3}=2f(n-2)$$ car $n-2$ est pair

donc $s(f(n))=s(f(n-1))+s(2f(n-2))$ avec 2 appels récursifs

$=s(f(n-1))+s(f(n-2))$ avec 1 appel récursif supplémentaire

donc $R(f(n))=R(f(n-1))+R(f(n-2))+3$

qu'on peut aussi écrire sous la forme :

$R(f(n))+3=(R(f(n-1))+3)+(R(f(n-2))+3)$

De même, si n est impair, supérieur ou égal à 3 :

$$f(n)=\frac{5\times 2^n-1}{3}$$

$$\frac{f(n)-1}{2}=\frac{5\times 2^{n-1}-2}{3}=2\times\frac{5\times 2^{n-2}-1}{3}=2f(n-2)$$ car $n-2$ est impair

$$\frac{f(n)+1}{2}=\frac{5\times 2^{n-1}+1}{3}=f(n-1)$$ car $n-1$ est pair

et on conclut de la même façon.

Ainsi, les suites $(s(f(n))$ et $(R(f(n)+3)$, et donc aussi $(\frac{R(f(n)+3)}{2})$ vérifient la même relation de récurrence que la suite de Fibonacci.

Comme on a $s(f(1))=s(3)=2$ et $s(f(2))=s(7)=3$ qui sont deux termes successifs de la suite de Fibonacci, la suite $(s(f(n))$ est la suite de Fibonacci (en démarrant à $2=F(3)$, $3=F(4)$, ...)

Comme on a $\frac{R(f(1))+3}{2}=3$ et $\frac{R(f(2))+3}{2}=5$ qui sont aussi deux termes successifs de la suite de Fibonacci, la suite $(\frac{R(f(n))+3}{2})$ est aussi égal à la suite de Fibonacci (en démarrant à $3=F(4)$, $5=F(5)$, ...). Plus précisément, posant $\varphi=\frac{1+\sqrt{5}}{2}$ et $\overline{\varphi}=\frac{1-\sqrt{5}}{2}$:

$$R(f(n))=\frac{2}{\sqrt{5}}(\varphi^{n+3}-\overline{\varphi}^{n+3})-3$$

Il en résulte que $R(f(n))$ est un $O(\varphi^n)$.

Si, par l'absurde, $R(n)$ était un $O(\ln(n))$, $R(f(n))$ serait un $O(\ln(f(n))$. Mais $2^n\leq f(n)<2^{n+1}$, donc $\ln(f(n))=O(n)$ et on aurait $R(f(n))=O(n)$, ce qui n'est pas. Donc $R(n)$ n'est pas un $O(\ln(n))$. La suite de Stern est plus longue à calculer que la suite de Thue.

d) On procède par récurrence sur n.

Pour $n=1$, l'ensemble $\{R(k),\,2\leq k<4\}$ vaut $\{1,3\}$ dont le maximum est $3=R(3)=R(f(1))$.

Pour $n=2$, l'ensemble $\{R(k),\,4\leq k<8\}$ vaut $\{2,6,4,7\}$ dont le maximum est $7=R(7)=R(f(2))$.

Supposons la propriété vraie jusqu'à un rang $n-1\geq 2$. Soit k élément de $[\![\,2^n,\,2^{n+1}-1\,[\![$.

❑ Si k est pair, alors :

$R(k) = 1 + R(\frac{k}{2})$ avec $\frac{k}{2} \in [\![2^{n-1}, 2^n - 1 [\![$

$\leq 1 + R(f(n-1))$ d'après l'hypothèse de récurrence

$\leq R(f(n-1)) + R(f(n-2)) + 3$ a fortiori

$\leq R(f(n))$

❑ Si k est impair différent de $2^{n+1} - 1$, alors :

$R(k) = 2 + R(\frac{k+1}{2}) + R(\frac{k-1}{2})$ avec $\frac{k \pm 1}{2} \in [\![2^{n-1}, 2^n - 1 [\![$

De plus, l'un des deux termes $\frac{k+1}{2}$ et $\frac{k-1}{2}$ est pair. Si c'est $\frac{k+1}{2}$ par exemple, on a :

$R(k) = 3 + R(\frac{k+1}{4}) + R(\frac{k-1}{2})$ avec $\frac{k+1}{4} \in [\![2^{n-2}, 2^{n-1} - 1 [\![$

donc $R(k) \leq 3 + R(f(n-2)) + R(f(n-1))$ d'après l'hypothèse de récurrence aux rangs $n-1$ ou $n-2$.

