TP n°1 - Suite - Récursivité

2. Le PGCD

On se propose dans cette partie, d'utiliser l'algorithme d'Euclide pour implémenter le calcul du pgcd de deux entiers grâce à une fonction récursive.

Démontrer les propriétés arithmétiques suivantes :

Proposition 1.

a) Pour tous entiers a, b,
$$\text{pgcd}(a, b) = \text{pgcd}(b, a)$$

b) Pour tout entier a,
$$\text{pgcd}(a, 0) = a$$

c) Pour tous entiers a, b :
$$\text{pgcd}(a, b) = \text{pgcd}(b, a - b).$$

d) Pour tous entiers a, b, si r est le reste de la division euclidienne de a par b, alors :
$$\text{pgcd}(a, b) = \text{pgcd}(b, r).$$

Exercice 1.

Grâce aux propriétés b) et d), écrire une fonction récursive `pgcd(a,b)` qui retourne le PGCD des entiers `a` et `b`.

3. Coefficients binomiaux

On va calculer numériquement les coefficients binomiaux grâce à la formule de Pascal. On note, pour $n, p \in \mathbb{N}$,
$$\binom{n}{p} = \begin{cases} \frac{n!}{p!(n-p)!} & \text{si } n \geq p, \\ 0 & \text{sinon.} \end{cases}$$

a. Méthode directe

Exercice 2.

En utilisant la fonction `facto`, écrire une fonction `binomFacto(n,p)` qui permette de calculer le coefficient binomial $\binom{n}{p}$.

b. Méthode récursive

Montrer la proposition suivante :

Proposition 2.

Soit $n, p \in \mathbb{N}$. On a :

1. $\binom{n}{n-p} = \binom{n}{p}$
2. $\binom{n}{0} = 1 = \binom{n}{n}$
3. $\binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1}$ pour $n, p \geq 1$.

Et au passage, montrer par récurrence sur $\mathbb{N}$, que pour tout $a, b \in \mathbb{C}$,

$$(a+b)^n = \sum_{i=1}^{n} \binom{n}{p} a^i b^{n-i}.$$

Puis en déduire $\sum_{i=1}^{n} \binom{n}{p}$ et $\sum_{i=1}^{n} (-1)^n \binom{n}{p}$.

Exercice 3.

En utilisant les propriétés 2. et 3., écrire une fonction `binom(n,p)` récursive double qui permette de calculer le coefficient binomial $\binom{n}{p}$.

4. Recherche Dichotomique

On veut implémenter une fonction recherche d'un élément dans un tableau trié par ordre croissant. Voici la première approche naïve :

a. Approche naïve

Exercice 4.

Écrire une fonction `rechercheTab(tab,nb)` où `tab` est une liste de nombres entiers triés par ordre croissant et `nb` un nombre entier à chercher dans la liste, qui retourne l'indice où se trouve `nb` dans la liste et -1 s'il ne s'y trouve pas. On utilisera un parcours de la liste valeur par valeur grâce à une boucle **`while`**.

b. Approche "dichotomique"

Le principe de recherche dichotomique est la suivante : puisque la liste `tab` est triée par ordre croissante, on peut comparer l'entier `nb` et l'entier `tab[moitie]` où `moitie` est l'indice à la moitié de la longueur de la table (grosso modo : la partie entière de la moitié en fait !).

— Si `nb` est plus grand que `tab[moitie]`, alors `nb` se trouve dans la liste `tab[moitie:fin]` où `fin` est le dernier indice de `tab`.

— Si `nb` est plus petit que `tab[moitie]`, alors `nb` se trouve dans la liste `tab[deb:moitie]` où `deb` est le premier indice de `tab` (0 ici).

Exercice 5.

Grâce à la description ci-dessus, écrire une fonction récursive `rechercheTab(tab,deb,fin,nb)` où `tab` est une liste de nombres entiers triés par ordre croissant et où :

- `nb` un nombre entier à chercher dans la liste ;
- `deb` est l'indice de début de recherche dans `tab` ;
- `fin` l'indice de fin de recherche dans `tab` ;

et qui retourne l'indice de `nb` dans la liste s'il s'y trouve et -1 s'il ne s'y trouve pas.

