

Série 9 - Corrigé

Sauf si spécifié autrement, tous les algorithmes demandés sont à écrire sous forme d'une fonction Python.

Exercice 1 (facultatif)

Soit $n \in \mathbb{N}$ et f, g des fonctions positives de n . Prouver les affirmations suivantes :

- (a) Si $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = +\infty$ alors $f = \mathcal{O}(g)$ et $g \neq \mathcal{O}(f)$.
- (b) Si $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ alors $f = \mathcal{O}(g)$ et $g \neq \mathcal{O}(f)$.

Remarque : la résolution de cet exercice est optionnelle mais il faut connaître le résultat, qui sera utile à l'exercice suivant.

Solution.

- (a) Si $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = +\infty$ alors pour toute constante $M > 0$ il existe un rang $N > 0$ (fonction de M) tel que pour tout $n > N$,

$$\frac{g(n)}{f(n)} > M.$$

Fixons n'importe quelle telle constante M . Par exemple, choisissons $M = 1$, et soit N le rang correspondant tel que pour tout $n > N$,

$$\frac{g(n)}{f(n)} > 1.$$

Alors pour tout $n > N$, on a $f(n) < g(n)$, donc $f(n) \leq 1 \cdot g(n)$ et donc $f(n) = \mathcal{O}(g(n))$.

Pour montrer que $g(n) \neq \mathcal{O}(f(n))$, on raisonne par l'absurde : supposons qu'il existe une constante $C > 0$ et un rang $N > 0$ tels que pour tout $n > N$,

$$g(n) \leq C \cdot f(n).$$

Mais alors, $C \geq \frac{g(n)}{f(n)}$, qui tend vers l'infini : une telle constante ne peut pas exister.

- (b) On peut utiliser le fait que $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ implique que $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = +\infty$ et appliquer le point (a).

Alternativement, si $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$, alors pour toute constante $C > 0$ il existe un rang $N > 0$ (fonction de C) tel que pour tout $n > N$,

$$\frac{f(n)}{g(n)} < C.$$

Fixons n'importe quelle telle constante C . Par exemple, choisissons $C = 1$, et soit N le rang correspondant tel que pour tout $n > N$,

$$\frac{f(n)}{g(n)} < 1.$$

On a donc que pour tout $n > N$, $f(n) \leq 1 \cdot g(n)$ et donc $f(n) = \mathcal{O}(g(n))$.

Pour montrer que $g(n) \neq \mathcal{O}(f(n))$, on raisonne par l'absurde : supposons qu'il existe une constante $C > 0$ et un rang $N > 0$ tels que pour tout $n > N$,

$$g(n) \leq C \cdot f(n).$$

Mais alors, $\frac{1}{C} \leq \frac{g(n)}{f(n)}$, qui tend vers 0 : une telle constante $1/C$ ne peut pas exister.

Exercice 2

Ordonner (sans justification) les fonctions suivantes par ordre de croissance, en groupant ensemble les fonctions qui ont le même ordre de croissance. Vous pouvez vérifier vos réponses en visualisant le graphes de ces fonctions à l'aide d'un outil de plotting comme à la Série 8.

$$n \log(n), (\log(n))^2, n, n^2, 2^n, 1000n^{10}, \sqrt{n}, (\log(n^2)), \sqrt{n} \log(n), \log(n).$$

On rappelle que $\log(n)$ dénote dans ce cours le logarithme de base 2.

Solution.

Voici les fonctions données ordonnées par ordre de croissance, où on a groupé avec des accolades les fonctions qui ont le même ordre de croissance :

$$\{\log(n), (\log(n^2))\}, (\log(n))^2, \sqrt{n}, \sqrt{n} \log(n), n, n \log(n), n^2, 1000n^{10}, 2^n.$$

Quelques explications :

- $\log(n^2) = 2 \log(n)$ par une propriété vue au cours, et donc $\log(n^2) = \Theta(\log(n))$.

