Exercices 31 à 34

Exercice 31 (écriture de pseudo-codes).

Pour chaque fonction ci-dessous, proposez un algorithme sous forme de pseudo-code.

(1) La fonction f_1 prend en entrée deux entiers k et n tels que $0 \leqslant k \leqslant n$, et renvoie le nombre d'arrangements de k éléments parmi n, c'est-à-dire le nombre $n(n-1)\dots(n-k+1)$, avec la convention que ce produit vaut 1 si $k = 0$.

(2) La fonction f_2 prend en entrée une liste de L de nombres réels et renvoie le plus petit élément strictement positif de L, ou le mot-clé `aucun` s'il n'y en a aucun.

(3) La fonction f_3 prend en entrée une matrice M et renvoie le nombre de coefficients entiers de M.

(4) La fonction f_4 prend en entrée une chaine de caractères s et renvoie deux chaînes de caractères, l'une constituée uniquement des voyelles (minuscules ou majuscules) de s, et l'autre constituée uniquement des consonnes de s (à elles deux, ces deux chaînes ont donc le même nombre de caractères que s).

(5) La fonction f_5 prend en entrée un polynôme P et renvoie $\lim\limits_{x\to+\infty} P(x)$, élément de $\mathbb{R} \cup \{-\infty, +\infty\}$.

(6) La fonction f_6 prend en entrée un graphe coloré G (c'est-à-dire un graphe simple non orienté pour lequel une couleur est associée à chaque sommet) et renvoie le booléen `vrai` s'il existe deux sommets adjacents de G qui ont la même couleur, et `faux` dans le cas contraire.

Exercice 32 (traduction de pseudo-codes en Python).

Pour chaque fonction ci-dessous, définie par un pseudo-code, proposez une implémentation Python.

(1)
```
fonction sans_cube(n)
    // teste si n n'est divisible par aucun nombre de type p^3 avec p ⩾ 2 entier
    p ← 2
    tant que n ≠ 1
        si n est divisible par p
            si n est divisible par p^3
                retourner faux
            n ← n/p
            si n est divisible par p
                n ← n/p
        p ← p + 1
    retourner vrai
```

(2)
```
fonction somme_poly(P1,P2)
    // calcule la somme de deux polynômes
    // chaque polynôme est représenté par une liste (la composante i est le coefficient de X^i)
    n1 ← taille de P1
    n2 ← taille de P2
    si n1 < n2
        retourner somme_poly(P2,P1) // on se ramène au cas n1 ⩾ n2
    P ← copie de P1
    pour k = 0, 1, ... n2 − 1
        P(k) ← P(k) + P2(k)
    retourner P
```

fonction texte(s,n)

```
// extrait d'une chaîne de caractères s toutes les sous-chaînes maximales de longueur au moins n
// composées uniquement de lettres (caractères 'a' à 'z', majuscules et minuscules)
vérifier que n ⩾ 1
ajouter le caractère '.' à la fin de s
L ← liste vide // résultat
m ← chaîne vide // mot extrait courant
(3) pour tout caractère c de s
    si c est une lettre
        ajouter c à m
    sinon
        si ( taille de m ) ⩾ n
            ajouter m à L
        m ← chaîne vide
retourner L
```

fonction produit_matrices(A,B)

```
// calcule le produit de deux matrices A et B
// chaque matrice est représentée par une liste de lignes
// chaque ligne est représentée par une liste de coefficients
n ← nombre de lignes de A
p ← nombre de colonnes de A
vérifier que p = nombre de lignes de B
q ← nombre de colonnes de B
(4) C ← liste vide
pour i = 0, 1, ..., n − 1
    L ← liste vide
    pour j = 0, 1, ..., q − 1
        ajouter à L la valeur Σ_{k=0}^{p−1} A_ik B_kj
    ajouter L à C
retourner C
```

fonction arité(r)

```
// calcule l'arité d'un arbre (nombre maximal d'enfants d'un nœud de l'arbre)
// chaque nœud est représenté par une liste dont le premier élément est l'étiquette
// et les éléments suivants les enfants (si il y en a)
// en entrée: r est la racine de l'arbre à analyser (supposé non vide)
(5) n ← nombre d'enfants de r
pour tout enfant e de r
    p ← arité(e)
    si p > n
        n ← p
retourner n
```

Exercice 33 (manipulation des types abstraits).

