RÉCURSIVITÉ

B. Landelle

Table des matières

Les modules scientifique et graphique seront importés avec leurs alias habituels :

```
import numpy as np, matplotlib.pyplot as plt
```

I Présentation

1 Définitions

Définition 1. *Une fonction* récursive *est une fonction qui fait appel à elle-même. Cet appel est dénommé* appel récursif *ou* récursion.

Remarque : On définit sur le même principe la notion d'algorithme *récursif*.

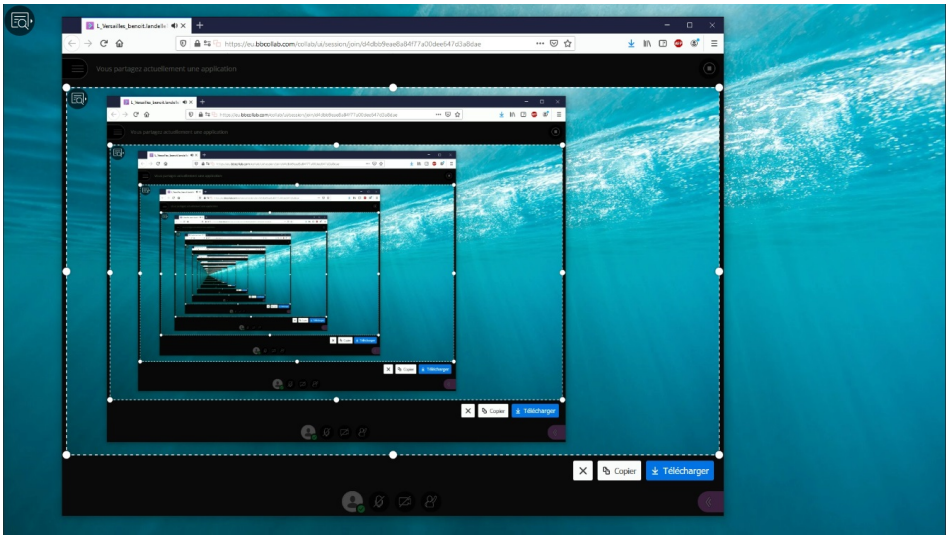


FIGURE 1 – Écran qui partage l'écran qui partage l'écran ...

Exemple : Un exemple très classique et très simple est celui de l'application factorielle. On a

$$\forall n \in \mathbb{N} \qquad n! = \begin{cases} n \times (n-1)! & \text{si } n \geqslant 1 \\ 1 & \text{si } n = 0 \end{cases}$$

```
def fact(n):
    """fact(n:int)->int
    renvoie n!"""
    if n==0:
        return 1
    else:
        return n*fact(n-1)
```

Proposition 1. *Une fonction récursive est constituée de deux alternatives fondamentales :*
- *un (ou plusieurs)* cas de base *sans appel récursif ;*
- *le* cas de propagation *avec un (ou plusieurs) appel récursif.*

Remarque : Le parallèle avec une suite définie $(u_n)_n$ définie par une relation de récurrence est flagrant : le cas de base correspond à la valeur initiale de la suite et le cas de propagation correspond à la relation de récurrence de la suite.

Exemple : Considérons la fonction `fact` de calcul de factorielle présentée précédemment.

```
def fact(n):
    if n==0:
        return 1                # cas de base
    else:
        return n*fact(n-1)      # cas de propagation avec appel récursif
```

La définition de 0! est le cas de base qui garantit la terminaison de la fonction et la relation de $n \times (n-1)!$ est le cas de propagation avec l'appel récursif à l'application factorielle.

Remarque : Le cas de base correspond à l'arrêt des appels récursifs. Si on omet ce cas, la fonction va s'appeler indéfiniment. De même, si le cas de propagation ne correspond pas à la monotonie du cas courant vers le cas de base (décroissante en général), la fonction s'appellera indéfiniment.

```
def fact_inf(n):
    return n*fact_inf(n-1)          # factorielle sans cas de base
```

```
def fact_cr(n):
    if n==0:
        return 1
    else:
        return fact_cr(n+1)//(n+1)   # propagation croissante
                                     # incohérente avec cas de base
```

Sur ces exemples, la fonction s'appelle indéfiniment. Le langage python ne détecte pas d'erreurs de syntaxe dans l'écriture de ces fonctions sans cas de base. En revanche, leurs appels provoquent des erreurs :

- `RuntimeError: maximum recursion depth exceeded` ;
- `RuntimeError: maximum recursion depth exceeded in comparison` ;

messages d'erreurs sur lesquels nous reviendrons ultérieurement.

Définition 2. *Si plusieurs fonctions s'appellent l'une l'autre, on parle de fonctions* mutuellement récursives *ou* récursives croisées.

Exemple : Considérons les suites $(u_n)_n$ et $(v_n)_n$ définies par $u_0 = v_0 = 1$ et pour tout n entier

$$\begin{cases} u_{n+1} = u_n + v_n \\ v_{n+1} = u_n - v_n \end{cases}$$

On peut facilement coder ces suites récursivement :

```
def u(n):
    if n==0:
        return 1
    else:
        return u(n-1)+v(n-1)

def v(n):
    if n==0:
        return 1
    else:
        return u(n-1)-v(n-1)
```

Exercice : Écrire une fonction récursive réalisant le calcul de $\binom{n}{k}$ avec k et n entiers et la convention $\binom{n}{k}=0$ si $k>n$.

Corrigé : On utilise la relation

$$\forall n \geqslant k \geqslant 1 \qquad \binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n}{k} \times \frac{(n-1)!}{(k-1)!((n-1)-(k-1))!} = \frac{n}{k} \times \binom{n-1}{k-1}$$

```
def binom(n,k):
    """binom(n:int,k:int)->int
    renvoie coefficient binomial k parmi n"""
    if  k>n:
        return 0
    elif k==0:
        return 1
    else:
        return n*binom(n-1,k-1)//k
```

Exercice : Écrire une fonction récursive `renverse(texte)` qui renverse la chaîne de caractères `texte`.

Corrigé : On saisit :

```
def renverse(texte):
    """renverse(texte:str)->str
    renvoie texte[::-1]"""
    if len(texte)<2:
        return texte
    else:
        return texte[-1]+renverse(texte[1:-1])+texte[0]
```

Exercice : Écrire une fonction récursive `dicho(f,a,b,eps)` qui détermine par dichotomie une racine de $f \in \mathscr{C}^0([a\,;b],\mathbb{R})$ vérifiant $f(a)f(b) \leqslant 0$.