$\leq R(f(n))$

On obtient la même inégalité si c'est $\frac{k-1}{2}$ qui est pair.

❑ Si $k = 2^{n+1} - 1$, alors :

$R(2^{n+1} - 1) = \frac{n(n+5)}{2}$ d'après a)

$R(2^n - 1) = \frac{(n-1)(n+4)}{2} \leq R(f(n-1))$ d'après l'hypothèse de récurrence au rang $n-1$

$R(2^{n-1} - 1) = \frac{(n-2)(n+3)}{2} \leq R(f(n-2))$ d'après l'hypothèse de récurrence au rang $n-2$

Or, on vérifiera que l'inégalité $\frac{n(n+5)}{2} \leq 3 + \frac{(n-1)(n+4)}{2} + \frac{(n-2)(n+3)}{2}$ est équivalente à l'inégalité $n^2 - n - 4 \geq 0$ qui est vérifiée pour $n \geq 3$. Donc :

$R(2^{n+1} - 1) \leq 3 + R(2^n - 1) + R(2^{n-1} - 1) \leq 3 + R(f(n-1)) + R(f(n-2)) \leq R(f(n))$

On a montré que, pour tout k élément de $[\![2^n, 2^{n+1} - 1 [\![$:

$R(k) \leq R(f(n))$

Comme $f(n)$ est lui-même élément de $[\![2^n, 2^{n+1} - 1 [\![$, cela prouve que :

$R(f(n)) = \text{Max } \{R(k), 2^n \leq k < 2^{n+1}\}$

e) Pour n tendant vers $+\infty$, notons m l'entier tel que $2^m \leq n < 2^{m+1}$, de sorte que :

$m \leq \frac{\ln(n)}{\ln(2)}$

et $R(n) \leq \text{Max } \{R(k), 2^m \leq k < 2^{m+1}\}$

$\leq R(f(m)) \sim \frac{2}{\sqrt{5}} \varphi^{m+3} \leq \frac{2\varphi^3}{\sqrt{5}} \varphi^{\ln(n)/\ln(2)} = \frac{2\varphi^3}{\sqrt{5}} n^{\ln(\varphi)/\ln(2)}$

La complexité en temps de calcul de $s(n)$ est négligeable devant n mais peut néanmoins croître comme une puissance de n proche de 0,7.

Sol.10) a) Pour construire de tels mots, on peut procéder comme suit :

décider combien de lettres apparaîtront deux fois. Soit k ce nombre, pouvant varier entre 0 et $\lfloor \frac{n}{2} \rfloor$, $\lfloor \ \rfloor$ désignant la partie entière.

choisir k lettres parmi n : $\binom{n}{k}$ choix possibles

choisir 2 places parmi n dans le mot pour la première des lettres précédentes, puis 2 places parmi $n-2$ pour la lettre suivante, puis 2 places parmi $n-4$ pour la suivante, etc. jusqu'à la k-ème pour laquelle on choisit 2 places parmi $n-2k+2$. Le nombre de choix possibles est :

$$\frac{n(n-1)}{2} \times \frac{(n-2)(n-3)}{2} \times \ldots \times \frac{(n-2k+2)(n-2k+1)}{2} = \frac{n!}{2^k(n-2k)!}$$

il reste $n-2k$ places à remplir dans le mot avec $n-k$ lettres possibles. Choisir une lettre parmi $n-k$ pour la première place, puis une lettre parmi $n-k-1$ pour la deuxième place, etc. jusqu'à la dernière place qu'on remplit avec une lettre parmi $k+1$. Le nombre de choix possibles est $\frac{(n-k)!}{k!}$.

