Série 7

Sauf si spécifié autrement, tous les algorithmes demandés sont à écrire sous forme d'une fonction Python.
Les exercices précédés d'une astérisque sont optionnels (mais pas nécessairement difficiles).

---

1. On donne l'algorithme suivant:

```
def f(a, b):
    '''
    Input: a, b entiers, a >= b > 0
    '''
    r = a
    while r >= b:
        r -= b

    return r
```

   (a) Donner la valeur de `f(3,2)`, `f(20,16)`, `f(20,5)`.

   (b) Que calcule cet algorithme?

   *(c) Prouver sa correctitude.

Solution.

   (a) `f(3,2) = 1, f(20,16) = 4, f(20,5) = 0.`

   (b) Cet algorithme calcule le reste de la division entière de `a` par `b`.

   *(c) Pour prouver la correctitude de l'algorithme, il faut d'abord prouver que la boucle **while** termine. On considère la variable `r` (la valeur de `r` est une fonction du nombre d'itérations de la boucle). Avant le début de la boucle **while**, `r = a` et donc $\texttt{r} \geq \texttt{b}$ par la spécification du problème. A chaque itération de la boucle, `r` décroît de `b`. Au bout d'un certain nombre d'itérations, `r` atteindra donc une valeur strictement inférieure à `b` et la boucle **while** terminera. A ce moment, on aura que $\texttt{r} < \texttt{b}$, mais aussi que $\texttt{r} \geq \texttt{0}$: sinon la boucle aurait dû s'arrêter plus tôt (si `r` était strictement négatif, on aurait que la valeur de `r` à l'itération précédente était strictement inférieure à `b`, mais alors la boucle aurait dû déjà s'arrêter à l'itération précédente).

   Pendant l'itération de la boucle **while**, on maintient l'invariant suivant:

   `a - r` est un multiple de `b`.

   En effet, avant le début de la boucle, $\texttt{a - r} = 0$ et est donc un multiple de `b`. De plus, à chaque itération de la boucle, `r` est décrémenté de `b` et donc `a - r` est incrémenté de `b`, il reste donc multiple de `b`.

   Lorsque la boucle **while** termine, on a donc que `a - r` est un multiple de `b` et que $0 \leq \texttt{r} < \texttt{b}$, donc que `r` est le reste de la division entière de `a` par `b`.

2. (a) Ecrivez un algorithme `countdown` récursif (sans boucle) qui prend un entier positif `n` en entrée et affiche les nombres de `n` à `0` par ordre décroissant. Par exemple, `countdown(4)` doit afficher

```
4
3
2
1
0
```

N'oubliez pas de tester votre algorithme pour quelques petites valeurs de `n`. Que se passe-t-il si vous appelez votre algorithme avec l'argument `n = -1` ?

(b) Ecrivez un algorithme itératif qui produit le même affichage.

Solution.

(a) L'algorithme suivant produit l'affichage désiré:

```
def countdown(n):
    '''
    Entree: entier n >= 0
    Affiche le compte a rebours n, n-1, ..., 0
    '''
    if n >= 0:
        print(n)
        countdown(n-1)
```

Remarquez que le cas de base est implicitement défini avec l'instruction `if`, puisque `countdown(-1)` va simplement retourner sans rien exécuter. En particulier, l'algorithme ne risque pas de rentrer dans une boucle infinie d'appels récursifs, même si on l'appelle avec un entier négatif en entrée. Un algorithme équivalent qui explicite le cas de base est le suivant:

```
def countdown(n):
    '''
    Entree: entier n >= 0
    Affiche le compte a rebours n, n-1, ..., 0
    '''
    if n < 0:
        return
    print(n)
    countdown(n-1)
```

(b) L'algorithme itératif suivant produit le même affichage:

```
def countdown_iter(n):
    '''
    Entree: entier n >= 0
    Affiche le compte a rebours n, n-1, ..., 0
    '''
    for i in range(n,-1,-1):
        print(i)
```

3. Ecrivez un algorithme récursif (sans boucle) qui prend en entrée une liste `L` non vide de nombres et retourne le maximum de `L`.