Corrigé : On saisit :

```
def dicho(f,a,b,eps):
    """dicho(f:func,a:float,b:float,eps:eps)->float
    renvoie une racine approchée de f entre a et b à eps près"""
    c=(a+b)/2
    if b-a<eps:
        return c
    else:
        if f(a)*f(c)<=0:
            return dicho(f,a,c,eps)
        else:
            return dicho(f,c,b,eps)
```

On expérimente :

```
>>> dicho(lambda t:t**2-2,0,2,1e-5)
1.4142112731933594
>>> dicho(np.sin,2,4,1e-5)
3.141590118408203
```

Avec la récursivité, on peut générer des images complexes. Si l'on souhaite construire un arbre qui démarre par un tronc puis qui se sépare en deux branches de taille $\frac{2}{3}$ de la taille du tronc, orientées de $\pm\frac{\pi}{6}$ par rapport à l'axe du tronc avec répétition de ce mécanisme de duplication, on saisit :

```
def arbre(n):
    """arbre(n:int)->list
    Génère une liste de n listes où la k-ième liste contient
    toutes les branches générées lors de la k-ième récursion"""
    if n==0:
        return [[[0,1j]]]
    else:
        res=arbre(n-1)
        aux=[]
        for branche in res[-1]: # on parcourt la dernière liste de branches
            a,b=branche
            c=b+(b-a)*2/3*np.exp(1j*np.pi/6)
            d=b+(b-a)*2/3*np.exp(-1j*np.pi/6)
            aux.append([b,c])
            aux.append([b,d])
        res.append(aux)
        return res
```

On obtient la figure suivante :

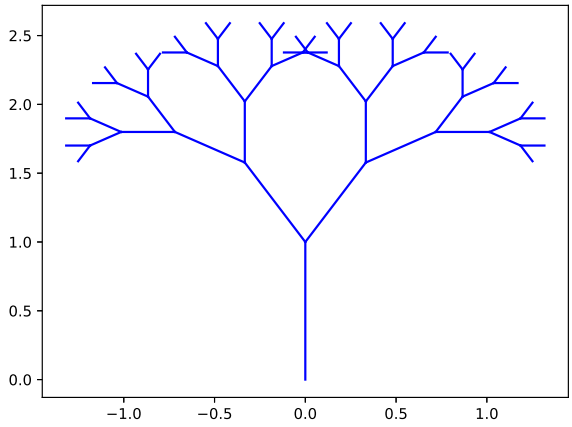


FIGURE 2 – Arbre construit récursivement

Pour cette construction récursive, le plus simple est de travailler sur des nombres complexes, affixes de points ou vecteurs de $\mathbb{R}^2$. Dans ce qui suit, on confond affixe et point/vecteur de $\mathbb{R}^2$. La première branche est définie par les point $(0,0)$ et $(0,1)$ d'affixes respectives `0` et `1j` (le i en mathématique). Ensuite, pour la propagation de la construction, étant donnée une branche d'extrémités a et b, on construit deux nouvelles extrémités c et d pour les deux branches issues de b avec la définition

$$c = b + \frac{2}{3}(b-a)\mathrm{e}^{\frac{\mathrm{i}\pi}{6}} \qquad d = b + \frac{2}{3}(b-a)\mathrm{e}^{-\frac{\mathrm{i}\pi}{6}}$$

La construction de c consiste à translater le point b d'un vecteur construit par rotation d'un angle de $\dfrac{\pi}{6}$ du vecteur $b-a$ multiplié par un facteur $\dfrac{2}{3}$ et de même pour d avec un angle de $-\dfrac{\pi}{6}$. La récursivité fait le reste.

En modifiant très légèrement le code ci-dessus, par exemple pour avoir trois branches naissantes depuis une extrémité, de longueurs différentes et réparties non symétriquement, on peut générer des arbres beaucoup plus réalistes.

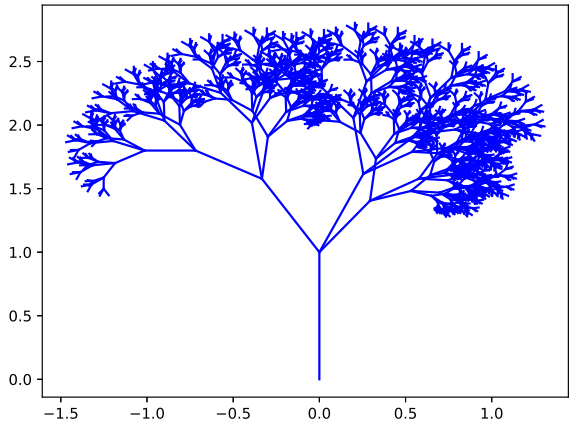


FIGURE 3 – Arbre construit récursivement

2 Problème de la terminaison

La question de la terminaison d'une fonction récursive peut être plus élaborée qu'il n'y paraît. Considérons la *suite de Syracuse* d'un entier N définie par

$$u_0 = \mathrm{N} \quad \text{et} \quad \forall n \in \mathbb{N} \qquad u_{n+1} = \begin{cases} \dfrac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{si } u_n \text{ est impair} \end{cases}$$

La conjecture dite *de Syracuse* affirme que pour tout entier $\mathrm{N} > 0$, il existe un indice entier $n \geqslant 1$ tel que $u_n = 1$. Considérons alors la fonction python suivante :

```
def syracuse(n):
    """syracuse(n:int)->None
    Calcule la suite de Syracuse démarrant à n finissant à 1"""
    if n!=1:
        if n%2==0:
            syracuse(n//2)
        else:
            syracuse(3*n+1)
```

La preuve de la terminaison de cette fonction équivaut à la démonstration de la conjecture de Syracuse, toujours irrésolue à ce jour.

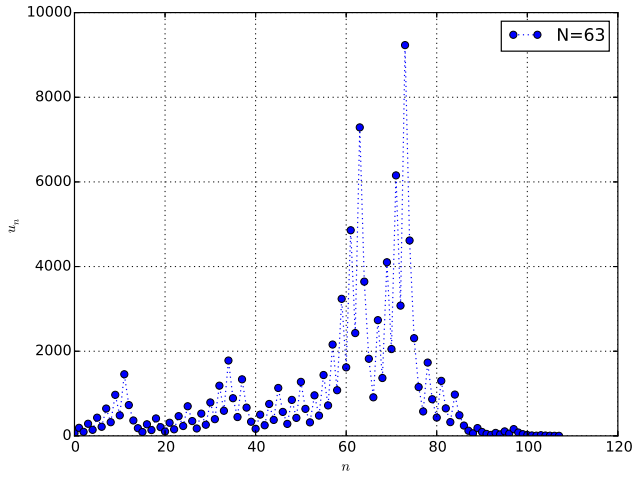


FIGURE 4 – Temps de « vol » d'une suite de Syracuse

Pourrait-on imaginer qu'un programme puisse prouver la terminaison d'un autre programme ? La réponse est donnée par un des théorèmes de Gödel [1].

Théorème 1 (Gödel). *Il n'existe pas de programme qui détermine si un programme quelconque se termine.*

Démonstration. Supposons qu'on dispose d'un programme `termine(f)` d'argument un autre programme `f` et qui renvoie `True` si `f` se termine et `False` sinon. Considérons alors le code suivant :

1. Kurt Gödel, 1906-1978, mathématicien austro-américain, célèbre notamment pour son théorème sur l'incomplétude des systèmes formels.

```
def f():
    while termine(f):
        print("on boucle")
```

Le programme `f` se termine-t-il ? Si oui, alors `termine(f)` renvoie `True` ce qui rend la boucle infinie et contredit la terminaison de `f`. Si non, alors `termine(f)` renvoie `False` ce qui provoque une sortie de la boucle et donc une terminaison de `f`. On ne peut donc pas décider de la terminaison de ce programme ce qui illustre *l'indécidabilité* du problème d'arrêt, notion majeure en informatique fondamentale. □

3 Apparente facilité et intérêt

Tous les exemples valides de fonctions récursives de la première section pourraient être codés de manière itérative avec des boucles inconditionnelles `for`. Toutefois, ce travail de codage itératif demanderait un effort supérieur. Cette grande facilité à coder récursivement des fonctions dont la définition se prête à ce mode de calcul est clairement un atout. Cependant, nous verrons que cette facilité d'écriture peut avoir un coût non négligeable en complexité spatiale et/ou temporelle.

