ITC – cours n°7

Algorithmes récursifs

I. Deux exemples introductifs

1.1. La fonction factorielle

Considérons la suite u_n factorielle définie pour tout entier n par $u_n = n!$. On souhaite créer une fonction `factorielle` prenant en entrée un entier n et retournant la valeur de u_n. Une première méthode consiste, de manière itérative, à utiliser la définition et une boucle for :

```
def fact(n):
    """
    Calcule avec une boucle la factorielle d'un nombre n.
    """
    fac = 1
    for i in range(1, n+1):
        fac *= i
    return fac
```

Cet algorithme utilise implicitement la définition par récurrence de la suite u_n :

$$u_{n+1} = u_n \times (n+1) \quad \text{et} \quad u_0 = 1$$

le $n+1^{\text{e}}$ terme est défini à partir du n^{e}. Ainsi, si on veut calculer u_n, on doit calculer u_{n-1}, pour calculer u_{n-1} on doit calculer u_{n-2}...La récursivité utilise cette propriété et consiste à écrire une fonction qui s'appelle elle-même :

```
def fact(n):
    """
    Calcule récursivement la factorielle d'un entier n
    """
    if n == 0:  # condition d'arrêt
        return 1
    return n*fact(n-1)
```

L'utilisation de la récursivité ici donne ici un algorithme plus intuitif : seule la valeur de départ est connue, les autres se calculent à l'aide de la valeur précédente.

1.2. Recherche d'un zéro par dichotomie

On remarque que l'algorithme de recherche d'un zéro est finalement une répétition des mêmes étapes sur un sous-problème similaire : celui de trouver un zéro dans le (bon) intervalle moitié. Ainsi, on peut en écrire une forme récursive :

```
def zéro(f, a, b, eps):
    """
    Cherche un zéro d'une fonction f dans un intervalle [a ; b] avec
    une précision de epsilon.
    On renvoie la moitié d'un intervalle plus petit que epsilon.
    Préconditions:
        f continue sur [a ; b]
        f(a)*f(b) < 0
    """
    if abs(b-a) < eps:  # condition d'arrêt
        return (a+b) / 2
    c = (a+b)/2
    if f(c) == 0:
        return c
    elif f(c)*f(b) < 0:
        return zéro(f, c, b, eps)
    else:
        return zéro(f, a, c, eps)
```

II. Récursivité

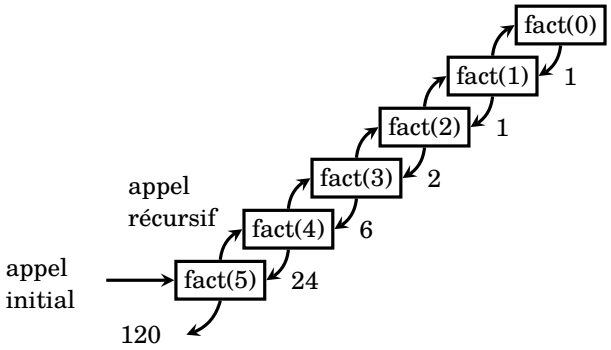
2.1. Analyse et conception d'un programme récursif

Définition – algorithme récursif

Un algorithme récursif est un algorithme qui résout un problème en calculant des solutions d'instances plus petites du même problème. Cela se traduit par une fonction qui s'appelle elle-même avec des arguments « plus petits ».

La plupart des langages de programmation permettent l'utilisation de la récursivité. C'est le cas de Python, qui limite toutefois la récursivité à 1000 appels de la fonction. Ainsi le calcul de 1000! par la méthode précédente ne fonctionne pas. Dans ce cas il faut stocker nous même les premières valeurs de $n!$, dans une liste par exemple.

On peut représenter les appels successifs et les renvois sous forme de graphe d'appels : par exemple, pour calculer 5! :



On remarque qu'il est important d'avoir une condition qui n'appelle pas la fonction, sinon la fonction s'appelle indéfiniment et le programme ne peut s'arrêter. Dans le cas de l'algorithme factorielle la condition est « si $n = 0$ » ; dans le cas de la dichotomie, il s'agit de « si $|b - a| < \varepsilon$ ».

Sans cette condition le programme ne s'arrête pas. Ainsi dans la conception d'une fonction récursive on fera bien attention a inclure une condition d'arrêt, c'est à dire une condition sous laquelle la fonction ne s'appelle pas.

Parmi les situations conduisant naturellement à l'utilisation d'un algorithme récursif, on peut noter :

- les problèmes qui se présentent comme des récurrences (par exemple factorielle) ;
- les algorithmes dichotomiques ;
- les algorithmes diviser-pour-régner (voir section 3.2).