Il en résulte que :

$$v_n = \sum_{k\geq 0} \binom{n}{k} \frac{n!}{2^k(n-2k)!} \frac{(n-k)!}{k!} = n! \sum_{k\geq 0} \frac{1}{(k!)^2} \frac{n!}{2^k(n-2k)!} = n! \sum_{k\geq 0} \frac{1}{2^k} \binom{2k}{k} \binom{n}{2k}$$

b) v_n est inférieur ou égal à n^n, nombre de mots de n lettres sur un alphabet de n lettres. De plus, la formule de Stirling donne : $n! \sim n^n e^{-n}\sqrt{2\pi n}$, donc $0 \leq \frac{v_n}{n!} \leq \frac{n^n}{n!} \sim \frac{e^n}{\sqrt{2\pi n}}$. On déduit de cette majoration que le rayon de convergence de la série génératrice est supérieur ou égal à $\frac{1}{e}$ (voir le chapitre sur les séries entières L2/SERIENTR.PDF. On a également besoin de ce chapitre pour les deux sommations de séries entières classiques utilisées ci-dessous). Dans son disque de convergence, la série est absolument convergente et on peut permuter les signes de sommation (voir le chapitre L2/SERIES.PDF pour la notion de famille sommable) :

$$H(x) = \sum_{n=0}^{\infty} \sum_{k\geq 0} \frac{1}{2^k} \binom{2k}{k} \binom{n}{2k} x^n$$

$$= \sum_{k\geq 0} \frac{1}{2^k} \binom{2k}{k} \sum_{n\geq 2k}^{\infty} \binom{n}{2k} x^n$$

$$= \sum_{k\geq 0} \frac{1}{2^k} \binom{2k}{k} \frac{x^{2k}}{(1-x)^{2k+1}} \qquad \text{car } \frac{x^p}{(1-x)^{p+1}} = \sum_{n=p}^{\infty} \binom{n}{p} x^n$$

$$= \frac{1}{1-x} \sum_{k\geq 0} \frac{1}{2^{2k}} \binom{2k}{k} \left(\frac{2x^2}{(1-x)^2}\right)^k$$

$$= \frac{1}{1-x} \frac{1}{\sqrt{1-\frac{2x^2}{(1-x)^2}}} \qquad \text{car } \frac{1}{\sqrt{1-u}} = \sum_{n=0}^{\infty} \frac{1}{2^{2n}} \binom{2n}{n} u^n$$

$$= \frac{1}{\sqrt{1-2x-x^2}}$$

c) On a $H'(x) = \frac{1+x}{(1-2x-x^2)^{3/2}} = \frac{(1+x)H(x)}{1-2x-x^2}$

$\Rightarrow \quad (1-2x-x^2)H'(x) = (1+x)H(x)$

$$\Rightarrow \qquad (1-2x-x^2)\sum_{n=1}^{\infty}\frac{v_n}{(n-1)!}x^{n-1} = (1+x)\sum_{n=0}^{\infty}\frac{v_n}{n!}x^n$$

$$\Rightarrow \qquad \sum_{n=0}^{\infty}\frac{v_{n+1}}{n!}x^n - 2\sum_{n=1}^{\infty}\frac{v_n}{(n-1)!}x^n - \sum_{n=2}^{\infty}\frac{v_{n-1}}{(n-2)!}x^n = \sum_{n=0}^{\infty}\frac{v_n}{n!}x^n + \sum_{n=1}^{\infty}\frac{v_{n-1}}{(n-1)!}x^n$$