Tester ce programme avec la liste de votre choix et le nombre de votre choix.
Exemple de liste : `L=[1,2,4,4,6,8,9,31,33,45,65,66,107,888]`

5. Rappels sur la complexité (en temps)

On rappelle ici quelques notion sur la complexité et sur le calcul de la complexité d'un algorithme. L'intérêt de la complexité est de pouvoir évaluer l'efficacité d'un algorithme indépendamment de la machine sur laquelle il est exécuté. En effet, la première idée qui pourrait venir pour savoir si un algorithme est efficace, serait de mesure le temps d'exécution de celui-ci : le problème est que ce temps dépend non seulement de la machine mais aussi des processus déjà en cours sur la machine : cette mesure de temps n'est donc pas fiable pour comparer universellement deux algorithmes.

Il nous vient alors l'idée de calculer le nombre de d'opérations élémentaires effectuées dans l'algorithme pendant son exécution.

a. Le coût d'un algorithme

Définition 1.

On appelle **coût** d'un algorithme le nombre d'opérations élémentaires effectuées par l'algorithme.

Mais comment définit-on une opération élémentaire ?

Définition 2.

On appelle **opération élémentaire** une des actions suivantes (la liste n'est pas exhaustive) :

- Faire une opération mathématique : addition, soustraction, multiplication, division, multiplication matricielle, puissance... ;
- Lire le contenu d'une variable ;
- Affecter une valeur à une variable ;
- Effectuer un test : `==`, `>=`, `!=`,... ;
- Faire une opération booléenne : `and`, `not`,...

Voici un exemple :

```
1 def sommeCarre(n):
2     """ Somme des n premiers carres non nuls """
3     S=0
```

```
4     while n>0:
5         S=S+n**2
6         n=n-1
7     return S
```

Question 1.

Quel est le coût de cette algorithme ?

Si on compte toutes les opérations élémentaires citées plus haut, on obtient :
— $n+n+n$ opérations mathématiques ;
— $n+n+n+n+1$ lectures ;
— $1+n+n$ affectations ;
— n comparaisons
Donc le coût total est de $10n+2$ opérations élémentaires.

On remarque alors deux choses qui semblent peu cohérentes dans ces considérations :
- Parmi ces opérations, certaines prennent sûrement plus de temps que d'autres !
- Pour une même opération, le temps d'exécution est proportionnel à la taille des données !

Comment avoir un calcul de coût pertinent dans ces conditions ? Et bien on va faire les hypothèses suivantes afin d'une part, de rendre le calcul de coût plus facile, et d'autre part, de le rendre représentatif de l'efficacité de l'algorithme :

Hypothèse de calcul du coût

1. On fera toujours l'approximation que toutes les opérations élémentaires prennent le même temps ;
2. Dans le calcul du coût, on ne comptera que les opérations élémentaires les plus significatives de l'algorithme.

La deuxième hypothèse est peut sembler subjective : et bien elle l'est ! Mais dans la plupart des cas, il y aura bien une opération plus important que les autres, et qui est la plus pertinente à sélectionner en fonction du contexte de l'algorithme. Par exemple, pour l'algorithme précédent, ce sont les opérations mathématiques (on calcule une somme de carrés) qui constitue les opérations importantes pour cet algorithme. et ainsi, notre calcul de coût devient $3n$ pour ces opérations.

On voit bien dans notre exemple que le coût d'un algorithme dépend fortement de l'argument n. Ainsi, le coût dépend de la taille des données ; mais qu'entend-on exactement par "taille des données" ?

Définition 3.

La taille des données d'un algorithme est (en général) un entier n qui mesure la données à traiter. Comme un algorithme est ici écrit comme une fonction, les données à traiter sont les paramètres d'appels de cette fonction.

Les cas les plus fréquents sont :
- donnée = entier $n \Rightarrow$ taille $= n$
- donnée = liste $\Rightarrow$ taille = longueur de la liste
- donnée = plusieurs entiers $\Rightarrow$ taille = le plus grand de ces entiers (par exemple).