(1) Que vaut la pile P après exécution de l'algorithme ci-dessous ?

```
F ← file_vide()
pour k = 1, 2, ..., 10
    ajouter_élément(k,F)
P ← pile_vide()
tant que F n'est pas vide
    x ← extraire_élément(F)
    ajouter_élément(x,F)
    x ← extraire_élément(F)
    empiler(x,P)
```

(2) Écrire en pseudo-code une fonction **inverse(F)** qui prend en entrée une file F et renvoie une file G, constituée des éléments de F dans l'ordre inverse. Cette fonction pourra faire appel à une pile auxilliaire P.

(3) Que vaut l'arbre A après exécution de l'algorithme ci-dessous ?

```
F ← file_vide()
pour k = 1, 2, ..., 8
    x ← feuille d'étiquette k
    ajouter_élément(x,F)
tant que F contient au moins deux éléments
    x ← extraire_élément(F)
    ajouter_élément(x,F)
    x ← extraire_élément(F)
    y ← extraire_élément(F)
    z ← nœud d'étiquette vide, et d'enfants x et y
    ajouter_élément(z,F)
x ← extraire_élément(F)
A ← arbre de racine x
```

(4) Écrire en pseudo-code une fonction récursive **arité2(r)** qui prend en entrée la racine d'un arbre A et renvoie le nombre maximal de petits-enfants (enfants d'enfants) pour un nœud de A.

Exercice 34 (expressions bien parenthésées).

On a vu en cours (chapitre 3, p. 31-32) que l'on pouvait vérifier le bon parenthésage d'une expression algébrique à l'aide d'une pile. Plus précisément, on analyse la chaîne de caractères codant l'expression avec l'algorithme suivant :

- on balaie les caractères un à un, dans l'ordre
- lorsqu'on tombe sur un symbole ouvrant, on l'empile
- lorsqu'on tombe sur un symbole fermant c :
  - on vérifie que la pile est non vide
  - on dépile un élément
  - on vérifie que cet élément dépilé est bien le symbole ouvrant correspondant au symbole fermant c
- lorsque l'on a balayé toute la chaîne, on vérifie que la pile est vide

(1) Écrire le pseudo-code de la fonction `parenthesage(s)`, qui prend en entrée une chaîne de caractères, et retourne le booléen vrai ou faux selon que la chaîne de caractères est bien parenthésée avec les symboles ( ) [ ] { } ou pas. On utilisera les fonctions `pile_vide()`, `est_vide()`, `empile()` et `dépile()` du type abstrait pile. Pour alléger le pseudo-code, on pourra également utiliser un dictionnaire pour décrire les associations entre symboles ouvrants et fermants (mais ce n'est pas une obligation).

(2) Si l'on implémente les piles à l'aide du type `list` Python, comment code-t-on les opérations suivantes ?

- $P \leftarrow$ pile_vide()
- est_vide(P)
- empile(x, P)
- $x \leftarrow$ dépile(P)

(3) En déduire une implémentation Python du pseudo-code de la question 1 (on traduira directement les opérations sur les piles dans le pseudo-code par des opérations sur le type `list` Python, sans écrire de fonctions supplémentaires).

(4) Vérifier à l'aide de ce code le parenthésage des expressions suivantes :

```
s1 = "f(x+[g(y,z)-3x+1]^2(x+1))+h{g}[x]()-1+y"
s2 = "(({[(A^2)^2]^2}^2)^2"
s3 = "exp(sin(x+1/[x^{1+n}+1)+x^2)/2)"
```

(**5***) Modifier la fonction de la question 3 pour qu'elle renvoie non pas un booléen, mais :

- `None` si l'expression est bien parenthésée;
- un chaîne de caractères décrivant le plus précisément possible l'erreur rencontrée si l'expression n'est pas bien parenthésée.

Tester ensuite cette fonction sur les expressions de la question 4.

Indications

31.1 S'inspirer du pseudo-code de la fonction factorielle vu au chapitre 3 p. 3

31.2 Faire une boucle sur les éléments de L et, dans la boucle, mettre à jour le plus petit élément strictement positif rencontré jusque là (initialisé à aucun).

