Série 9

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 $\lim_{n\to\infty} \frac{g(n)}{f(n)} = +\infty$ alors $f = \mathcal{O}(g)$ et $g \neq \mathcal{O}(f)$.

(b) Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ 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.

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.

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'autre valeurs de `x`.

Exercice 4

Vous avez vu en cours un algorithme de recherche binaire **itératif**. Donnez un algorithme de recherche binaire **récursif** `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 `bas` $\leq$ `i` $\leq$ `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`.

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 $\log(n)$.

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` $\geq$ `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)`.

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 :

```python
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.

Exercice 8 (facultatif)

Les tours de Hanoi

On dispose de trois colonne `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 1 montre la position initiale et la position finale des disques pour $n = 3$.

(a) Position initiale (b) Position finale

FIGURE 1 – 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 jeu [1].

(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 : [2]

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.