Correction du DS 3

Julien Reichert

Exercice 1

```
def maximum(L):
    maxi = L[0]
    for i in range(1, len(L)):
        if L[i] > maxi:
            maxi = L[i]
    return maxi
```

La terminaison est garantie par l'absence de boucle conditionnelle. La complexité de chaque tour de boucle étant constante, la complexité de la fonction est linéaire en la taille de la liste en entrée. La preuve de correction de la fonction repose sur l'invariant de boucle suivant : la variable `maxi` contient après le tour de boucle où la variable `i` vaut un certain `k` le maximum des `k+1` premiers éléments de la liste (preuve facile).

Exercice 2

```
def dichotomie(L, x):
    n = len(L)
    ind_deb = 0
    ind_fin = n-1
    while (ind_deb <= ind_fin):
        ind_mil = (ind_deb + ind_fin)//2
        if L[ind_mil] == x:
            return True
        if L[ind_mil] < x:
            ind_deb = ind_mil+1
        else:
            ind_fin = ind_mil-1
    return False
```

La terminaison est garantie par le variant `ind_fin - ind_deb`, qui décroît strictement à chaque tour de boucle (en l'occurrence, il est divisé par deux en gros, ce qui établit par ailleurs la complexité logarithmique). La preuve de correction de la fonction repose sur l'invariant de boucle suivant : à tout moment, la valeur `x` est dans `L` si, et seulement si, elle y est entre les indices `ind_deb` et `ind_fin` inclus. Ainsi, si cet intervalle devient vide, c'est que l'élément est absent de la liste, et si on le trouve, la fonction retourne évidemment le booléen `True`.

Exercice 3

```
def compte_6(LL):
    reussites = 0
    for L in LL:
        ok = False
        for element in L:
            if element == 6:
                ok = True
        if ok:
            reussites += 1
    return reussites
```

Version profitant de l'opérateur `in` :

```
def compte_6(LL):
    reussites = 0
    for L in LL:
        if 6 in L:
            reussites += 1
    return reussites
```

Exercice 4

Version avec des tableaux ici :

```
import numpy as np

def matrice_des_ecarts(m, n):
    tab = np.zeros((n, m))
    for i in range(n):
        for j in range(m):
            tab[i, j] = abs(i-j)
    return tab
```

Les listes de listes sont imaginables, mais moins pratiques.

Exercice 5

Version naturelle :

```
def est_somme(m):
    total = 0
    n = 0
    while total < m:
        n += 1
        total += n
    if total == m:
        return n
    else:
        return -1
```

La complexité est en $\mathcal{O}(\sqrt{\texttt{m}}$ ici, et la preuve de correction repose sur l'invariant de boucle suivant : à chaque étape, la variable `total` vaut la somme des nombres de `1` à `n`, la variable `m` est donc une telle somme si, et seulement si, il s'agit d'une des valeurs de `total` à un moment, et en cas de dépassement il n'y a plus d'espoir de trouver une solution.

On peut également recalculer `total` à chaque étape en utilisant la formule du cours, mais les multiplications sont plus couteuses (à utiliser si on a peur d'avoir inversé deux lignes et de créer un bug, par exemple). Ceci étant, la formule permet de remarquer que les inégalités $k^2 \leq k(k+1) < (k+1)^2$, vraies pour tout entier naturel k, donnent directement la seule valeur possible pour la réponse à la fonction, d'où en temps constant :

```
def est_somme_astuce(m):
    n = int((2*m)**0.5)
    if n*(n+1) == 2*m:
        return n
    else:
        return -1
```

Il n'y a pas de pénalité si on recalcule les valeurs successives de `total` pour chaque `n` en imbriquant une deuxième boucle, mais la complexité devient alors linéaire en `m` car quadratique en sa racine.