- $\log(n) = \mathcal{O}((\log(n))^2)$ (vu au cours)
- $(\log(n))^2 = \mathcal{O}(\sqrt{n})$ (vu au cours)
- La croissance de \sqrt{n} est dominée par la croissance de $\sqrt{n} \log(n)$ puisque

$$\lim_{n \rightarrow \infty} \frac{\sqrt{n} \log(n)}{\sqrt{n}} = \lim_{n \rightarrow \infty} \log(n) = +\infty$$

Par l'exercice 1 on a que $\sqrt{n} = \mathcal{O}(\sqrt{n} \log(n))$.

- La croissance de $\sqrt{n} \log(n)$ est dominée par la croissance de n puisque

$$\lim_{n \rightarrow \infty} \frac{\sqrt{n} \log(n)}{n} = \lim_{n \rightarrow \infty} \frac{\log(n)}{\sqrt{n}} = 0.$$

Par l'exercice 1, cela implique que $\sqrt{n} \log(n) = \mathcal{O}(n)$.

- La croissance de n est dominée par la croissance de $n \log(n)$ puisque

$$\lim_{n \rightarrow \infty} \frac{n \log(n)}{n} = +\infty.$$

Par l'exercice 1, cela implique que $n = \mathcal{O}(n \log(n))$.

- La croissance de $n \log(n)$ est dominée par la croissance de n^2 puisque

$$\lim_{n \rightarrow \infty} \frac{n \log(n)}{n^2} = \lim_{n \rightarrow \infty} \frac{\log(n)}{n} = 0.$$

Par l'exercice 1, cela implique que $n \log(n) = \mathcal{O}(n^2)$.

- $n^2 = \mathcal{O}(1000n^{10})$ (puissances rationnelles de n , vu au cours)
- $1000n^{10} = \Theta(n^{10})$ et $n^{10} = \mathcal{O}(2^n)$ (croissance polynomiale vs croissance exponentielle).

Exercice 3

On vous donne ci-dessous l'algorithme de recherche binaire vu au cours 9, où on a rajouté des instructions `print()` dans la boucle `while`.

```
def recherche_binaire(L, x):
    bas = 0
    haut = len(L)-1

    while haut >= bas:

        milieu = (haut + bas)//2
        print("bas =", bas, ", haut =", haut, ", milieu =", milieu,
              end = " , ")
        print("L[" ,milieu, "] = ", L[milieu], sep = " ", end = " ")

        if L[milieu] == x:
            return milieu

        if L[milieu] > x:
            haut = milieu - 1
            print(">", x)

        if L[milieu] < x:
            bas = milieu + 1
            print("<", x)
```

Qu'affichent les instructions suivantes?

```
L = [10, 12, 14, 16, 18, 20, 22]
```

```
x = 17
```

```
recherche_binaire(L, x)
```

Répétez l'exercice avec $x = 20$, $x = 23$, et d'autres valeurs de x .

Solution.

Exécutez le code. Testez d'autres valeurs de L et de x .

Exercice 4

Vous avez vu en cours un algorithme de recherche binaire **itératif**. Donnez un algorithme de recherche binaire **récurif** `recherche_binaire_rec` qui :

- prend en entrée une liste de nombres L triée, un élément x à rechercher dans la liste, et deux indices `bas` et `haut` (qui correspondent à la tranche de la liste L dans lequel on recherche l'élément x)
- retourne un indice i tel que $\text{bas} \leq i \leq \text{haut}$ et $L[i] = x$ si un tel indice existe, `None` sinon.

Etant donné une liste L et un élément x, le premier appel à votre algorithme sera recherche_binaire_rec(L, 0, len(L)-1, x).

La condition d'arrêt doit être formulée en fonction des valeurs de haut et bas.

Solution.