31.3 Faire deux boucles imbriquées pour parcourir la matrice M, un boucle sur les indices de lignes et une boucle sur les indices de colonnes.
À l'intérieur de la double boucle, mettre à jour un compteur si le coefficient courant est entier.

31.4 Faire une boucle sur les caractères de s, et dans la boucle un test pour savoir si le caractère courant est une voyelle.

31.5 Considérer à part le cas où $P = 0$, puis le cas où P est de degré nul.
Dans les cas restants, considérer c, le coefficient dominant de P, et conclure en fonction de c.

31.6 Faire deux boucles imbriquées: l'une sur les sommets du graphe, l'autre sur les voisins du sommet considéré dans la première boucle.

32.1 La divisibilité de n par p se teste en considérant `n%p` (reste dans la division euclidienne de n par p).

32.2 Pour faire une copie de P_1 on peut utiliser la fonction `list()` ou la syntaxe `P1[:]`.

32.3 Attention, les chaînes de caractères sont immutables; pour ajouter le caractère '.' à la fin de s, il faut donc écrire `s = s+'.'`
Pour tester si un caractère `c` est une lettre minuscule, on peut écrire `c>='a' and c<='z'`

32.4 Le nombre de lignes de `A` est simplement `len(A)`; le nombre de colonnes de `A` est `len(A[0])`
Le coefficient A_{ik} s'écrit `A[i][k]`.
Pour calculer la somme cherchée, on peut utiliser une compréhension de liste

32.5 Le nombre d'enfants de `r` est simplement `len(r)-1`.
Pour parcourir les enfants `e` de la racine `r`, il suffit d'écrire `for e in r[1:]:`

33.1 Exécuter pas-à-pas les instructions du pseudo-code, en modifiant au fil des instructions la file F et la pile G (voir le cours chapitre 3 p. 23 et 27)

33.2 Extraire les éléments de F un à un à l'aide d'une boucle, et les empiler au fur et à mesure dans la pile P (initialisée vide).
Ensuite, dépiler les éléments de P un à un à l'aide d'une boucle, et les ajouter au fur et à mesure à la liste G (initialisée vide).

33.3 Exécuter pas-à-pas les instructions du pseudo-code, en modifiant au fil des itérations de la boucle la file F, qui contient des arbres en construction (au départ, uniquement des feuilles).

33.4 On peut s'inspirer de la fonction **arité()** de l'exercice 32 question 5.
Commencer par calculer le nombre de petits-enfants de r en sommant le nombre d'enfants des enfants de r (faire une boucle sur les enfants de r).
Ensuite, appliquer la fonction **arité2()** à tous les enfants de r et prendre la valeur maximale entre n et toutes les valeurs ainsi obtenues.

34.1 Voir le cours chapitre 3 p.32 pour un pseudo-code sans les symboles { et }.
On peut utiliser un dictionnaire qui à chacun des 3 symboles ouvrants fait correspondre le symbole fermant associé.
Si l'on rencontre une clé du dictionnaire, il faut l'empiler (symbole ouvrant).
Si l'on rencontre une valeur du dictionnaire, il faut vérifier qu'elle correspond bien à la valeur associée par le dictionnaire à l'élément au sommet de la pile P.

34.2 On peut s'inspirer du cours chapitre 3 p.28, étant entendu qu'ici on ne cherche pas à écrire des fonctions.

34.3 Traduire le pseudo-code de la question 1 en implémentant directement les opérations sur la pile P à l'aide des instructions sur le type `list` rappelées à la question 2.

34.4 Appeler la fonction **parenthésage()** avec comme argument `s1` (défini comme dans l'énoncé), puis `s2`, puis enfin `s3`.

Vérifier que la valeur retournée par la fonction est correcte (`True` pour `s1`, qui est bien parenthésée, `False` pour `s2` et `s3`, qui sont mal parenthésées).

34.5* En cas de chaîne mal parenthésée, on peut par exemple retourner la partie déjà lue de la chaîne `s` au moment où l'on rencontre l'erreur, suivie d'une chaîne expliquant quel caractère était attendu.
Plutôt qu'une boucle `for c in s:`, utiliser plutôt une boucle avec un indice (`for i in range(len(s)):`) puis poser `c = s[i]`. Ceci permet d'extraire ensuite facilement la partie de `s` déjà analysée, avec `s[:i]`.