Exemple : On définit la suite $(u_n)_n$ par $u_0 = 1$ et $u_{n+1} = \sum_{i=0}^{n} u_i u_{n-i}$ (nombres de *Catalan*). La recherche d'une formule explicite de u_n ou même plus simplement l'écriture d'un algorithme itératif n'est pas immédiate. Tandis que l'écriture récursive découle naturellement de la définition de la suite.

```
def suite(n):
    """suite(n:int)->int
    Calcule le nombre de Catalan d'indice n"""
    if n==0:
        return 1
    else:
        s=0
        for i in range(n):
             s+=suite(i)*suite(n-1-i)
        return s
```

Tous les exemples de fonctions récursives présentées jusqu'à présent (exceptée `syracuse`) peuvent être codés avec des boucles inconditionnelles `for`. La question qui s'impose assez naturellement est : existe-t-il des fonctions récursives qu'on ne puisse coder avec des boucles `for` ?

Exemple célèbre : la fonction d'Ackermann

La fonction d'Ackermann est définie comme suit

$$\mathrm{A}(m,n) = \begin{cases} n+1 & \text{si } m = 0 \\ \mathrm{A}(m-1,1) & \text{si } m > 0 \text{ et } n = 0 \\ \mathrm{A}(m-1,\mathrm{A}(m,n-1)) & \text{si } m > 0 \text{ et } n > 0 \end{cases}$$

```
def A(m,n):
    if m==0:
        return n+1
    elif m>0 and n==0:
        return A(m-1,1)
    else:
        return A(m-1,A(m,n-1))
```

Elle constitue l'exemple le plus célèbre de fonction récursive dite *non récursive primitive*. Ainsi, on peut démontrer (mais ce n'est pas simple) que la fonction d'Ackermann ne peut se coder avec des boucles `for`.

4 Empilement des appels récursifs

Devant l'apparente simplicité de certains codes récursifs, on peut légitimement se demander : comment sont gérés les appels successifs de la fonction par elle-même ?

La réponse est simple et élégante : le programme utilise une pile. Les appels récursifs sont empilés jusqu'à atteindre le(s) cas de base puis sont dépilés pour réaliser le calcul de proche en proche.

Exemple : Considérons l'exemple de la fonction `fact`.

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

Si on demande à l'ordinateur de calculer `fact(3)`, il procède à l'empilement des appels récursifs jusqu'à atteindre `fact(0)` le cas de base.

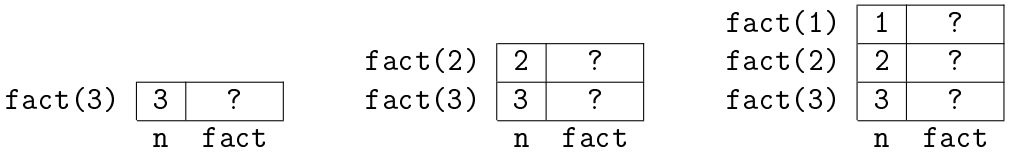


Schéma 1 - Empilement des appels récursifs

Une fois le cas de base atteint, il ne reste plus qu'à dépiler les appels et à effectuer les calculs successifs.

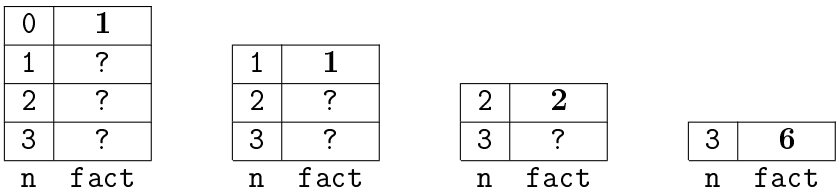


Schéma 2 - Dépilement des appels récursifs

⚠**Avertissement :** La pile utilisée pour les empilements d'appels est une pile finie. Ainsi, un programme récursif qui boucle indéfiniment provoquera immanquablement un dépassement de pile et donc un message d'erreur comme ceux précédemment cités :

```
RuntimeError: maximum recursion depth exceeded
```

On peut connaître la hauteur de la pile de récursion fixée par défaut en python :

```
>>> import sys
>>> sys.getrecursionlimit()
1000
```

Définition 3. *Une fonction est dite* récursive terminale *(*tail-recursive *en anglais) si la dernière instruction exécutée est un appel simple de la fonction elle-même, sans autre opération sur cet appel.*

Exemple : Considérons la fonction `recterm` suivante :

```
def recterm(n):
    if n==0:
        print("FIN")
    else:
        print(n)
        recterm(n-1)
```

La dernière instruction est un appel simple de `recterm` ce qui en fait donc une fonction récursive terminale.

Certains langages optimisent les fonctions récursives terminales (*TCO, Tail Call Optimization*) en les exécutant comme des programmes itératifs donc sans empiler les appels récursifs. Ce n'est pas le cas de python qui traite la récursivité terminale comme une récursivité quelconque.

Expérimentation : Considérons les deux versions suivantes de calcul de factorielle :

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

def fact_term(n,k=1):
    if n==0:
        return k
    else:
        return fact_term(n-1,k*n)
```

La fonction `fact_term(n,k)` est récursive terminale. Les produits successifs de la factorielle sont réalisés dans le passage du second argument (appelé *accumulateur*) de l'appel récursif.

Remarque : L'argument `k` est facultatif pour `fact_term`. S'il est absent, sa valeur par défaut est celle mentionnée dans la déclaration de la fonction à savoir `k=1`. Ce mode de déclaration permet notamment de pouvoir appeler `fact_term` de la même façon qu'on appelle `fact`, sans que cela restreigne l'usage récursif de `fact_term`.

```
>>> fact(5)
120
>>> fact_term(5)
120
>>> fact(994)
...
RuntimeError: maximum recursion depth exceeded in comparison
>>> fact_term(994)
...
RuntimeError: maximum recursion depth exceeded in comparison
```

L'instruction `fact(994)` provoque un dépassement de pile, ce qui était prévisible mais l'instruction `fact_term(994)` également ce qui est un peu décevant. L'erreur de dépassement se produit avant 1000 récursions, la pile servant à stocker certaines informations spécifiques en plus des appels récursifs.

II Complexité et récursivité

On fait l'hypothèse (très) simplificatrice que les opérations arithmétiques sur les entiers sont à coût constant. Pour une fonction d'argument égal à n ou de taille n, on note $\mathrm{S}(n)$ sa complexité spatiale et $\mathrm{T}(n)$ sa complexité temporelle. On rencontrera des situations du type

$$\mathrm{T}(n) = a\mathrm{T}\left(n/b\right) + f(n)$$

et de même pour $\mathrm{S}(n)$. La quantité n/b s'entend indifféremment au sens $\lfloor n/b \rfloor$ ou $\lceil n/b \rceil$. Le lecteur trouvera des informations complètes et détaillées en consultant les ouvrages de référence [1] et [3] et plus précisément les parties dédiées au *Master Theorem* et aux *Divide-and-Conquer Recurrences*.

1 Complexité spatiale

Par convention, on considérera qu'un cas de base compte pour une récursion.

Proposition 2. *Pour une fonction, récursive ou non, la complexité spatiale est au plus égale à la complexité temporelle.*

Démonstration. Chaque occupation en mémoire, que ce soit l'utilisation de variables locales ou l'empilement de récursions, requiert le temps d'écriture en mémoire. La majoration s'en déduit. □

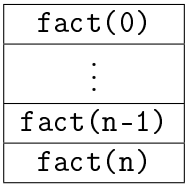
- **Factorielle**

Reprenons l'écriture récursive de l'application factorielle.

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

Complexité spatiale : La complexité spatiale $\mathrm{S}(n)$ de la fonction correspond au nombre d'appels empilés lors de son exécution. La propagation se fait en passant de n à $n-1$ et l'arrêt d'empilement correspond à $n=0$:

$$\forall n \geqslant 1 \qquad \mathrm{S}(n) = \mathrm{S}(n-1) + 1 \quad \text{et} \quad \mathrm{S}(0) = 1 \quad \Longrightarrow \quad \mathrm{S}(n) = n+1$$



La complexité spatiale de `fact` est donc en $\mathrm{O}(n)$.

Complexité temporelle : L'ordinateur empile les appels en n, $n-1$, ..., jusqu'à 0 puis les dépile tous en effectuant une multiplication à chaque étape. Notant b le coût de cette multiplication et a celui du cas de base, la complexité temporelle vérifie $\mathrm{T}(n) = a + bn$ d'où $\mathrm{T}(n) = \mathrm{O}(n)$.

Proposition 3. *Pour une fonction récursive de complexité temporelle ou spatiale vérifiant*

$$\mathrm{U}(n) = \mathrm{U}(n-1) + O(1)$$

alors

$$\mathrm{U}(n) = O(n)$$

Démonstration. Soit n entier. On a

$$\mathrm{U}(n) = \mathrm{U}(0) + \sum_{k=1}^{n} [\mathrm{U}(k) - \mathrm{U}(k-1)] = \mathrm{U}(0) + \sum_{k=1}^{n} \mathrm{O}(1) = \mathrm{O}(n)$$

□

Remarque : On retrouve les résultats annoncés pour les complexités spatiale et temporelle de `fact`.

Après importation du module `inspect`, on peut, avec l'instruction `len(inspect.stack(0))-4`, récupérer la hauteur de pile au cours des récursions :