2.2. Avantages et inconvénients d'un programme récursif.

L'avantage de la récursivité réside essentiellement dans la clarté du programme, ainsi que la simplicité de la preuve (puisqu'elle se formule comme une récurrence, outil mathématique bien plus facile à maîtriser que les invariants de boucles). Dans bien des problèmes, l'approche récursive est intuitive (calculs des termes d'une suite définie par récurrence, travail dans les graphes, tris...). Dans ces cas l'écriture du programme en version itérative serait lourde et peu compréhensible alors que l'écriture récursive est claire et simple. De plus, certains problèmes sont quasiment insolvables sans l'utilisation de la récursivité (nous en verrons un exemple à la section 3.3).

En revanche, l'approche itérative est très souvent plus efficace aussi bien d'un point de vue de la mémoire que du temps d'exécution, sauf si le langage de programmation est fondamentalement conçu pour optimiser la récursivité (c'est le cas de OCaml par exemple, mais pas celui de Python). En effet, si dans un programme récursif on maîtrise mal le nombre d'appel à la fonction et ceci peut engendrer des catastrophes temporelles. Illustrons ceci avec la suite de Fibonacci. On rappelle que la suite de Fibonacci u_n est définie par

$$\begin{cases} u_0 = u_1 = 1 \\ u_{n+2} = u_{n+1} + u_n \end{cases}$$

L'algorithme programmé en Python de manière itérative est :

```
def fib(n):
    """
    Calcule la suite de Fibonacci au terme n.
    """
    u, u1 = 1, 1
    for i in range(1, n):
        u, u1 = u + u1, u
    return u
```

et la version récursive est :

```
def fib(n):
    """
    Calcule la suite de Fibonacci au terme n.
    """
    if n < 2:  # condition d'arrêt
        return 1
    return fib(n-1) + fib(n-2)
```

Lorsque l'on compare les temps d'exécutions dans les deux cas, on voit que le programme itératif est incroyablement plus efficace. En observant de plus près le comportement de la fonction on voit que le nombre d'appels explose car on recalcule plusieurs fois le même résultat (par exemple dans un appel à $fib(6)$, $fib(3)$ est calculée trois fois) :

De plus, lorsque l'on exécute une fonction récursive celle ci stocke dans une pile les différents appels à la fonction et occupe ainsi de la mémoire. Si on reprend l'algorithme `factorielle`, dans sa version itérative il ne stocke qu'une seule variable qu'il fait évoluer. Dans sa version récursive il stocke une pile d'appels à la fonction, donc n appels avec

leurs variable de retour.

2.3. Complexité d'un programme récursif

On a vu dans la partie précédente que la complexité d'un programme récursif peut vite exploser, il est donc judicieux de s'intéresser à la complexité temporelle de tels algorithmes. Sans surprise, l'étude de la complexité utilisera la notion mathématique de suites et notamment la définition par récurrence d'une suite. Le principe est de noter c_n le nombre d'opérations élémentaires (affectations, comparaisons, opérations arithmétiques...) nécessaires à l'exécution de l'algorithme, puis la définition récursive de la fonction permettra d'établir un lien entre c_n et c_{n-1}. L'étude de la suite $(c_n)_{n\in\mathbb{N}}$ conduira alors à la complexité cherchée.

Pour illustrer cette méthode, reprenons l'exemple de la fonction factorielle définie dans la première partie : chaque appel correspond à deux opérations élémentaires, une comparaison et une multiplication (sauf dans le cas $n = 0$, qui n'arrive qu'une fois), plus l'appel à la fonction avec la valeur $n - 1$; la récurrence est donc :

$$\begin{cases} c_{n+1} = c_n + 2 \\ c_0 = 1 \end{cases} \quad \Rightarrow \quad c_n = 2n + 1 = \mathcal{O}(n)$$

C'est donc une complexité linéaire, comme pour la version itérative.

On peut également étudier la complexité de l'algorithme de calcul des termes de la suite de Fibonacci : ici, on considère comme opérations élémentaires le test et la somme, et en ajoutant les appels récursifs :

$$\begin{cases} c_{n+2} = c_{n+1} + c_n + 2 \\ c_0 = c_1 = 1 \end{cases}$$

On obtient une suite récurrente d'ordre 2 si on pose $d_n = c_n + 2$:

$$d_{n+2} = c_{n+2} + 2 = c_{n+1} + c_n + 4 = d_{n+1} + d_n$$

et l'équation caractéristique $x^2 = x + 1$ a pour solutions

$$\varphi = \frac{1+\sqrt{5}}{2} \quad \text{et} \quad \varphi' = \frac{1-\sqrt{5}}{2}$$

d'où la forme générale de la solution

$$c_n = a\varphi^n + b\varphi'^n - 2 \quad , \quad (a,b) \in \mathbb{R}^2$$

Puisque $|\varphi| > 1$ et $|\varphi'| < 1$, à la limite $n \to \infty$,

$$c_n = \mathcal{O}(\varphi^n)$$

La complexité est donc exponentielle[1].