L'algorithme suivant recherche récursivement un élément x dans une liste triée L :

```
def recherche_binaire_rec(L, bas, haut, x):
    """
    Entree: nombre x, liste L triee, indices bas et haut
    Sortie: i tq bas<=i<=haut et L[i]=x, s'il existe
           None sinon
    """
    milieu = (bas+haut)//2

    if haut < bas:
        return None
    elif L[milieu] == x:
        return milieu
    elif L[milieu] > x:
        return recherche_binaire_rec(L,bas,milieu-1,x)
    else:
        return recherche_binaire_rec(L,milieu+1,haut,x)
```

Pour mieux comprendre cet algorithme, vous pouvez rajouter des instructions print, comme suit :

```
def recherche_binaire_rec(L, bas, haut, x):

    milieu = (bas+haut)//2
    print("Je suis dans recherche_binaire_rec", end = "")
    print("(L,", bas, ",", haut, ",", x, ")", end = " ")
    print("L[" , milieu, "] = " , L[milieu])

    if haut < bas:
        print(haut, "<", bas, "je retourne!")
        return None
    elif L[milieu] == x:
        return milieu
    elif L[milieu] > x:
        return recherche_binaire_rec(L,bas,milieu-1,x)
    else:
        return recherche_binaire_rec(L,milieu+1,haut,x)
```

Qu'affichent les instructions suivantes?

```
L = [10, 12, 14, 16, 18, 20, 22]
x = 17
print("recherche_binaire_rec(L, 0, 6, x) retourne", recherche_binaire_rec(L,0,6,x))
```

Répétez l'exercice pour $x = 20$, $x = 23$, et d'autres valeurs de x . Pour vérifier votre réponse, exécutez le code.

Exercice 5

On se propose d'écrire un algorithme qui prend en entrée une liste de nombres L triée et une valeur x , et retourne l'indice de la **première occurrence** de x dans la liste. Si aucun élément de valeur x n'est dans la liste, l'algorithme doit retourner `None`.

Par exemple, pour la liste $[10, 20, 20, 20, 20, 30, 30]$ et la valeur 20 en entrée, l'algorithme doit retourner l'indice 1 (alors que `recherche_binaire` retournerait l'indice 3).

- (a) Implémentez en Python l'algorithme suivant, qui résoud ce problème :
- Appeler `recherche_binaire(L, x)`
 - Si cet appel retourne un indice i , parcourir L en allant à gauche depuis i jusqu'à trouver un élément différent de x (ou jusqu'à arriver au début de la liste)
 - Retourner l'indice i de la première occurrence de x dans L .

Quel est l'ordre de croissance du temps de parcours de cet algorithme au pire des cas, en fonction de la taille n de la liste L ?

- (b) Donnez un autre algorithme qui résoud le même problème mais dont le temps de parcours au pire des cas est de l'ordre de $\log(n)$.

Solution.

- (a) Le code ci-dessous implémente l'algorithme donné :

```
def recherche_premier_lineaire(L, x):
    i = recherche_binaire(L,x)

    if i != None:
        for j in range(i,0,-1):
            if L[j-1] != x:
                return j
        return 0

L = [0, 1, 2, 2, 2, 3, 4, 4, 4, 5, 5, 5, 5, 6, 7]
print(recherche_premier_lineaire(L, 5))
```

Cet algorithme a un temps de parcours linéaire en n au pire des cas. En effet, considérons une liste L contenant n fois la valeur x . L'appel à `recherche_binaire` retourne un indice i qui est approximativement égal à $n/2$. Pour trouver la première occurrence de x , l'algorithme devra parcourir la moitié de la liste, en un temps $\Theta(n)$.

- (b) Le code ci-dessous adapte l'algorithme de recherche binaire vu en cours pour rechercher la première occurrence d'un élément x dans une liste triée L :

```
def recherche_premier(L, x):
    n = len(L)
    bas = 0
    haut = n-1

    while haut >= bas:
        milieu = (bas + haut)//2
        if L[milieu] == x:
            if milieu == 0 or L[milieu - 1] != x:
                return milieu
            else:
                haut = milieu - 1
        elif L[milieu] > x:
            haut = milieu - 1
        else:
            bas = milieu + 1

    return None

L = [0, 1, 2, 2, 2, 3, 4, 4, 4, 5, 5, 5, 5, 6, 7]
print(recherche_premier(L, 5))
```

La seule différence avec la recherche binaire est que quand on trouve l'élément x à un certain indice de la liste, au lieu de retourner cet indice il faut aller à gauche, sauf si on a trouvé le premier tel indice : c'est le cas si l'indice précédent ($\text{milieu} - 1$) contient un élément différent de x , ou si on est au tout début de la liste (milieu est égal à 0).