```
import inspect
L_stack=[]

def fact_stack(n):
    global L_stack
    L_stack.append(len(inspect.stack(0))-4)
    if n==0:
        return 1
    else:
        return n*fact_stack(n-1)
```

On obtient :

```
>>> fact_stack(10)
3628800
>>> L_stack
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

• **Coefficient binomial**

```
def binom(n,k):
    if  k>n:
        return 0
    elif k==0:
        return 1
    else:
        return n*binom(n-1,k-1)//k
```

Les récursions s'arrêtent lorsque $k = 0$. C'est donc vis-à-vis de la variable k que se fait l'examen des complexités. On a $S(k) = S(k-1) + 1$ et $T(k) = T(k-1) + O(1)$ d'où

$$S(k) = O(k) \quad \text{et} \quad T(k) = O(k)$$

Exercice : Considérons le codage itératif suivant de l'application factorielle.

```
def fact_iter(n):
    res=1
    for k in range(2,n+1):
        res*=k
    return res
```

Cette version est-elle préférable à la version récursive ?

Corrigé : Sa complexité spatiale est en $O(1)$ et la complexité temporelle en $O(n)$. Cette version est donc préférable à la version récursive même si sa conception et son étude demande plus d'effort.

Proposition 4. *Pour une fonction récursive de complexité temporelle ou spatiale vérifiant*

$$U(n) = U(n-1) + O(n)$$

alors

$$U(n) = O(n^2)$$

Démonstration. Soit n entier. Par sommation des relations de comparaison, il vient

$$U(n) = U(0) + \sum_{k=1}^{n} [U(k) - U(k-1)] = U(0) + \sum_{k=1}^{n} O(k) = U(0) + O\left(\sum_{k=1}^{n} k\right) = O(n^2)$$

□

Exercice : Complexité spatiale et temporelle de `renverse`.

```
def renverse(texte):
    if len(texte)<2:
        return texte
    else:
        return texte[-1]+renverse(texte[1:-1])+texte[0]
```

Corrigé : Lors d'une récursion, on passe d'un argument de taille n à un argument de taille $n-2$ qu'il faut créer d'où une relation $\mathrm{S}(n) = \mathrm{S}(n-2) + \mathrm{O}(n-2) = \mathrm{S}(n-2) + \mathrm{O}(n)$. Par suite, pour n entier

$$\mathrm{S}(2n) = \mathrm{S}(0) + \sum_{k=1}^{n} [\mathrm{S}(2k) - \mathrm{S}(2(k-1))] = \mathrm{S}(0) + \sum_{k=1}^{n} \mathrm{O}(k) = \mathrm{O}(n^2)$$

et on procède de même avec $\mathrm{S}(2n+1)$. Il s'ensuit $\mathrm{S}(n) = \mathrm{O}(n^2)$. Pour la complexité temporelle, lors de chaque récursion, il faut tenir compte en plus d'une concaténation de chaîne d'un coût en $\mathrm{O}(n)$ pour une chaîne de taille $n-2$ concaténée avec deux caractères. Par conséquent, la complexité temporelle $\mathrm{T}(n)$ suit la même relation et donc $\mathrm{T}(n) = \mathrm{O}(n^2)$.

Remarque : En admettant la croissance de la complexité spatiale, on déduit le cas impair du cas pair avec $\mathrm{S}(2n) \leqslant \mathrm{S}(2n+1) \leqslant \mathrm{S}(2(n+1))$.

Exercice : Écrire une fonction `binaire(n)` qui renvoie une liste de chaînes de caractères de tous les nombres binaires de taille n. Par exemple, l'instruction `binaire(2)` doit renvoyer `['00', '10', '01', '11']`. Estimer les complexités temporelle et spatiale de `binaire`.

Corrigé : L'idée récursive consiste à chercher les nombres binaires de taille $n-1$ et à concaténer ceux-ci avec un `"0"` ou `"1"` pour avoir tous les nombres binaires de taille n. On saisit :

```
def binaire(n):
    """binaire(n:int)->list
    Renvoie la liste des mots binaires à n chiffres"""
    if n==0:
        return [""]
    else:
        liste=binaire(n-1)
        return [mot+"0" for mot in liste]+[mot+"1" for mot in liste]
```

La fonction `binaire` renvoie une liste. Lors de chaque récursion, la taille de la liste double et la taille des mots qui la composent est augmentée de 1. Ainsi, après l'appel `binaire(n-1)`, on génère deux listes de tailles 2^{n-1} d'où 2^n mots en tout chacun de taille n. Par conséquent, les complexités spatiale et temporelle vérifient une relation de la forme

$$\mathrm{U}(n) = \mathrm{U}(n-1) + \mathrm{O}(n2^n)$$

On trouve

$$\mathrm{U}(n) = \mathrm{U}(0) + \sum_{k=1}^{n} [\mathrm{U}(k) - \mathrm{U}(k-1)] = \mathrm{U}(0) + \sum_{k=1}^{n} \mathrm{O}(k2^k) = \mathrm{O}(n2^n)$$

Ce sont des complexités catastrophiques !

2 Complexité logarithme

• Exponentiation rapide

L'algorithme d'exponentiation rapide, dont l'écriture itérative est non triviale et requiert d'avoir bien compris la représentation binaire, s'écrit très simplement en récursif. Pour $(x,n)\in\mathbb{R}\times\mathbb{N}$, il suffit de remarquer

$$x^n=\begin{cases}\left(x^{\frac{n}{2}}\right)^2 & \text{si } n \text{ pair}\\ x\times\left(x^{\frac{n-1}{2}}\right)^2 & \text{si } n \text{ impair}\end{cases}$$

Selon la parité de n, la quantité $\frac{n}{2}$ pour n pair et $\frac{n-1}{2}$ pour n impair est le quotient de la division euclidienne de n par 2. Ainsi, on en déduit le codage récursif suivant :

```
def expo_rec(x,n):
    """expo_rec(x:float,n:int)->float
    Renvoie x**n calculé récursivement par exponentiation rapide"""
    if n==0:
        return x**0              # cas de base avec exposant nul
    else:
        r=expo_rec(x,n//2)**2    # propagation avec le quotient de n par 2
        if n%2==0:               # distinction selon la parité
            return r
        else:
            return x*r
```

Rappelons l'écriture binaire de n pour étudier les complexités de l'algorithme précédent. On a

$$n=\langle d_{p-1},\dots,d_0\rangle=\sum_{i=0}^{p-1}d_i2^i \quad\text{avec}\quad d_i\in\{0,1\} \qquad d_{p-1}=1 \quad\text{et}\quad p=\lfloor\log_2(n)\rfloor+1$$

L'opération de propagation `n//2` consiste en la perte du bit de poids faible d'où

$$n=\langle d_{p-1},\dots,d_0\rangle \qquad n//2=\langle d_{p-1},\dots,d_1\rangle \qquad \underbrace{(\dots(n//2)\dots)//2}_{p-1 \text{ divisions}}=\langle d_{p-1}\rangle=1$$

Ainsi, au bout de p appels récursifs, on arrive à un exposant nul.