[1] La version itérative est linéaire.

III. Quelques exemples supplémentaires

3.1. Algorithme dichotomique : exponentiation rapide

On reprend l'algorithme d'exponentiation rapide vu dans le chapitre sur la dichotomie. Pour rappel, on cherche à calculer x^n et on utilise les propriétés suivantes :

- si $n = 0$, $x^n = 1$;
- sinon, si on connaît $y = x^{\lfloor n/2 \rfloor}$, alors :
  - ▷ si n est pair, $x^n = y^2$;
  - ▷ sinon, $x^n = x \times y^2$.

Puisque y est lui-même une puissance de x, il peut lui-même être calculé par un appel à cet algorithme : cela suggère une version récursive.

```
def expo(x, n):
    """
    Calcule x**n
    """
    if n == 0:
        return 1
    y = expo(x, n//2)
    if n%2 == 0:
        return y**2
    else:
        return x * y**2
```

On constate que cette version est beaucoup plus concise et simple à comprendre que la version itérative. La complexité se calcule en constatant qu'il n'y a que des opérations élémentaires en-dehors de l'appel récursif, donc

$$\begin{cases} c_n = c_{n//2} + a \\ c_0 = 1 \end{cases}$$

L'analyse peut se terminer de deux manières :

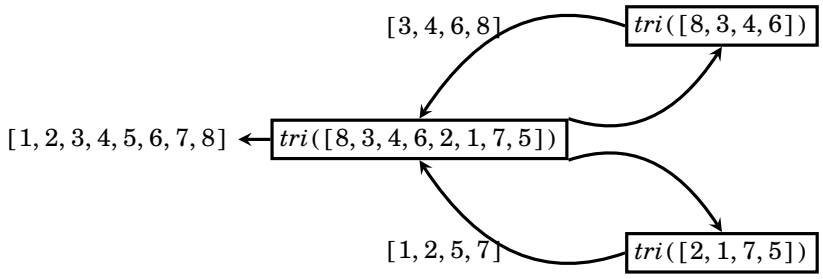
- soit on compte le nombre d'appels récursifs : à chaque appel n est divisé par deux jusqu'à atteindre 0, il y a donc environ $\mathcal{O}(\log_2(n))$ appels de coût constant : la complexité est donc $\mathcal{O}(\log_2(n))$
- soit on peut montrer par récurrence forte que $c_n \leqslant a + a\log_2 n$: si on suppose que cette majoration est vraie pour tout rang jusqu'à $n-1$, alors

$$c_n = c_{n/2} + a \overset{\text{hyp}}{\leqslant} a + a\log_2\frac{n}{2} + a = a + a\log_2 n - a\log_2 2 + a = a + a\log_2 n$$

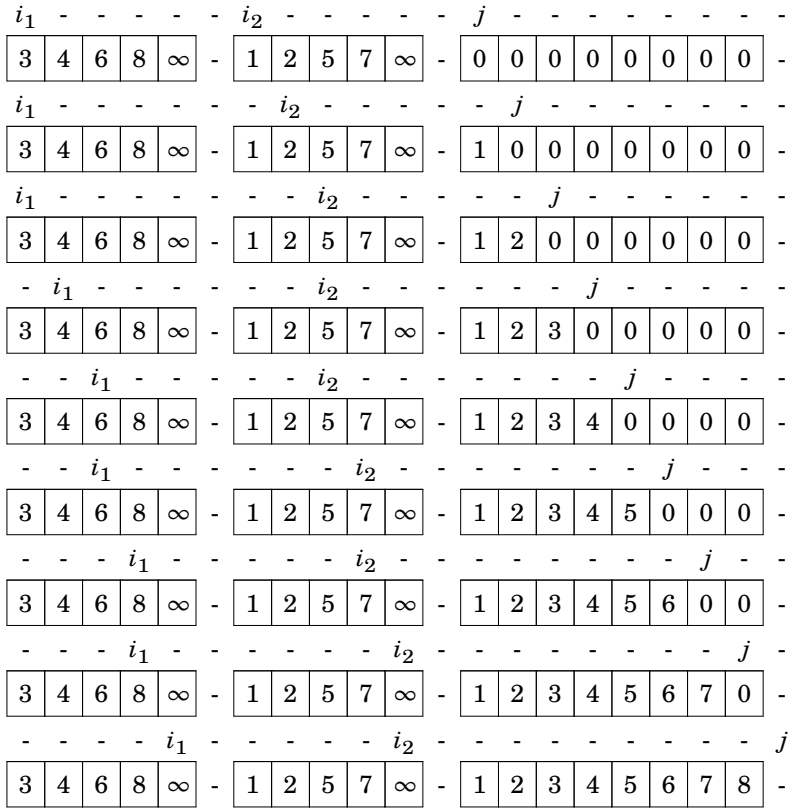
ce qui prouve la récurrence et permet de conclure que la complexité est en $\mathcal{O}(\log_2 n)$.