Tout comme la recherche binaire, cet algorithme "coupe" la liste en deux à chaque itération et a donc un temps de parcours qui est $\Theta(\log(n))$.

Pour une comparaison empirique des temps de parcours au pire des cas des deux algorithmes, on construit des listes qui contiennent une même valeur répétée n fois, et on recherche la première occurrence de cette valeur (ainsi, l'indice trouvé par recherche binaire, vers le milieu

de la liste, est le plus loin possible de la première occurrence, qui est à l'indice 0). Puis on prend des valeurs de plus en plus grandes de n . La figure 1 illustre les temps de parcours de chacun des deux algorithmes.

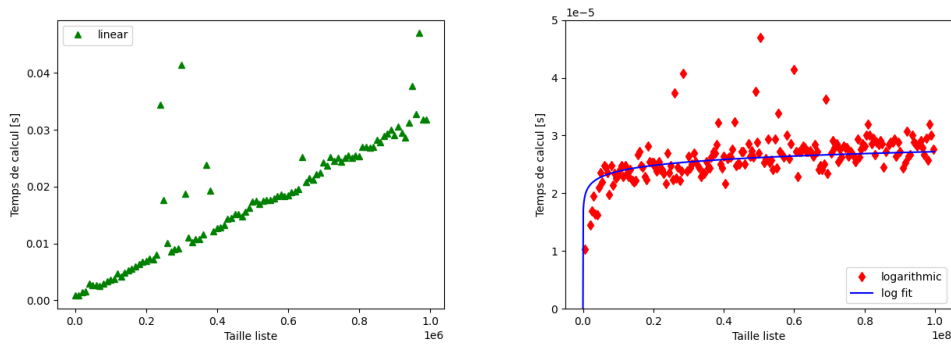


FIGURE 1 – Temps de parcours de l’algorithme linéaire (gauche) et de l’algorithme logarithmique (droite). Sur ce dernier, la fonction $y = A \log x$ a été reportée, où A est une constante de normalisation.

Exercice 6

Implémentez une version récursive (sans boucle) de l’algorithme de recherche linéaire vu en cours. Votre algorithme `recherche_rec` doit :

- prendre en entrée une liste de nombres L , un élément x à rechercher dans la liste, et un indice `debut` qui correspond au début de la sous-liste de L dans laquelle on recherche x
- retourner un indice $i \geq \text{debut}$ tel que $L[i] = x$ si un tel indice existe, `None` sinon.

Etant donné une liste L et un élément x , le premier appel à votre algorithme sera `recherche_rec(L, x, 0)`.

Solution.

L’algorithme suivant recherche récursivement un élément dans une liste :

```
def recherche_rec(L, x, debut):
    if debut == len(L):
        return
    if L[debut] == x:
        return debut
    return recherche_rec(L, x, debut + 1)
```

Par exemple, l'appel
`print(recherche_rec([10, 12, 14, 16, 18], 16, 0))`
retourne 3, et l'appel
`print(recherche_rec([10, 12, 14, 16, 18], 17, 0))`
retourne None.

Exercice 7

Vous avez vu au cours et à la série 7 que le n -ème nombre de Fibonacci f_n peut être calculé récursivement ou itérativement :

```
def fib_rec(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib_rec(n-1) + fib_rec(n-2)

def fib_iter(n):
    if n == 0 or n == 1:
        return n
    fib_old = 0
    fib_new = 1
    for i in range(2, n+1):
        fib_old, fib_new = fib_new, fib_old + fib_new
    return fib_new
```

- (a) Exprimez (avec une brève justification) l'ordre du temps de parcours de `fib_iter` en notation $\Theta(\cdot)$ en fonction de n .