Complexité spatiale : Le programme réalise un empilement de ces p appels et possède donc une complexité spatiale en $\mathrm{O}(\log(n))$.

Complexité temporelle : Pour évaluer celle-ci, on s'intéresse également au nombre d'appels. En effet, à chaque appel récursif, on réalise au moins un calcul de carré et éventuellement, selon la parité de l'exposant en cours, une multiplication. On en déduit ici aussi une complexité temporelle en $\mathrm{O}(\log(n))$.

Remarque : On fait l'hypothèse très simplificatrice que la multiplication est à coût constant. Si on travaille avec de « petits » entiers ou avec des flottants dont la plage de définition est fixée, alors la multiplication est effectivement à coût constant. Sorti de ce contexte, le résultat devient faux.

Proposition 5. *Étant donné un algorithme, notant* $\mathrm{R}(n)$ *le nombre de récursions, si celui-ci vérifie la relation*

$$\mathrm{R}(n) = \mathrm{R}\left(n//2\right) + 1$$

alors, le nombre de récursions est en $O(\log(n))$.

Démonstration. On note $n = \langle d_{p-1}, \ldots, d_0 \rangle = \sum_{i=0}^{p-1} d_i 2^i$ son écriture binaire avec $p = \lfloor \log_2(n) \rfloor + 1$, $d_i \in \{0, 1\}$ et $d_{p-1} = 1$. On a

$$\mathrm{R}(n) = \mathrm{R}(0) + \sum_{k=0}^{p-1} \left[\mathrm{R}\left(n//2^k\right) - \mathrm{R}\left(n//2^{k+1}\right)\right] = \mathrm{R}(0) + p = \mathrm{O}(\log(n))$$

□

Remarque : On a $\mathrm{R}(2^p) = 1 + \mathrm{R}(2^{p-1})$ qui suit une progression arithmétique et si on admet la croissance de R, on peut conclure avec l'encadrement $2^{p-1} \leqslant n \leqslant 2^p$.

Théorème 2. *Pour une fonction récursive de complexité temporelle ou spatiale vérifiant*

$$\mathrm{U}(n) = \mathrm{U}\left(n//2\right) + O(1)$$

alors

$$\mathrm{U}(n) = O(\log(n))$$

Démonstration. On procède exactement comme dans la preuve de la proposition précédente. □

Remarque : On retrouve les résultats annoncés pour les complexités temporelle et spatiale de l'exponentiation rapide récursive.

Exercice : Considérons les codage itératifs de l'exponentiation rapide.

```
def expo_iter(x,n):
    a,r,e=x,x**0,n
    while e>0:
        if e%2==1:
            r*=a
        a*=a;e//=2
    return r

def expo_iter(x,n):
    a,r=x,x**0
    if n>0:
        p=int(np.log2(n)+1);
        e=n
        for k in range(p):
            if e%2==1:
                r*=a
            a*=a;e//=2
    return r
```

Ces versions sont-elles préférables à la version récursive ?

Corrigé : La complexité spatiale est $O(1)$ et la complexité temporelle est $O(\log(n))$ donc là encore, la version itérative est plus performante que la version récursive.

Remarque : Une fois n'est pas coutume, la version avec la boucle conditionnelle `while` est plus simple à coder que celle avec la boucle `for` même si les deux codes font la même chose.

Remarque : Dans tous les exemples précédents, les complexités temporelles et spatiales coïncident pour les fonctions récursives. Est-ce toujours le cas ? Le paragraphe suivant apporte des éléments de réponse.

3 Récursions multiples

La suite de Fibonacci $(u_n)_n$ est définie par $u_0 = u_1 = 1$ et $u_{n+2} = u_{n+1} + u_n$. Son codage récursif se fait très naturellement :

```
def fibo(n):
    """fibo(n:int)->int
    Renvoie le terme d'indice n de la suite de Fibonacci"""
    if n<=1:                        # cas de base pour n=0 et n=1
        return 1
    else:
        return fibo(n-1)+fibo(n-2)  # propagation avec récursion double
```

Complexité temporelle : Notons $\mathrm{R}(n)$ le nombre de récursions de `fibo`. Il vérifie la relation

$$\mathrm{R}(0) = \mathrm{R}(1) = 1 \quad \text{et} \quad \forall n \geqslant 2 \qquad \mathrm{R}(n) = \mathrm{R}(n-1) + \mathrm{R}(n-2) + 1$$

Par une récurrence immédiate, on montre

$$\forall n \in \mathbb{N} \qquad \mathrm{R}(n) = 2u_n - 1$$

Notons $\varphi = \dfrac{1+\sqrt{5}}{2}$ (le nombre d'or). D'après le théorème sur les suites récurrentes linéaires d'ordre deux, on obtient

$$\forall n \in \mathbb{N} \qquad u_n = \frac{1}{\sqrt{5}} \left[\varphi^{n+1} - \frac{(-1)^{n+1}}{\varphi^{n+1}} \right]$$

La complexité temporelle est du même ordre que le nombre de récursions et on a donc une complexité temporelle exponentielle en $O(\varphi^n)$.

Complexité spatiale : Pour comprendre la complexité spatiale de la fonction, représentons le graphique des appels de `fibo(4)`.

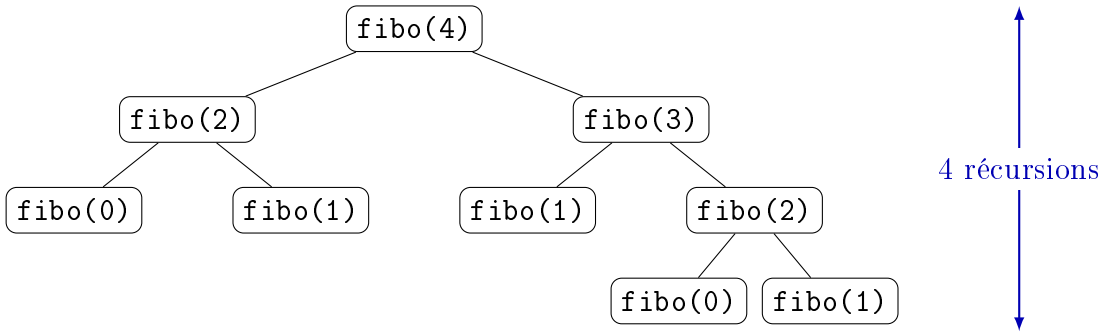


FIGURE 5 – Arbre des appels de `fibo(4)`

Ce graphique est vu comme un *arbre* (avec une arborescence vers le bas, comme dans le système racinaire d'une plante), les étiquettes `fibo(n)` étant appelées des *noeuds* de l'arbre. Cet arbre est parcouru « en profondeur ». Ainsi, la branche la plus longue de l'arbre de `fibo(4)` à `fibo(1)` donne lieu à autant d'empilement que de noeuds dans l'arbres. Puis on dépile en remontant cette branche jusqu'à `fibo(2)`, puis on empile pour parcourir l'autre branche jusqu'à `fibo(0)`, puis on dépile en remontant cette branche jusqu'à `fibo(3)`, etc. ... Ainsi, la complexité spatiale correspond à la longueur de la plus longue branche de l'arbre d'appel. Lors du calcul de `fibo(n)`, la plus longue branche va de `fibo(n)` à `fibo(1)` d'où une complexité spatiale en $\mathrm{O}(n)$.

