INFO-H-100 - Programmation
TP 12 — Révisions

Lorsque l'exercice demande d'écrire une fonction, écrivez la fonction demandée et testez-la avec plusieurs valeurs pertinentes.

Cette séance se passera uniquement sur papier !

Vous désirez aller plus loin ? Allez consulter l'examen de Janvier 2012 et son corrigé sur le site du cours !

Ex. 1. En Python, il est possible de représenter un polynôme de degré n sous la forme d'une liste p de taille $n+1$ où chaque $p[i]$ donne le coefficient d'indice i du polynôme :

$$p^{(n)}(x) = \sum_{i=0}^{n} a_i x^i \iff p[i] = a_i$$

On décide de stocker plusieurs polynômes dans une liste de telle sorte que chaque élément de la liste représente un polynôme donné selon l'encodage fourni ci-dessus.

Lorsqu'on souhaite comparer des polynômes selon leur valeur respectivement obtenue pour un réel x donné très grand, une bonne première approximation consiste à comparer les polynômes selon leur degré et en cas d'égalité selon leur coefficient de ce degré :

$$p_1^{(m)} > p_2^{(n)} \iff (m > n) \vee ((m == n) \wedge (a_m > b_n))$$

où $a_i = p_1[i]$ et $b_j = p_2[j]$.

On demande d'écrire une fonction `poly_sort` qui :
– reçoit une liste de polynômes comme décrit ci-dessus,
– et trie l'ensemble des polynômes en utilisant un tri par insertion et la relation d'ordre fournie.
Veillez à découper ce problème en fonctions.

Ex. 2. La série

$$\sum_{n \geq 2} \frac{1}{n^\alpha}$$

converge pour $\alpha > 1.0$. Pour une valeur `cible` entre 0.1 et 1.0, il existe un α tel que la série ci-dessus converge vers `cible`.

Etant données les constantes globales `EPS` (précision) et `NMAX`, on vous demande d'écrire la fonction `alpha(cible)` qui calcule la valeur de α qui fait converger vers la valeur `cible` donnée en paramètre (à `eps` près) la série :

$$\sum_{n=2..\texttt{nMax}} \frac{1}{n^\alpha}$$

On suppose que `cible` est une valeur entre 0.1 et 1.0 et que le résultat α sera compris entre 1.0 et 100.0. On remarque que plus α est grand, plus la série a une petite valeur. La recherche se fera par *dichotomie* jusqu'à obtenir un résultat suffisamment proche pour la série.

Rappel. Une *recherche dichotomique* d'une valeur x dans un intervalle trié $[bi, bs]$ compare x avec la valeur m au milieu de l'intervalle. Si la recherche n'est pas fructueuse, on réitère la recherche avec la première partie $[bi, m]$ de l'intervalle ou la seconde $[m, bs]$ suivant l'endroit où x est potentiellement présent.

Remarque. Vous pouvez utiliser l'opérateur `**` et la fonction `abs`.

Conseil. Décomposez le problème en deux fonctions :
– la première qui calcule la série pour un α donné et
– la seconde, `alpha(cible)`, qui effectue la recherche dichotomique en utilisant la première fonction.

Ex. 3. Soit un polynome représenté sous la forme d'une liste p où chaque $p[i]$ donne le coefficient d'indice i du polynôme.

Ecrivez une fonction `derivKieme` qui reçoit un polynôme et un entier positif k qui transforme le polynôme en sa dérivée $k^{\text{ième}}$.

La dérivée $k^{\text{ième}}$ d'un polynôme est définie comme suit :

$$P_n^{(k)} = \sum_{i=0}^{n-k} a_{i+k} \frac{(i+k)!}{i!} x^i$$

où $P_n^{(k)}$ est la dérivée $k^{ème}$ du polynôme P_n.

Vous devez veiller à optimiser vos calculs et vous ne pouvez pas utiliser de liste supplémentaire à la liste P elle-même.

Ex. 4. La sécante hyperbolique d'un angle x est calculée à l'aide de la série suivante :

$$sech(x) = \left(1 - \frac{E_2}{2!} \cdot x^2 + \frac{E_4}{4!} \cdot x^4 - \frac{E_6}{6!} \cdot x^6 + \cdots\right)$$

C'est-à-dire :

$$sech(x) = \left(\sum_{n \geq 0} \frac{(-1)^n \cdot E_{2n}}{(2n)!} x^{2n}\right)$$

où E_n représente le n^{eme} nombre de Euler (ou nombre Zig-Zag).