Cet algorithme itératif est bien plus efficace que l'algorithme récursif `fib_rec` (pour vous en convaincre, appelez `fib_iter(n)` et `fib_rec(n)` pour des valeurs de n de plus en plus grandes).

Mais pour arriver au calcul de f_n , `fib_iter` calcule toutes les valeurs de f_i pour $i \leq n$. Donc si on appelle la fonction `fib_iter` plusieurs fois avec diverses valeurs de n , on recalcule plusieurs fois les mêmes valeurs. Par exemple, si on appelle `fib_iter(10)` et qu'on appelle plus tard `fib_iter(15)`, la valeur de f_{10} sera recalculée pendant le deuxième appel.

Pour éviter ces calculs non nécessaires, on peut sauvegarder les valeurs de f_n déjà calculées de telle sorte qu'on puisse les lire au lieu de les recalculer au cours de calculs ultérieurs.

- (b) Quel serait une bonne structure de données pour stocker ces f_n ?
(c) Modifiez l'algorithme `fib_rec` de la manière suivante :

- S'il calcule une valeur f_n , il la stocke dans une structure de données définie globalement (à l'extérieur de la fonction)
- Avant de calculer une valeur f_n , il doit vérifier si cette valeur a déjà été calculée auparavant.

Pour tester votre algorithme, vous pouvez sa sortie à celle de `fib_iter` pour des entrées de votre choix. A la différence de `fib_rec`, vous pourrez appeler votre algorithme avec de grandes valeurs de n .

Solution.

- (a) Toutes les instructions s'exécutent en un temps constant et la boucle `for` itère $n - 1$ fois. `fib_iter` a donc un temps de parcours qui est $\Theta(n)$.
- (b) Une solution naturelle serait d'utiliser un dictionnaire, où les clés sont les entiers naturels et la valeur correspondant à la clé n est le nombre de Fibonacci f_n .
On pourrait aussi utiliser une liste où le n -ème nombre de Fibonacci f_n est stocké à l'indice n .
- (c) L'algorithme suivant calcule le n -ème nombre de Fibonacci et utilise un dictionnaire global pour stocker les valeurs déjà calculées.

```
fib_dict = {}

def fib_rec(n):
    """
    Entree: n entier naturel
    Sortie: nieme nombre de Fibonacci
    """
    if n == 0:
        return 0
    if n == 1:
        return 1
    if n in fib_dict:
        return fib_dict[n]

    fib_dict[n] = fib_rec(n-1) + fib_rec(n-2)
    return fib_dict[n]
```

Exercice 8 (facultatif)

Les tours de Hanoi

On dispose de trois colonnes A (la colonne source), B (la colonne intermédiaire) et C (la colonne destination). Sur la colonne A sont empilés n disques percés de diamètres différents, ordonnés du plus grand tout en bas au plus petit tout en haut. Les disques sont numérotés de 1 (le plus petit) à n (le plus grand). Le but est de bouger tous les disques de la colonne A à la colonne C, en respectant les règles du jeu suivantes :

- On peut bouger un seul disque à la fois
- On ne peut bouger qu'un disque qui est tout en haut de sa pile
- On ne peut pas placer un disque au-dessus d'un disque plus petit.

La figure 2 montre la position initiale et la position finale des disques pour $n = 3$.