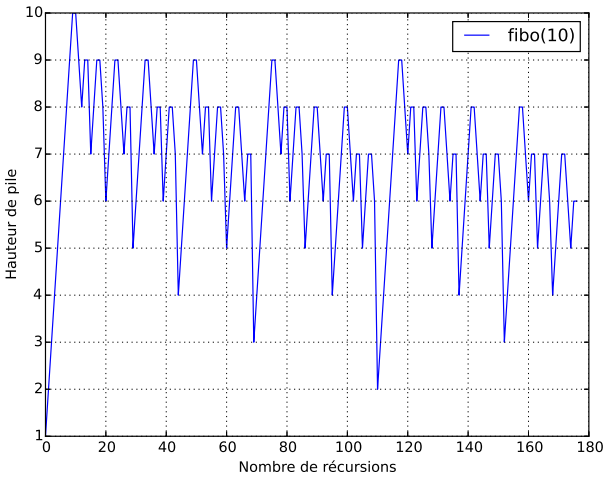


FIGURE 6 – Évolution de la hauteur de pile au cours des récursions de `fibo(10)`

Remarque : Cet exemple de codage récursif des suites de Fibonacci montre qu'il existe des fonctions récursives dont la complexité temporelle diffère de la complexité spatiale.

Théorème 3. *Pour un algorithme récursif d'argument n effectuant p récursions à chaque propagation pour un argument décroissant linéairement et dont les autres opérations sont de complexité temporelle et spatiale en $O(1)$, la complexité spatiale est en $O(n)$ et la complexité temporelle en $O(\rho^n)$ avec $\rho > 1$.*

[Admis]

Remarque : Le cas difficile où les autres opérations ne sont pas de complexité temporelle et spatiale en O(1) est abordé dans le célèbre algorithme de FFT étudié en dernière partie.

• **Comparatif de différentes versions**

```
def fibo1(n):
    u=[1,1]
    for k in range(2,n+1):
        u.append(u[k-1]+u[k-2])
    return u[n]
```

```
def fibo2(n):
    u,v=1,1
    for k in range(2,n+1):
        u,v=v,u+v
    return v
```

Les deux fonctions ne font pas exactement la même chose puisque `fibo1` peut potentiellement renvoyer toutes les valeurs de $(u_0, \dots, u_n)$ alors que `fibo2` ne conserve que u_n. Ces approches s'appuient sur le principe de *programmation dynamique* avec une approche *ascendante* contrairement à la fonction récursive `fibo` qui suit une approche *descendante*.

En posant $\mathrm{X}_n = \begin{pmatrix} u_n \\ u_{n+1} \end{pmatrix}$ pour n entier, on a la relation

$$\mathrm{X}_0 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad \text{et} \quad \forall n \in \mathbb{N} \qquad \mathrm{X}_{n+1} = \begin{pmatrix} u_{n+1} \\ u_{n+2} \end{pmatrix} = \underbrace{\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}}_{=\mathrm{A}} \mathrm{X}_n$$

D'où $$\forall n \in \mathbb{N} \qquad \mathrm{X}_n = \mathrm{A}^n \mathrm{X}_0$$

```
def fibo3(n):
    """exponentiation récursive"""
    A=np.mat([[0,1],[1,1]])
    return expo_rec(A,n)[1,1]
```

```
def fibo4(n):
    """exponentiation itérative"""
    A=np.mat([[0,1],[1,1]])
    return expo_iter(A,n)[1,1]
```

	`fibo`	`fibo1`	`fibo2`	`fibo3`	`fibo4`
Complexité temporelle	$\mathrm{O}(\varphi^n)$	$\mathrm{O}(n)$	$\mathrm{O}(n)$	$\mathrm{O}(\log(n))$	$\mathrm{O}(\log(n))$
Complexité spatiale	$\mathrm{O}(n)$	$\mathrm{O}(n)$	$\mathrm{O}(1)$	$\mathrm{O}(\log(n))$	$\mathrm{O}(1)$

SCHÉMA - Tableau comparatif des complexités


On a fait l'hypothèse simpliste que les opérations arithmétiques sont à coût constant. Pour dresser une comparaison réaliste, il faudrait tenir compte du coût réel de ces opérations : l'addition d'entiers à n bits est en $\mathrm{O}(\log(n))$ et la multiplication d'entiers à n bits en $\mathrm{O}(n\log(n))$ (voir [4] et [5] pour plus de détails).

Exercice : Écrire une fonction récursive pour le calcul des suite de Fibonacci plus performante que la précédente (mais sans exponentiation rapide).

Corrigé : En posant $\mathrm{X}_n = \begin{pmatrix} u_n \\ u_{n+1} \end{pmatrix}$ pour n entier, on a la relation

$$\mathrm{X}_0 = \begin{pmatrix}1\\1\end{pmatrix} \quad \text{et} \quad \forall n \in \mathbb{N} \qquad \mathrm{X}_{n+1} = \begin{pmatrix}u_{n+1}\\u_{n+2}\end{pmatrix} = \begin{pmatrix}u_{n+1}\\u_{n+1}+u_n\end{pmatrix}$$

On en déduit le codage récursif suivant :

```
def Fibo(n):
    if n==0:
        return [1,1]
    else:
        X=Fibo(n-1)
        return [X[1],X[0]+X[1]]
```

Les complexités temporelles et spatiales sont en $\mathrm{O}(n)$. Ça reste moins bien que les complexités obtenues en itératif mais c'est toute de même beaucoup mieux que la version récursive naïve.

Expérimentation :

```
>>> Fibo(30)[0]
1346269
>>> fibo(30)
1346269
```

III Fast Fourier Transform [FFT]

1 Présentation

Soit N un entier non nul et $x = (x_0, \dots, x_{\mathrm{N}-1}) \in \mathbb{C}^{\mathrm{N}}$ les mesures d'un signal échantillonné. La *transformée de Fourier discrète* (TFD) est le N-uplet $y = (y_0, \dots, y_{\mathrm{N}-1})$ défini par

$$\forall k \in [\![\, 0\,;\, \mathrm{N}-1 \,]\!] \qquad y_k = \sum_{n=0}^{\mathrm{N}-1} x_n \mathrm{e}^{-2\mathrm{i}k\pi\frac{n}{\mathrm{N}}}$$

Elle permet d'obtenir une représentation fréquentielle du signal. On peut démontrer mathématiquement que le signal d'origine peut être reconstruit à partir de sa transformée de Fourier discrète avec la transformation inverse

$$\forall n \in [\![\, 0\,;\, \mathrm{N}-1 \,]\!] \qquad x_n = \frac{1}{\mathrm{N}} \sum_{k=0}^{\mathrm{N}-1} y_k \mathrm{e}^{2\mathrm{i}k\pi\frac{n}{\mathrm{N}}}$$

Comme les formules de TFD et TFD inverse sont très proches, on peut les coder en référant à une même fonction `TFD_iter(x,s)` où `x` est un N-uplet de $\mathbb{C}^{\mathrm{N}}$ et `s` est à valeurs dans $\{-1, 1\}$, avec `s` $= -1$ pour réaliser la TFD et `s` $= 1$ pour son inverse sans le facteur $1/\mathrm{N}$ (celui-ci sera intégré lors du renvoi de résultat de `TFDI`).

```
def TFD(x,s=-1):
    """TFD(x:int)->list
    Renvoie la transformée de Fourier discrète de x"""
    N=len(x)
    y=[]
    w=np.exp(s*2*1j*np.pi/N)
    wk=1
```

```
    for k in range(N):
        yk=0
        wn=1
        for  n in range(N):
            yk+=x[n]*wn
            wn*=wk
        y.append(yk)
        wk*=w
    return y

def TFDI(x):
    """TFDI(x:int)->list
    Renvoie la transformée de Fourier discrète inverse de x"""
    return 1/len(x)*np.array(TFD(x,1))
```

On remarquera que le calcul coûteux de l'exponentielle complexe est réalisé une seule fois et qu'on procède ensuite à de simples multiplications pour obtenir le facteur souhaité dans la formule de TFD ou TFD inverse.