3.2. Un tri récursif : le tri fusion

Le tri fusion applique le paradigme diviser-pour-régner : l'idée est qu'avec deux tableaux triés, on peut facilement les fusionner pour créer un nouveau tableau lui-même trié :



Le cœur de la procédure est l'opération de fusion des deux sous-tableaux : on la réalise en parcourant le tableau réponse avec un indice j, tout en maintenant un parcours sur les deux sous-tableaux en parallèle à l'aide de deux compteurs i_1 et i_2 ; à chaque étape, on choisit le bon élément à copier en case j du tableau de résultat. Afin de simplifier la gestion des bords et des dernières cases, on ajoutera une case « sentinelle » de valeur infinie (ou un majorant des deux tableaux triés) dans la dernière case.



On propose l'implémentation suivante, dans laquelle on renvoie une copie triée du tableau :

```python
from math import inf

def fusionner(t1, t2):
    """
    Prend en paramètre deux tableaux triés et renvoie un tableau
    contenant les éléments de l'ensemble, trié.
    """
    n1, n2 = len(t1), len(t2)
    i1, i2 = 0, 0       # compteurs pour savoir quel est l'élément courant
    tr = [0] * (n1+n2)  # tableau à renvoyer
    t1.append(inf)      # sentinelle
    t2.append(inf)      # sentinelle
    for j in range(n1+n2):
        if t1[i1] > t2[i2]:
            tr[j], i2 = t2[i2], i2+1
        else:
            tr[j], i1 = t1[i1], i1+1
    return tr

def tri_fusion(tab):
    """
    Renvoie une copie triée de tab
    """
    n = len(tab)
    if n == 1:  # condition d'arrêt
        return tab
    t1 = tri_fusion(tab[:n//2])
    t2 = tri_fusion(tab[n//2:])
    return fusionner(t1, t2)
```

On peut étudier la complexité du tri fusion : La fonction fusion est en $\mathcal{O}(n) = an$ et un appel à tri-fusion sur un tableau de taille n coûte donc

$$c_n = 2c_{n//2} + an$$

On raisonne sur un tableau de départ de taille $n = 2^k$; il s'ensuit que

$$c_{2^k} = 2c^{2^{k-1}} + a2^k \quad \Rightarrow \quad \frac{c_{2^k}}{2^k} = \frac{c_{2^{k-1}}}{2^{k-1}} + a \quad \Rightarrow \quad w_k = w_{k-1} + a$$

où on a posé $w_k = c_{2^k}/2^k = c_n/n$; il s'agit d'une suite arithmétique, donc

$$w_k = w_0 + ka \quad \Rightarrow \quad \frac{c_n}{n} = b + a\log_2 n \quad \Rightarrow \quad \boxed{c_n = \mathcal{O}(n\log_2 n)}$$

Le résultat se maintient par encadrement si $n \neq 2^k$:

$$2^k \leqslant n < 2^{k+1} \quad \Rightarrow \quad k2^k \leqslant c_n < (k+1)\,2^{k+1} \quad \Rightarrow \quad c_n = \mathcal{O}(n\log_2 n)$$

3.3. Combinatoire : énumération des permutations d'une liste

On peut élégamment créer l'ensemble $S(\ell)$ des permutations d'une liste $\ell = \{\ell_i\}$ en considérant que