On vous demande d'écrire une fonction `sech`, qui retourne la valeur de $sech(x)$, pour `x` *réel*. Vos calculs s'arrêteront lorsque la valeur du dernier terme ajouté est inférieure, en valeur absolue, à un ε défini comme constante globale dans votre programme (par exemple $\varepsilon = 10^{-4}$), ou lorsque le nombre de termes considérés atteint une borne également définie au sein de votre programme.

Pour répondre à cette question, vous devez faire appel à la fonction `euler(n)`, qui renvoie le n^{eme} nombre d'Euler, cette fonction vous est donnée, vous ne devez ni la définir, ni la déclarer.

Veillez à optimiser les calculs effectués (en particulier les calculs des termes successifs).

Ex. 5. En Python, on peut représenter une matrice à l'aide d'une liste de listes. Un damier d'échec peut être représenté par une matrice où chaque case contient un pion représenté par un tuple (type, couleur). L'absence de pion sera représentée par la valeur `None`.

Par exemple, voici une représentation d'un jeu d'échec dans sa configuration initiale :

```
jeu = [ [ ("tour","noir"), ("cheval","noir"), ("fou","noir"), ("reine","noir"),
          ("roi","noir"), ("fou","noir"), ("cheval","noir"), ("tour","noir") ],
        [ ("pion","noir"), ("pion","noir"), ("pion","noir"), ("pion","noir"),
          ("pion","noir"), ("pion","noir"), ("pion","noir"), ("pion","noir") ],
        [ None,None,None,None,None,None,None,None ],
        [ None,None,None,None,None,None,None,None ],
        [ None,None,None,None,None,None,None,None ],
        [ None,None,None,None,None,None,None,None ],
        [ ("pion","blanc"), ("pion","blanc"), ("pion","blanc"), ("pion","blanc"),
          ("pion","blanc"), ("pion","blanc"), ("pion","blanc"), ("pion","blanc") ],
        [ ("tour","blanc"), ("cheval","blanc"), ("fou","blanc"), ("reine","blanc"),
          ("roi","blanc"), ("fou","blanc"), ("cheval","blanc"), ("tour","blanc") ]
      ]
```

On vous demande d'écrire une fonction qui trie une liste de jeux d'échec (une liste de listes de listes de tuples) de façon décroissante par rapport au nombre de fous noirs en utilisant l'algorithme du tri par sélection.

Pour cela, découpez votre programme en fonctions :
- une fonction qui compte le nombre de fous noirs dans un jeu,
- une fonction qui compare deux jeux et
- les fonctions nécessaires au tri par sélection.

INFO-H-100 - Programmation
TP 12 — Révisions
Corrections

Solution de l'exercice 1:

```
def is_smaller(pol1, pol2):
    """True si pol1 est plus petit (au sens de l'enonce) que pol2"""
    return len(pol1) < len(pol2) or (len(pol1) == len(pol2) and pol1[-1] < pol2[-1])

def poly_sort(ls):
    for i in range(1, len(ls)):
        val = ls[i]
        j = i
        while j > 0 and is_smaller(val, ls[j - 1]):
            ls[j] = ls[j - 1]
            j -= 1
        ls[j] = val

# Un exemple de liste de polynomes
polys = [
    [-3, 0, 2],
    [0, 0, -5, 7],
    [1, -2, 0, 0, 6],
    [-1, 1],
    [-5, 7, -1, 4, 2],
    [-4, 1, 1]
    ]

def print_poly(p):
    """Affichage lisible d'un polynome"""
    text = ''
    for i in range(len(p) - 1, -1, -1):
        if p[i] > 0 and i != len(p) - 1: # Les positifs sauf le 1er...
            text += '+'                 # ... on met un +
        if p[i] != 0:
            if p[i] != 1 or i == 0:     # Sauf pour un 1 devant un x...
                text += str(p[i])       # ... on met le coefficient
            if i > 1:                   # si > 1 on montre le degre
                text += 'x^' + str(i)
            elif i == 1:                # 1er degre on met juste x
                text += 'x'
    print(text)

def print_polys(ls):
    """affichage d'une liste de polynomes"""
    for p in ls:
        print_poly(p)
```

```
>>> poly_sort(polys)
>>> print_polys(polys)
x-1
x^2+x-4
2x^2-3
7x^3-5x^2
2x^4+4x^3-1x^2+7x-5
6x^4-2x+1
```