FIGURE 2 – Tours de Hanoi

- (a) Résolvez le problème à la main pour $n = 1, 2, 3$, c'est-à-dire donnez la liste des mouvements qu'il faut faire pour déplacer tous les disques de la colonne A à la colonne C en respectant les règles du jeu¹.
- (b) Ecrivez un algorithme `hanoi` qui prend en entrée un entier $n \geq 1$ et trois strings contenant les noms des colonnes source, intermédiaire et destination. Votre algorithme doit imprimer la liste de mouvements qu'il faut faire pour déplacer tous les disques de la colonne source à la colonne destination en respectant les règles du jeu. Par exemple, `hanoi(2, 'A', 'B', 'C')` devrait afficher
- ```
Déplacer le disque 1 de A à B
Déplacer le disque 2 de A à C
Déplacer le disque 1 de B à C
```

**Indice :**<sup>2</sup>

---

1. Pour mieux comprendre le problème vous pouvez aussi jouer aux tours de Hanoi [en ligne](#).

2. Pour déplacer une tour de taille  $n$ , il faut mettre de côté la tour des  $n - 1$  plus petits disques, déplacer le disque  $n$ , et ramener la tour de taille  $n - 1$  sur le plus grand disque.

(c) Soit  $D_n$  le nombre de déplacements simulés par votre algorithme pour une entrée de  $n$  disques. Exprimer  $D_n$  sous forme d'une récurrence en exprimant  $D_n$  en fonction de  $D_{n-1}$  pour  $n \geq 2$  et en donnant la valeur du cas de base  $D_1$ .

(d) Prouver que  $D_n = 2^n - 1$ .

Cet algorithme est un exemple d'algorithme exponentiel : son temps de parcours (qui est proportionnel au nombre de déplacements effectués) est une fonction exponentielle du nombre de disques  $n$ .

### Solution.

(a) Pour  $n = 1$  :

— Déplacer le disque 1 de A à C.

Pour  $n = 2$  :

— Déplacer le disque 1 de A à B

— Déplacer le disque 2 de A à C

— Déplacer le disque 1 de B à C.

Pour  $n = 3$  :

— Déplacer le disque 1 de A à C

— Déplacer le disque 2 de A à B

— Déplacer le disque 1 de C à B

— Déplacer le disque 3 de A à C

— Déplacer le disque 1 de B à A

— Déplacer le disque 2 de B à C

— Déplacer le disque 1 de A à C.

On commence à voir un motif qui apparaît : pour déplacer  $n$  disques d'une colonne source à une colonne destination, il faut déplacer la tour des  $n - 1$  plus petits disques sur la colonne intermédiaire, déplacer le disque  $n$  de la colonne source à la colonne destination, et ramener la tour de taille  $n - 1$  sur la colonne destination.

(b) L'algorithme suivant affiche la liste des mouvements à faire pour une entrée de taille  $n$  :

```
def hanoi(n, source, intermediaire, dest):
 if n == 1:
 print("Déplacer le disque", n, "de", source, "à", dest)
 else:
 hanoi(n-1, source, dest, intermediaire)
 print("Déplacer le disque", n, "de", source, "à", dest)
 hanoi(n-1, intermediaire, source, dest)
```

(c) On définit  $D_n$  comme le nombre de déplacements effectués pour déplacer une tour de  $n$  disques d'une colonne à une autre.

Pour le cas de base (colonne de 1 disque), il suffit d'un seul déplacement, donc  $D_1 = 1$ .

Pour déplacer une tour de  $n > 1$  disques, on déplace deux fois une tour de  $n - 1$  disques et on déplace le disque de taille  $n$  une fois. On a donc  $D_n = 2D_{n-1} + 1$ .

Le nombre de déplacements  $D_n$  satisfait donc la récurrence

$$D_n = \begin{cases} 2D_{n-1} + 1, & n > 1, \\ 1, & n = 1. \end{cases}$$

(d) On prouve par récurrence sur  $n$  que  $D_n = 2^n - 1$ .

— Pas de base :

$$D_1 = 1 = 2^1 - 1.$$

— Pas de récurrence : Pour  $n \geq 1$ , supposons que  $D_n = 2^n - 1$ . Alors

$$D_{n+1} = 2D_n + 1 = 2(2^n - 1) + 1 = 2^{n+1} - 2 + 1 = 2^{n+1} - 1.$$

On a donc prouvé que  $D_n = 2^n - 1$ . Le nombre de déplacements effectués par l'algorithme `hanoi` sur  $n$  disques est exponentiel en  $n$ .