Pour $n = 0$, on obtient $v_1 = v_0$.
Pour $n = 1$, on obtient $v_2 - 2v_1 = v_1 + v_0$ soit $v_2 = 3v_1 + v_0$.
Pour $n \geq 2$, on obtient :

$$\frac{v_{n+1}}{n!} - \frac{2v_n}{(n-1)!} - \frac{v_{n-1}}{(n-2)!} = \frac{v_n}{n!} + \frac{v_{n-1}}{(n-1)!}$$

$$\Leftrightarrow \qquad v_{n+1} - 2nv_n - n(n-1)v_{n-1} = v_n + nv_{n-1}$$

$$\Leftrightarrow \qquad v_{n+1} = (2n+1)v_n + n^2 v_{n-1}$$

d) On peut prouver que $\frac{\pi}{4} = \sum_{n=0}^{\infty}\frac{(-1)^n}{(n+1)u_n u_{n+1}}$. (J'ai une démonstration merveilleuse de cette formule, mais elle est trop longue pour figurer dans la marge).

Sol.11) On a ici :

B = 1
$f(1) = 1$
$g(p) = (p+1)/2$ division entière
$h(p, z) = \begin{cases} p + 1 - 2z \text{ si } p \text{ est pair} \\ p + 2 - 2z \text{ si } p \text{ est impair} \end{cases}$

L'algorithme itératif avec pile s'écrit alors :

```
fonction F(p)
    L ← [ ]
    tantque (p ≠ 1) faire
        L.empile(p)
        p ← (p + 1)/2          # division entière
    finfaire
    y ← 1;
    tant que L ≠ [ ] faire
        temp ← L.depile()
        si temp pair alors
            y ← temp + 1 - 2*y
        sinon
            y ← temp + 2 - 2*y
        finsi
    finfaire
    retourner(y)
```

Sol.12) a) –

b) Toutes les répartitions étant équiprobables, pour tout m, la probabilité que le pivot x choisi soit le m-ème élément du tableau est $\frac{1}{n}$. On aura alors $m - 1$ éléments dans Tinf et $n - m$ dans Tsup. Si $m = k$, alors le pivot x est l'élément cherché. Si $m - 1 \geq k$, l'élément cherché est dans Tinf. Si $k > m$, l'élément cherché est dans Tsup et en serait le $(k - m)$-ème élément si ce sous-tableau était trié. Donc, en appliquant le théorème de l'espérance totale :

$$C(n, k) = n + \frac{1}{n} \left(\sum_{m=k+1}^{n} C(m-1, k) + \sum_{m=1}^{k-1} C(n-m, k-m) \right)$$

le premier n dans la somme de droite correspondant aux n comparaisons que l'on fera entre les éléments du tableau avec le pivot x pour créer les deux sous-tableaux `Tinf` et `Tsup`. Par ailleurs, $C(1, k) = 0$.

c) La relation $C(n, k) \leq 4n$ est vérifiée pour $n = 1$. Supposons qu'elle soit vérifiée jusqu'à un rang $n - 1$. On a alors :

$$\begin{aligned} C(n, k) &\leq n + \frac{1}{n} \left(\sum_{m=k+1}^{n} 4(m-1) + \sum_{m=1}^{k-1} 4(n-m) \right. \\ &\leq n + \frac{1}{n} \left(2(n-k)(k+n-1) \right. + 2(k-1)(2n-k)) \\ &\leq n + \frac{1}{n} \left(2n^2 - 2n - 2k^2 + 2k + 4(k-1)n - 2k^2 + 2k \right) \\ &\leq n + \frac{1}{n} \left(2n^2 - 6n + 4nk - 4k^2 + 4k \right) \end{aligned}$$

Or la fonction $x \rightarrow 2n^2 - 6n + 4nx - 4x^2 + 4x$ admet un maximum pour $x = \frac{n+1}{2}$, où elle prend la valeur $3n^2 - 4n + 1$. Donc :

$$C(n, k) \leq n + \frac{3n^2 - 4n + 1}{n} = \frac{4n^2 - 4n + 1}{n} \leq 4n$$

On a donc montré que $C(n, k)$ est un $O(n)$, uniformément en k.

◆