   N'oubliez pas de tester votre algorithme sur quelques exemples de `L`.

   Que se passe-t-il si vous appelez votre algorithme avec une liste vide?

   Solution. L'algorithme suivant trouve récursivement le maximum d'une liste:

```python
def max_rec(L):
    '''
    entree: L liste de nombres non vide
    sortie: max de L
    '''
    if len(L) == 1:
        return L[0]
    s = max_rec(L[1:])
    if L[0] > s:
        return L[0]
    return s
```

   Le cas de base correspond à une liste de taille 1, dans quel cas l'unique élément de la liste est le maximum. Pour une liste `L` de taille supérieure à 1, on calcule récursivement le maximum de la sous-liste `L[1:]` et on le compare au premier élément de `L` pour trouver le maximum de `L`.

   Si on appelle `max_rec` avec une liste vide `L` en entrée, le cas de base ne s'appliquera pas et l'appel `max_rec(L[1:])` se fera avec encore une liste vide en argument, entraînant ainsi une boucle infinie d'appels à la fonction. L'interpréteur Python interrompt cette boucle et génère une exception de type `RecursionError` après un certain nombre d'appels récursifs (vous pouvez accéder à ce nombre avec la fonction `getrecursionlimit()` et le modifier avec la fonction `setrecursionlimit()` du module `sys`).

4. Ecrivez une version récursive (sans boucle `while`) de l'algorithme d'Euclide vu en cours.

   Solution. L'algorithme suivant calcule récursivement le plus petit diviseur commun de deux nombres naturels a et b:

```python
def pgcd_rec(a, b):
    '''
    Entree: a, b entiers strictement positifs
    Sortie: plus grand diviseur commun de a et b
    '''
    if b == 0:
        return a

    return pgcd_rec(b, a % b)
```

5. Vous avez vu au cours des algorithmes récursifs pour calculer la factorielle d'un nombre `n` et le `n`-ième nombre de Fibonacci. Dans cet exercice, vous allez implémenter des versions itératives (avec une boucle) de ces algorithmes.

   (a) Ecrivez un algorithme `fact_iter` qui prend en entrée un entier $\texttt{n} \geq \texttt{0}$ et calcule itérativement (avec une boucle, sans appels récursifs) la factorielle de `n`.

   *(b) Prouvez la correctitude de cet algorithme.

   (c) Ecrivez un algorithme `fib_iter` qui prend en entrée un entier $\texttt{n} \geq \texttt{0}$ et calcule itérativement (avec une boucle, sans appels récursifs) le `n`-ième nombre de Fibonacci. Regardez en bas de page pour un **indice**[1].

   *(d) Prouvez la correctitude de cet algorithme.

Solution.

(a) L'algorithme ci-dessous calcule, pour un entier positif `n` en entrée, la factorielle de `n`. Remarquez que l'itération `i` calcule la factorielle de `i`.

```
def fact_iter(n):
    '''
    calcule la factorielle de n
    suppose n entier >= 0
    '''
    fact = 1
    for i in range(1, n+1):
        fact *= i
    return fact
```

*(b) La remarque du point précédent est exactement l'invariant qui est maintenu à travers les itérations de la boucle `for`:

Invariant de boucle: Au début de l'itération `i`, la valeur de `fact` est la factorielle de `i-1`.

- **Initialisation:** avant l'itération `i = 1`, `fact` contient la valeur 1 qui est bien la factorielle de 0.
- **Maintenance:** on suppose qu'au début de l'itération `i`, `fact` contient la valeur $(\texttt{i}-1)!$ (la factorielle de `i-1`). A l'itération `i`, la valeur de `fact` est multipliée par `i`, ce qui nous donne bien que `fact` contient la valeur $\texttt{i} \cdot (\texttt{i}-1)! = \texttt{i}!$ au début de l'itération `i+1`.
- **Terminaison:** A la sortie de la boucle, c'est-à-dire avant l'itération `n+1`, qui n'aura pas lieu, `fact` contient donc la valeur `n`! .