- l'ensemble des permutations d'une liste vide est le singleton « liste vide » ;
- pour avoir la permutation d'un ensemble de taille n, on peut tour à tour isoler chaque élément, le mettre en position 0, et considérer l'ensemble des permutations de taille $n-1$;

ce qui se traduit mathématiquement par :

$$S(\ell) = \bigcup_{k=0}^{n-1} (\ell_k + S(\ell \setminus \{\ell_k\}))$$

Cette analyse conduit à la fonction récursive suivante :

```python
def permutations(tab):
    """
    Génère l'ensemble des permutations possibles pour un tableau fourni.
    Renvoie cet ensemble dans une liste.
    """
    n = len(tab)
    if n == 0:       # condition d'arrêt
        return [[]]  # l'ensemble des perm. est la perm. vide
    perms = []
    for i in range(n):
        # on met de côté l'élément n°i et prépare le sous-tableau restant
        elt0 = tab[i]
        sous_tab = tab[:i] + tab[i+1:]
        # appel récursif et génération des nouvelles permutations
        sous_perms = permutations(sous_tab)
        for sp in sous_perms:
            perms.append([elt0] + sp)
    return perms
```

IV. Exercices

1 – L'algorithme d'Euclide permettant de calculer le PGCD de deux nombres entiers est donné par le pseudo-code suivant :

tant que $b \neq 0$

$r \leftarrow a \bmod (b)$

$a \leftarrow b$

$b \leftarrow r$

Renvoie a

Écrire pour l'algorithme d'Euclide une version itérative et une version récursive.

Correction :

On peut le coder manière itérative par :

```python
def pgcd(a, b):
    """
    Calcule un PGCD avec l'alogrithme d'Euclide
    """
    x, y = a, b
    while y != 0:
        x, y = y, x%y
    return x
```

alors que sa version récursive est la suivante :

```python
def pgcd(a, b):
    """
    Calcule un PGCD avec l'alogrithme d'Euclide
    """
    if b == 0:  # condition d'arrêt
        return a
    return pgcd(b, a%b)
```

L'algorithme récursif est très concis.

---

2 – Écrire la recherche dichotomique dans un tableau trié sous forme récursive (on renverra `True` si l'élément cherché est dans le tableau).

Correction

```python
def recherche(tab, x):
    """
    Recherche l'élément x dans la liste triée tab.
    Returns:
    True si x \in tab, False sinon
    """
    n = len(tab)
    if n == 1:  # condition d'arrêt
        if tab[0] == x:
```

```
            return True
        return False
    m = n//2
    if tab[m] == x:
        return True
    elif x < tab[m]:  # attention aux intervalles ouverts/fermés
        return recherche(tab[:m], x)  # n//2 est exclu
    else:
        return recherche(tab[m:], x)
```

3 – Écrire une fonction récursive calculant les deux derniers termes de la suite de Fibonacci. Quelle est sa complexité ?

Correction

```
def fib_2(n):
    """
    Calcule les deux derniers termes de la suite de Fibonacci.
    """
    if n == 2:  # condition d'arrêt
        return 1, 1
    u1, u2 = fib_2(n-1)
    return u1+u2, u1
```

Puisque chaque appel ne fait qu'un seul appel récursif pour $n-1$, et que la fonction elle-même est en temps constant,

$$c_n = c_{n-1} + a \quad \Rightarrow \quad c_n = na + c_0 = \mathcal{O}(n)$$

Le fait de travailler sur les deux derniers termes permet de régler le problème précédent.

4 – Programmer récursivement une fonction `rallonge(mot, car, n)` qui prend en paramètre une chaîne de caractères `mot` , un caractère `car` et un nombre n, et qui renvoie la chaîne de caractères entourée de n fois le caractère car à gauche et à droite.

Correction

```
def rallonge(mot, car, n):
    """
    Ajoute n fois le caractère car à gauche et à droite de mot
    """
    if n == 0:  # condition d'arrêt
        return mot
    return car + rallonge(mot, car, n-1) + car
```

5 – Programmer récursivement une fonction `palindrome(mot)` qui prend en paramètre une chaîne de caractères `mot` et vérifie qu'il s'agit d'un palindrome, c'est-à-dire un mot symétrique. Elle renverra `True` si le mot est un palindrome et `False` sinon.

Correction

```
def palindrome(mot):
    """
    Teste si le mot passé en paramètre est un palindrome.
    """
    if len(mot) <= 1:  # condition d'arrêt
        return True
    if mot[0] == mot[-1]:
        return palindrome(mot[1:-1])
    else:
        return False
```

6 – Programmer récursivement une fonction `compteur(mot, car)` qui prend en paramètre une chaîne de caractères `mot` et un caractère `car` , et compte les occurrences du caractère dans le mot.

Correction

```
def compteur(mot, car):
    """
    Compte le nombre d'occurrences du caractère car dans mot
    """
    if len(mot) == 0:  # condition d'arrêt
        return 0
    if mot[0] == car:
        return compteur(mot[1:], car) + 1
    else:
        return compteur(mot[1:], car)
```