La complexité temporelle de `TFD_iter` est en $\mathrm{O}(\mathrm{N}^2)$ (double boucle avec $\sum_{k=0}^{\mathrm{N}-1}\sum_{n=0}^{\mathrm{N}-1}\mathrm{O}(1)=\mathrm{O}(\mathrm{N}^2)$) et sa complexité spatiale en $\mathrm{O}(\mathrm{N})$ avec la variable `y` dont la taille augmente de 1 à chaque passage dans la boucle en `k`.

La transformée de Fourier discrète permet d'obtenir une représentation fréquentielle du signal, un peu à l'image de notre propre perception auditive : nous percevons les sons et donc les fréquences davantage que l'aspect temporel d'un signal.

Réalisons une expérimentation avec le signal

$$\forall t\in\mathbb{R}\qquad x(t)=\sin(t)+2\cos(3t)+\sin(6t)-\sin(10t)$$

Celui-ci est composé de 4 fréquences distinctes. Le code qui suit permet la visualisation du signal d'origine, du signal reconstruit par application successive de la transformée de Fourier discrète et de son inverse et de l'amplitude de la transformée de Fourier discrète.

Expérimentation :

```
t=np.linspace(0,10,2**10)
w=len(t)/10
x=np.sin(t)+2*np.cos(3*t)+np.sin(6*t)-np.sin(10*t)
fx=TFD(x);fix=TFDI(fx)
plt.figure()
plt.subplot(311);plt.plot(t,x);
plt.title('Signal original')
plt.subplot(312);plt.plot(t,np.real(fix),'r');
plt.title('Signal reconstruit')
tf=np.arange(0,len(t))*w/len(t)
plt.subplot(313);plt.xlim(0,2);plt.plot(tf,np.abs(fx))
plt.title('Amplitude de la TFD');plt.show()
```

On obtient la figure suivante :

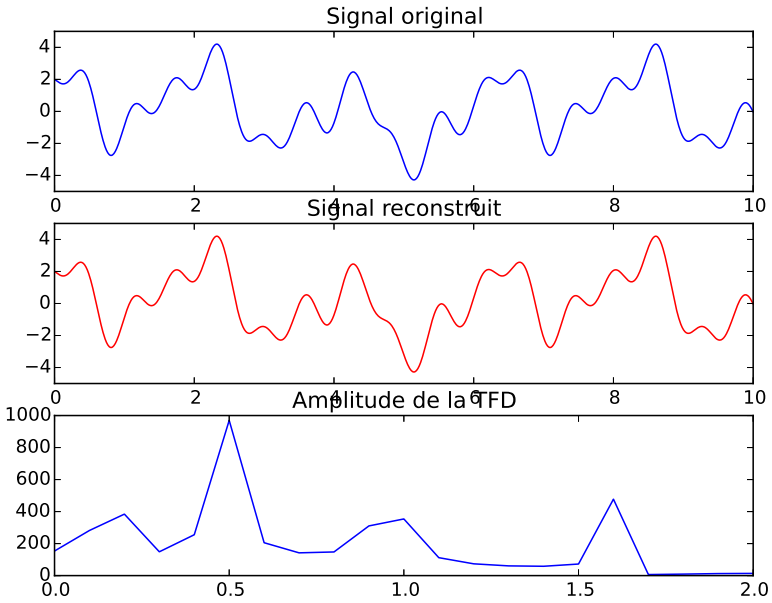


FIGURE 7 – Amplitude de la transformée de Fourier discrète

On observe nettement la présence de 4 pics dans l'amplitude de la TFD, ceux-ci correspondant au nombre de fréquences distinctes dans le signal d'origine à savoir pour 0.16, 0.47, 0.95 et 1.59 hertz.

2 Paradigme « diviser pour régner »

La récursivité gagne incontestablement ses lettres de noblesse avec le paradigme [2] « diviser pour régner ». Il s'agit d'un paradigme fondamental de l'algorithmique : pour résoudre un problème, on le divise en sous-problème plus simples et indépendants.

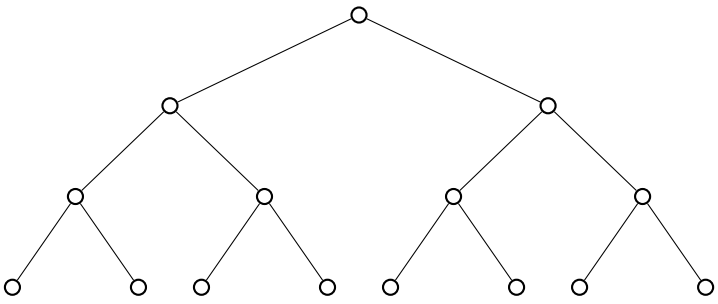


FIGURE 8 – Diviser pour régner

L'algorithme FFT (*Fast Fourier Transform*) de Cooley-Tukey [3] permet, en s'appuyant sur ce concept, un calcul performant de la transformée de Fourier discrète.

2. Un paradigme, concept introduit par le philosophe Thomas Samuel Kuhn en 1962, désigne un ensemble de principes et méthodes partagés par une communauté scientifique.

3. James Cooley, né en 1926, John Tukey, 1915-2000, mathématiciens américains.

On fait l'observation suivante :

$$\forall k \in [\![0\,;\, \mathrm{N}-1]\!] \qquad y_k = \sum_{n=0}^{\mathrm{N}-1} x_n \mathrm{e}^{-2\mathrm{i}k\pi\frac{n}{\mathrm{N}}}$$
$$= \sum_{0\leqslant 2n\leqslant \mathrm{N}-1} x_{2n}\mathrm{e}^{-2\mathrm{i}k\pi\frac{n}{\mathrm{N}/2}} + \mathrm{e}^{-2\mathrm{i}\pi\frac{k}{\mathrm{N}}} \times \sum_{0\leqslant 2n+1\leqslant \mathrm{N}-1} x_{2n+1}\mathrm{e}^{-2\mathrm{i}k\pi\frac{n}{\mathrm{N}/2}}$$

et de même pour la transformation inverse. On constate que le calcul de y_k peut se décomposer en deux sous-calculs de même type avec une extraction des indices selon leur parité ce qui ouvre la voie à un calcul récursif. Cette approche justifie le choix d'*une puissance de 2 pour l'entier* N.