Remarquez aussi que pour `n = 0`, aucune itération de la boucle n'est exécutée et l'algorithme retourne simplement la factorielle de `0`.

(c) L'algorithme ci-dessous calcule, pour un entier positif `n` en entrée, le `n`-ième nombre de Fibonacci $f_{\texttt{n}}$. Remarquez que l'itération `i` calcule le nombre $f_{\texttt{i}}$. Les variables `fib_old` et `fib_new` servent à stocker les deux derniers nombres de Fibonacci ($f_{\texttt{n-2}}$ et $f_{\texttt{n-1}}$ respectivement) dont on aura besoin pour calculer $f_{\texttt{n}}$.

[1] A travers les itérations de la boucle, vous devez vous rappeler de deux valeurs: les deux derniers nombres de Fibonacci calculés.

```python
def fib_iter(n):
    '''
    calcule le n-ieme nombre de Fibonacci
    suppose n entier >= 0
    '''
    fib_old = 0
    fib_new = 1
    for i in range(2,n+1):
        fib_old, fib_new = fib_new, fib_old + fib_new
    if n == 0:
        return fib_old
    return fib_new

N = 15
print(fib_iter(N))
```

*(d) Remarquez d'abord que pour `n = 0` ou `n = 1` , l'algorithme retourne la valeur $f_0 = 0$ et $f_1 = 1$ sans rentrer dans la boucle `for`. Il s'agit des conditions initiales.

La boucle maintient l'invariant suivant:

Invariant de boucle: Au début de l'itération `i`, `fib_old` contient la valeur f_{i-2} et `fib_new` contient la valeur f_{i-1}.

- **Initialisation:** avant l'itération `i = 2`, `fib_old` contient la valeur $0 = f_0$ et `fib_new` contient la valeur $1 = f_1$.
- **Maintenance:** on suppose qu'au début de l'itération `i`, `fib_old` contient la valeur f_{i-2} et `fib_new` contient la valeur f_{i-1} . A l'itération `i`, on affecte simultanément à `fib_old` l'ancienne valeur de `fib_new`, c'est-à-dire f_{i-1}; et à `fib_new` la somme des anciennes valeurs de `fib_old` et `fib_new`, c'est-à-dire $f_{i-2} + f_{i-1} = f_i$. Donc au début de l'itération `i+1`, `fib_old` contient la valeur f_{i-1} et `fib_new` contient la valeur f_i.
- **Terminaison:** A la sortie de la boucle, c'est-à-dire avant l'itération `n+1`, qui n'aura pas lieu, `fib_old` contient donc la valeur f_{n-1} et `fib_new` contient la valeur f_n. Comme l'algorithme retourne `fib_new` à la sortie de la boucle, il retourne bien le `n`-ième nombre de Fibonacci.

---

6. Pour une chaîne de caractères `s`, on définit une **sous-chaîne** de `s` (dans le cadre de cet exercice) comme un sous-ensemble des caractères de `s`, apparaissant dans le même ordre que dans `s`. Par exemple, si `s = "abac"`, alors `"b"`, `"aa"`, `"bc"` et la chaîne vide sont des sous-chaînes de `s`, mais `"ca"` et `"cc"` ne le sont pas.

   On veut écrire un algorithme récursif `sous_chaînes` qui prend en entrée une chaîne de caractères `s` et produit une liste contenant toutes les sous-chaînes contenues dans `s` (en permettant les répétitions, et dans un ordre quelconque).

   Par exemple, `sous_chaînes("abc")` doit retourner (à l'ordre près)
   `['', 'c', 'b', 'bc', 'a', 'ac', 'ab', 'abc']`
   et `sous_chaînes("aa")` doit retourner (à l'ordre près)
   `['', 'a', 'a', 'aa']`.