Exemple. Calcul de coût

Considérons l'algorithme suivant qui tri dans l'ordre croissant une liste de nombres donnée en argument :

```
def tri(L):
    n = len(L)
    for i in range(n - 1):
        for j in range(i, n):
            if L[i] > L[j]:
                x = L[i]
                L[i] = L[j]
                L[j] = x
    return L
```

Ici, les opérations qui semblent prédominantes sont les tests et les affections des variables. Calculons le nombres de ces opérations dans l'algorithme :

— Les tests : il y en a

$$\sum_{i=0}^{n-2} n-1-i = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2}$$

— Les affectations : il y en a 1 au départ et 1 à la fin et en fonction de la véracité de chaque test il y en a entre 0 et

$$3\sum_{i=0}^{n-2} n-1-i = 3\frac{n(n-1)}{2}$$

On remarque sur cet exemple simple que le coût dépend non seulement de la taille, mais aussi de la valeur des données !

b. Coût d'un algorithme itératif - boucle `for`

Grâce à l'exemple précédent, on observe que le coût d'un bloc d'une boucle `for` revient s'obtient de la manière suivante :

Calcul du coût d'une boucle `for`

```
for i in range(N):
    instructions(i) #c_f(i) opérations
```

- Pour chaque `i` de la boucle `for`, on détermine le nombre $c_f(i)$ d'opérations élémentaires effectuées par `instructions(i)` dans le bloc de la boucle `for` ;
- Puis on somme ces nombres $c_f(i)$ d'opérations pour i allant de l'indice de début de la boucle `for` (ici 0) à l'indice de fin de la boucle (ici $N-1$).
  Dans le cas précédent, cela donne :

$$\sum_{i=0}^{N-1} c_f(i).$$

Exercice 6.

Calculer le coût des algorithmes suivants en termes d'opérations arithmétiques :

1.
```
def f1(n):
    a=1
    for i in range(n):
        a=a+i*3
    return a
```

2.
```
def f2(n):
    a=1
    for i in range(n):
        a=1+2*a
    for j in range(n):
        a=a//2
    return a
```

3.
```
def f3(n):
    x=1
    for i in range(n):
        for j in range(n):
            x=x+5
    return x
```

Exercice 7.

Calculer le coût, en terme d'opérations mathématiques i.e. additions, produits et racines de l'algorithme suivant en fonction de la donnée n :

```
def mystere(a,b,n):
    for i in range(n):
        a,b=0.5*(a+b),sqrt(a*b)
    return b
```

Que calcule et à quoi correspond cet algorithme ???

c. Coût d'un algorithme itératif - boucle `while`

Dans le cas d'une boucle **`while`**, le principe est le même qu'avec une boucle **`for`** mais une difficulté apparaît : alors qu'on connaît le nombre de tours de boucle d'un **`for`**, le nombre de tour d'un **`while`** n'est pas connu a priori !
Il faut donc le déterminer afin de pouvoir effectuer le même calcul que pour une boucle **`for`**.

Calcul du coût d'une boucle `while`

```
u=valeur_initiale
while condition(u):
    instructions(u) #u n'est pas modifié ici
    u=modif(u)
```

- Pour chaque `i` de la boucle **`while`**, on détermine le nombre $c_w(u)$ d'opérations élémentaires effectuées par `instructions(u)` et `u=modif(u)` dans le bloc de la boucle **`while`** ;
- On modélise les valeurs de u lors de chaque tour de la boucle **`while`** grâce à une suite récurrente $(u_k)_{k\in\mathbb{N}}$:

$$\begin{cases} u_0 &= \texttt{valeur\_initiale} \\ u_k &= \texttt{modif}(u_{k-1}) \text{ pour } k \in \mathbb{N}^*. \end{cases}$$

On utilise alors nos talents mathématiques pour déterminer une expression explicite de cette suite pour $k \in \mathbb{N}$:

$$u_k = f(k)$$

Grâce à cela, on peut déterminer le nombre de tours N de la boucle **`while`**. En effet, ce nombre N est déterminé par :

$$\texttt{condition}(u_{N-1}) \text{ est } \textbf{VRAI !} \text{ et } \texttt{condition}(u_N) \text{ est } \textbf{FAUX !}$$

Dans la pratique, on sera souvent amené à utiliser la fonction réciproque de la fonction f telle que $u_k = f(k)$ pour déterminer N (si tant est quelle soit bijective !).
De plus, il arrivera très souvent que l'on ne puisse pas déterminer exactement le nombre N ! On essaiera alors d'obtenir un encadrement de N en fonction de la taille de la donnée n.