Solution de l'exercice 2:

```
EPS = 0.001
NMAX = 100

def series(alpha):
    result = 0
    for n in range(2, NMAX + 1):
        result += 1.0/n**alpha
    return result

def alpha(target):
    bi = 1.0
```

```
    bs = 100.0
    m = (bi + bs) / 2.0
    value = series(m)

    while abs(value-target)>=EPS:
        if value < target:
            bs = m
        else:
            bi = m
        m = (bi + bs) / 2.0
        value = series(m)
    return m

print(alpha(0.5))
print(series(alpha(0.5)))
```

Solution de l'exercice 3:

```
def fact(n):
    res = 1
    for i in range(2,n+1):
        res *= i
    return res

def derivKieme(p, k):
    coef = fact(k)
    degree = len(p) - k
    if degree < 0:
        degree = 0
    for i in range(degree):
        p[i] = p[i + k] * coef
        coef *= float(i + 1 + k)/(i + 1)
    del p[degree:]

pol = [2,3,4,5]    #polynome 2 + 3x + 4x2 + 5x3
derivKieme(pol,2)
print(pol)          #polynome 8 + 30x
```

Solution de l'exercice 4:

```
EPS = 0.001
NMAX = 100

def sech(x):
    x2 = float(x) * x
    n2 = 0
    sign = 1
    x_factor = 1
    den = 1
    term = 1
    result = 1
    nb_terms = 1
    while term >= EPS and nb_terms < NMAX:
        nb_terms += 1
        n2 += 2
        sign = -sign
        x_factor *= x2
        den *= (n2) * (n2 - 1)
        term = euler(n2) * x_factor / den
        result += sign * term
    return result
```

Solution de l'exercice 5:

```
def count(game, pawn):
    c = 0
    for ligne in game:
        for case in ligne:
            if case == pawn:
                c+=1
    return c

def compare_games(game1,game2):
    return count(game1,("fou","noir")) > count(game2,("fou","noir"))
```

```
def pos_min_from(ls, i):
    """Min position in list ls, starting with index i.
    """
    res = i
    while i < len(ls):
        if compare_games(ls[i],ls[res]):
            res = i
        i += 1
    return res

def swap(ls, i1, i2):
    ls[i1], ls[i2] = ls[i2], ls[i1]

def sort_games(ls):
    for i in range(len(ls) - 1):
        pos = pos_min_from(ls, i)
        swap(ls, i, pos)

j0 = [[("tour","noir"),None],[("reine","blanc"),("fou","noir")]]
j1 = [[("fou","noir"),("fou","noir")],[None,("fou","noir")]]
j2 = [[("fou","blanc"),None],[("fou","noir"),("fou","noir")]]

games_list = [j0,j1,j2]

sort_games(games_list)
print(games_list)
```

ou

```
def count(game, pawn):
    c = 0
    for line in game:
        for case in line:
            if case == pawn:
                c+=1
    return c

def compare_games(game1,game2):
    return count(game1,("fou","noir")) > count(game2,("fou","noir"))

def pos_min_from(ls, i, compare):
    """Min position in list ls, starting with index i.
    """
    res = i
    while i < len(ls):
        if compare(ls[i],ls[res]):
            res = i
        i += 1
    return res

def swap(ls, i1, i2):
    ls[i1], ls[i2] = ls[i2], ls[i1]

def selection_sort(ls, comparator):
    for i in range(len(ls) - 1):
        pos = pos_min_from(ls, i, comparator)
        swap(ls, i, pos)

g0 = [[("tour","noir"),None],[("reine","blanc"),("fou","noir")]]
g1 = [[("fou","noir"),("fou","noir")],[None,("fou","noir")]]
g2 = [[("fou","blanc"),None],[("fou","noir"),("fou","noir")]]

games = [g0,g1,g2]

selection_sort(games,compare_games)
print(games)
```