(a) Pour une chaîne de caractères non vide `s`, supposez que vous avez à disposition la liste `L` des sous-chaînes de `s[1:]`. A partir de cette liste `L`, comment forme-t-on la liste des sous-chaînes de `s`?

(Par exemple, pour la chaîne `s = "abc"`, on a `s[1:] = "bc"` et la liste de sous-chaînes de `s[1:]` est `L = ['', 'c', 'b', 'bc']`.)

Déduisez-en le ou les appels récursifs de l'algorithme.

(b) A quelle chaîne de caractères en entrée correspond le cas de base? Que faut-il sortir dans ce cas?

(c) Donnez l'algorithme `sous_chaines`.

Solution.

(a) Pour une chaîne de caractères non vide `s`, on suppose qu'on a à disposition la liste `L` des sous-chaînes de `s[1:]`. Chaque sous-chaîne de `s` peut soit inclure le caractère `s[0]`, soit l'exclure. Les sous-chaînes de `s` excluant le caractère `s[0]` sont simplement les éléments de `L`; et les sous-chaînes incluant le caractère `s[0]` sont formées en prenant la concaténation de `s[0]` et d'un élément de `L`.

Par exemple, pour la chaîne `s = "abc"`, la liste de sous-chaînes de `"bc"` est `L = ['', 'c', 'b', 'bc']`. La liste de sous-chaînes de `"abc"` contient les éléments de `L`, ainsi que les sous-chaînes `'a', 'ac', 'ab', 'abc'` formées en concaténant le caractère `'a'` à chacune des sous-chaînes dans `L`.

Pour une chaîne de caractères non vide `s`, l'appel à `sous_chaines(s)` fera donc un appel récursif à `sous_chaines(s[1:])`.

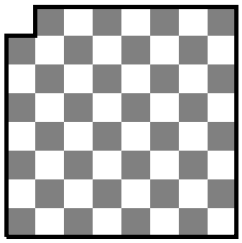
(b) Le cas de base correspond à une chaîne de caractères vide, dans quel cas on retourne une liste contenant uniquement la chaîne de caractères vide.

(Alternativement, on pourrait décider que le cas de base correspond à une chaine de caractères `s` de longueur 1: dans ce cas on retournerait la liste `[s, '']`.)

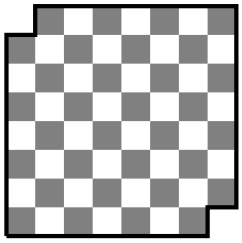
(c) L'algorithme récursif `sous_chaines` est donné ci-dessous:

```
def sous_chaines(s):
    '''
    Entree: chaine de caracteres s
    Sortie: liste des sous_chaines de s
    '''
    if len(s) == 0: #cas de base
        return [s]

    L = sous_chaines(s[1:]) #appel recursif

    L1 = [s[0] + x for x in L]
    L.extend(L1)
    return L
```

* 7. (a) On dispose d'un échiquier où il manque un coin, comme dans la figure ci-dessous, et de dominos pouvant chacun couvrir deux cases adjacentes d'un échiquier. Peut-on couvrir cet échiquier avec ces dominos entièrement et exactement (sans aucun domino qui dépasse)?



(b) Même question pour l'échiquier ci-dessous, avec deux coins manquants.



Solution.

(a) C'est impossible. Un domino couvre deux cases et donc tout algorithme de placement de dominos maintient l'invariant qu'après chaque domino placé, le nombre de cases occupées par des dominos est pair. Or l'échiquier à couvrir a un nombre impair de cases.

(b) C'est aussi impossible. En se rendant compte que chaque domino doit couvrir une case blanche et une case noire, on peut formuler un invariant pour tout algorithme de placement de dominos: après chaque domino placé, le nombre de cases noires occupées est égal au nombre de cases blanches occupées. Or l'échiquier à couvrir a plus de cases noires que de cases blanches.