On obtient ainsi toutes les valeurs de u, il s'agit de :

$$u_0, u_1, ..., u_{N-1}.$$

- Finalement, on somme les nombres $c_w(u)$ d'opérations pour chaque valeur de u.
Dans le cas précédent, cela donne :

$$\sum_{k=0}^{N-1} c_w(u_k).$$

Exemple 1.

Voici un exemple concret :

```python
def algo(n):
    a=1
    S=0
    while a<=n:
        S=S+a
        a=a*2
    return S
```

On calcule le coût de `algo(n)` en termes d'opérations arithmétiques :

- Il y a 2 opérations arithmétiques dans le bloc de la boucle `while`.
- Soit (a_k) la suite des valeurs de la variables a. Alors, on a :

$$\begin{cases} a_0 &= 1 \\ a_k &= 2a_{k-1} \text{ pour } k \in \mathbb{N}^*. \end{cases}$$

Ainsi, en calculant de deux manières le produit $\prod_{i=1}^{k} \frac{a_i}{a_{i-1}}$, on obtient :

$$a_k = \frac{a_k}{a_0} = \prod_{i=1}^{k} \frac{a_i}{a_{i-1}} = 2^k.$$

Le nombre de tours N de la boucle `while` vérifie alors

$$a_{N-1} \leq n \text{ i.e. } 2^{N-1} \leq n \text{ et } a_N > n \text{ i.e. } 2^N > n;$$

d'où l'encadrement :

$$\log_2(n) < N \leq \log_2(n) + 1.$$

bu On peut alors calculer le coût $c(n)$ de la boucle `while` et a fortiori de `algo(n)` puisqu'il n'y a pas opérations en dehors de cette boucle :

$$c(n) = \sum_{k=0}^{N-1} 2 = 2N$$

Or, d'après l'encadrement précédent, on a :

$$2\log_2(n) < c(n) \leq 2\log_2(n) + 2.$$

Remarque : Nous verrons dans la suite qu'un tel encadrement nous convient très bien !

Exercice 8.

Déterminer la complexité des algorithmes suivant en termes d'opérations arithmétiques :

1.
```python
def f1(n):
    u=1
    while u<=2*n:
        u+=1
    return u
```

2.
```python
def f2(n):
    u=2**n
    P=1
    while u>1:
        P=P*u
        u=u//2
    return P
```

3.
```python
def f3(n):
    a=1
    S=0
    while a<=n:
          for i in range(a):
              S+=i
          a=a*2
    return S
```

6. La complexité

Malgré la simplicité de nos exemple, on peut voir que le calcul du coût peut très vite devenir impossible pour des algorithmes très grands. Et on remarque que si n est grand, le fait qu'il y ait n^2 opérations où n^2+5n+6 opérations revient quasiment au même puisque les autres termes sont négligeables devant n^2.

Ainsi on a l'idée de définir la complexité d'un algorithme de la façon suivante :

Définition 4. *Complexité*

La **complexité** d'un algorithme est une fonction simple telle que le coût de l'algorithme (pour des opérations élémentaires données) *dans le cas le plus défavorable* est dominée par quand la taille des données tend vers l'infini.

Ainsi, si on note n la taille des données, et $c(n)$ le coût ; la complexité $C(n)$ de l'algorithme vérifie :

$$c(n) = O(C(n)).$$

Remarque 1.

Pour rappeler que la complexité est une approximation asymptotique, on notera souvent $C(n) = O(n^2)$ par exemple au lieu de $C(n) = n^2$.