En exploitant également la relation :

$$\forall (k,n) \in [\![0\,;\, \mathrm{N}-1]\!]^2 \qquad \mathrm{e}^{-2\mathrm{i}(k+\mathrm{N}/2)\pi\frac{n}{\mathrm{N}/2}} = \mathrm{e}^{-2\mathrm{i}k\pi\frac{n}{\mathrm{N}/2}} \quad \text{et} \quad \mathrm{e}^{-2\mathrm{i}\pi\frac{k+\mathrm{N}/2}{\mathrm{N}}} = -\mathrm{e}^{-2\mathrm{i}\pi\frac{k}{\mathrm{N}}}$$

il s'ensuit pour $k \in [\![0\,;\, \mathrm{N}/2-1]\!]$

$$y_k = \sum_{0\leqslant 2n\leqslant \mathrm{N}-1} x_{2n}\mathrm{e}^{-2\mathrm{i}k\pi\frac{n}{\mathrm{N}/2}} + \mathrm{e}^{-2\mathrm{i}\pi\frac{k}{\mathrm{N}}} \times \sum_{0\leqslant 2n+1\leqslant \mathrm{N}-1} x_{2n+1}\mathrm{e}^{-2\mathrm{i}k\pi\frac{n}{\mathrm{N}/2}}$$

et
$$y_{k+\mathrm{N}/2} = \sum_{0\leqslant 2n\leqslant \mathrm{N}-1} x_{2n}\mathrm{e}^{-2\mathrm{i}k\pi\frac{n}{\mathrm{N}/2}} - \mathrm{e}^{-2\mathrm{i}\pi\frac{k}{\mathrm{N}}} \times \sum_{0\leqslant 2n+1\leqslant \mathrm{N}-1} x_{2n+1}\mathrm{e}^{-2\mathrm{i}k\pi\frac{n}{\mathrm{N}/2}}$$

On en déduit l'implémentation suivante :

```
def FFT(x,s=-1):
    """FFT(x:int)->list
    Renvoie la transformée de Fourier discrète de x calculée par FFT"""
    N=len(x)
    if N==1:
        return x
    even=FFT(x[0::2],s)
    odd=FFT(x[1::2],s)
    w=np.exp(s*2*1j*np.pi/N)
    res=[0]*N
    wk=1
    for k in range(N//2):
        tk=wk*odd[k]
        res[k]=even[k]+tk
        res[k+N//2]=even[k]-tk
        wk*=w
    return res

def FFTI(x):
    """FFTI(x:int)->list
    Renvoie la transformée de Fourier discrète inverse de x calculée par FFT"""
    return 1/len(x)*np.array(FFT(x,1))
```

Il suffit alors de générer la transformée et son inverse avec l'instruction `fx=FFT(x);fix=FFTI(fx)`.

3 Complexités temporelle et spatiale

Théorème 4. *La complexité temporelle de l'algorithme FFT est quasi-logarithmique, c'est-à-dire*

$$\mathrm{T}(\mathrm{N}) = O(\mathrm{N}\log(\mathrm{N}))$$

Démonstration. Un appel de la fonction `FFT` avec un argument de taille N génère deux appels récursifs avec des sous-listes de taille N/2, des opérations à coût constant et une boucle avec N/2 répétitions d'opérations à coût constant. Ainsi, la complexité temporelle vérifie la relation

$$\mathrm{T(N)} = 2\mathrm{T(N/2)} + \mathrm{O(N)}$$

On a choisi pour N une puissance de 2. Notant $\mathrm{N} = 2^p$ avec p entier, on a

$$\mathrm{T}(2^p) = 2\mathrm{T}(2^{p-1}) + \mathrm{O}(2^p)$$

d'où

$$\frac{\mathrm{T}(2^p)}{2^p} = \frac{\mathrm{T}(2^{p-1})}{2^{p-1}} + \mathrm{O}(1)$$

puis

$$\frac{\mathrm{T}(2^p)}{2^p} = \mathrm{T}(1) + \sum_{k=1}^{p} \left[\frac{\mathrm{T}(2^k)}{2^k} - \frac{\mathrm{T}(2^{k-1})}{2^{k-1}} \right] = \mathrm{O}(p)$$

Par suite, on trouve $\mathrm{T}(2^p) = \mathrm{O}(p2^p)$, c'est-à-dire $\mathrm{T(N)} = \mathrm{O(N\log(N))}$. □

Théorème 5. *La complexité spatiale de l'algorithme FFT est linéaire, c'est-à-dire*

$$\mathrm{S(N)} = O(\mathrm{N})$$

Démonstration. L'arbre des appels récursifs est parcouru en profondeur. La fonction `FFT` crée une variable `res` après les appels récursifs ce qui signifie que les appels descendants ont été dépilés et que l'espace alloué aux créations de la variable `res` lors de ces appels a été libéré. La complexité spatiale correspond donc à la plus grande taille possible de la variable `res` ajoutée à la longueur d'une branche d'appels. Le long d'une branche d'appels, on passe d'un argument de taille $\mathrm{N} = 2^p$ à $\mathrm{N}/2 = 2^{p-1}$. La longueur branche sera donc de taille $p = \log_2(\mathrm{N})$ et par conséquent, la complexité spatiale est en

$$\mathrm{O(N)} + \mathrm{O(\log(N))} = \mathrm{O(N)}$$

□

Exercice : On considère la variante suivante :

```
def FFT(x,s=-1):  # Transformée de Fourier discrète rapide du signal x
    ...
    res=[0]*N
    even=FFT(x[0::2],s)
    odd=FFT(x[1::2],s)
    w=np.exp(s*2*1j*np.pi/N)
    ...
```

La classe de complexité est-elle modifiée ?

Corrigé : Avec cette variante, la variable `res` est créée avant les appels récursifs ce qui signifie qu'au cours des appels successifs, il faut conserver les différentes variables créées. Ainsi, lors du parcours d'une branche de l'arbre d'appels, l'espace mémoire occupé est

$$\mathrm{N} + \frac{\mathrm{N}}{2} + \frac{\mathrm{N}}{2^2} + \ldots + \frac{\mathrm{N}}{2^{p-1}} = \sum_{k=0}^{p-1} \frac{\mathrm{N}}{2^k} = 2\mathrm{N}\left(1 - \frac{1}{2^p}\right) = \mathrm{O(N)}$$

La complexité spatiale est un peu dégradée mais du même ordre que la version initiale.

Variante : En fait, la complexité spatiale vérifie la relation

$$\mathrm{S(N)} = \mathrm{S(N/2)} + \mathrm{O(N)}$$

Avec $\mathrm{N} = 2^p$ où p entier, il vient par sommation des relations de comparaison

$$\begin{aligned}\mathrm{S}(2^p) &= \mathrm{S}(1) + \sum_{k=1}^{p} \left[\mathrm{S}(2^k) - \mathrm{S}(2^{k-1})\right] \\ &= \mathrm{S}(1) + \sum_{k=1}^{p} \mathrm{O}(2^k) = \mathrm{S}(1) + \mathrm{O}\left(\sum_{k=1}^{p} 2^k\right) = \mathrm{O}(2^p) = \mathrm{O(N)}\end{aligned}$$

et on retrouve le résultat précédent.

Références

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms*, Third Edition MIT Press and McGraw-Hill, 2009

[2] Donald Knuth, *The Art of Computer Programming, Volume 1 : Fundamental Algorithms*, Third Edition, Addison-Wesley, 1997

[3] Robert Sedgewick, Philippe Flajolet, *An Introduction to the Analysis of Algorithms*, Second Edition, Addison-Wesley, 2013

[4] Richard Beigel, William I. Gasarch, Ming Li, Louxin Zhang, *Addition in* $\log_2(n) + O(1)$ *Steps on Average : A Simple Analysis*, Theoretical Computer Science, Volume 191, Issues 1-2, 1998

[5] David Harvey, Joris van der Hoeven, *Integer multiplication in time* $O(n \log(n))$, Annals of Mathematics, Volume 193, Issue 2, 2021