Quelques fonctions de complexité

$C(n) = O(1)$	complexité constante	extraordinaire !
$C(n) = O(\ln(n))$	complexité logarithmique	excellent
$C(n) = O(n)$	complexité linéaire	super bien
$C(n) = O(n\ln(n))$	complexité quasi-linéaire	très bien
$C(n) = O(n^2)$	complexité quadratique	moyen
$C(n) = O(n^3)$	complexité cubique	mauvais
$C(n) = O(n^p)$	complexité polynomiale ($p > 3$)	très mauvais
$C(n) = O(a^n)$	complexité exponentielle	horrible
$C(n) = O(n!)$	complexité factorielle	à proscrire

7. Calcul de Complexité

a. Cas itératif

Pour un algorithme itératif, il s'agit donc simplement de calculer des sommes pour obtenir le coût dans le pire des cas, et de chercher une fonction simple qui domine ce coût pour obtenir la complexité :

Revenons à notre tri de l'exemple précédent :

```
def tri(L):
    n = len(L)
    for i in range(n - 1):
        for j in range(i, n):
            if L[i] > L[j]:
                x = L[i]
                L[i] = L[j]
                L[j] = x
    return L
```

On a vu que dans le pire des cas, $c(n) = 2 + 4\frac{n(n-1)}{2}$. Donc $C(n) = O(n^2)$! La complexité est quadratique pour cet algorithme.

Exercice 9.

Déterminer la complexité des algorithmes des exercices 6 et 8.

Exercice 10.

Considérons l'algorithme suivant : (remarque, si ce n'est pas déjà fait, n'oubliez pas le `from math import *` pour avoir accès au fonctions mathématiques de Python)

```
def mystere(n):
    for d in range(2, floor(sqrt(n)) + 1):
        if n % d == 0:
            return False
    return True
```

1. Que détermine l'algorithme suivant ?
2. Calculer sa complexité en fonction de n en prenant en compte les congruences (les `%`) comme opérations élémentaires.

b. Cas récursif

Dans le cas d'un algorithme récursif, on calcule le coût par récurrence sur la taille des données n :

Voyons ceci sur un exemple. Considérons la suite récurrente $u_0 = 2$ et $u_{n+1} = \frac{1}{2}(u_n + \frac{1}{u_n})$. Calculons la complexité (en terme d'opérations arithmétiques) de deux implémentations récursives différentes de cette suite :

Programme récursif 1 (naïf)

```
def u1(n):
    if n == 0:
        return 2
    return 0.5 * (u1(n - 1) + 1 / u1(n - 1))
```

Soit $c(n)$ le coût de `u1(n)`. Alors on a :

— $c(0) = 0$

— pour $n \geq 1$, $c(n) = 2c(n-1) + 3$.

Donc, par récurrence, $c(n) = 3(2^n - 1)$. Et donc la complexité est $C(n) = O(2^n)$; c'est exponentiel donc très très très mauvais !

Essayons de faire mieux !

Programme récursif 2 (amélioré !)

```
def u2(n):
    if n == 0:
        return 2
    x=u2(n-1)
    return 0.5*(x+1/x)
```

Soit $c'(n)$ le coût de `u2(n)`. Alors on a :

— $c'(0) = 0$

— pour $n \geq 1$, $c'(n) = c'(n-1) + 3$.

Donc, par récurrence, $c'(n) = 3n$. Et donc la complexité est $C'(n) = O(n)$; complexité linéaire ! c'est super !

Avant de s'exercer sur le calcul de complexité d'algorithmes récursifs, comparons les résultats précédents avec le cas itératif :

Exercice 11.

Donner une version itérative du programme de calcul de la suite précédente et calculer sa complexité.

Maintenant, exerçons nous aux calculs de complexité pour des algorithmes récursifs :

Exercice 12.

Calculer la complexité des différents algorithmes de calcul de factorielles que nous avons étudié au début du TP en terme de multiplications.

Exercice 13.

Considérons l'algorithme suivant :

```
1 def myst(n):
2     if n < 2:
3         return 1
4     return myst(n-1) + myst(n-2)
```

1. Que calcule l'algorithme suivant ?
2. Calculer sa complexité en fonction de n en prenant en compte les additions. Que dire de cette algorithme ?
3. Donner un algorithme permettant d'obtenir le même résultat mais avec un meilleure complexité.