Informatique – T.D. 3

1. *Utilisation d'une pile pour tester le parenthésage d'une expression*

Dans de nombreux cas, la première étape de l'analyse syntaxique d'une expression est la vérification du parenthésage. Une *expression bien parenthésée* (représentée par une chaîne de caractères) est définie (récursivement) comme étant :

- soit une chaîne ne contenant aucune parenthèse (éventuellement vide !) ;
- soit une expression bien parenthésée entre parenthèses ;
- soit la concaténation de deux expressions bien parenthésées.

Pour contrôler le parenthésage d'une expression, il s'agit de vérifier que chaque parenthèse fermante est associée à une parenthèse ouvrante située à sa gauche et que toutes les parenthèses ouvrantes sont refermées. D'où l'idée d'utiliser une pile : on parcourt de gauche à droite la chaîne représentant l'expression et on empile les positions des parenthèses ouvrantes. Lorsqu'on rencontre une parenthèse fermante, on dépile, le sommet de la pile étant alors la position de la parenthèse ouvrante associée (en option, on peut afficher le couple des positions des deux parenthèses associées).

Programmer ce contrôle en Python, sans oublier de gérer les erreurs !

2. *Implémentation d'une file d'attente*

Le type abstrait de données "files d'attente" est une structure linéaire de stockage munie des primitives suivantes :

- `est_file_vide(f)` : renvoie `Vrai` si la file `f` est vide, `Faux` sinon ;
- `file_vide()` : renvoie la file vide ;
- `ajouter(x,f)` : ajoute le *client* `x` **à la fin** de la file ;
- `suivant(f)` : renvoie le client **en tête** de la file **non vide** `f` **et le supprime** de `f`.

a) Proposer une implémentation en Python où la file est stockée dans une simple liste. Évaluer la complexité des fonctions `ajouter` et `suivant`.

b) Pour obtenir une complexité en temps constant (ou temps constant amorti) des **deux** fonctions `ajouter` et `suivant`, on se propose de stocker une file à l'aide de deux piles :
  - ∗ la première contient le début de la file, le premier client à servir se trouvant au sommet ;
  - ∗ la deuxième contient la fin de la file, le dernier client entré se trouvant au sommet.

  Mettre en œuvre ce principe en Python ; concrètement, la file `f` pourra être définie comme une liste de deux piles `[p0,p1]` ; expliquer son intérêt du point de vue de la complexité.

  Outre les primitives ci-dessus, on programmera l'affichage du contenu de la file.

3. Pour évaluer une expression arithmétique postfixée à l'aide d'une pile, on se limite pour simplifier à des additions et multiplications sur des entiers.

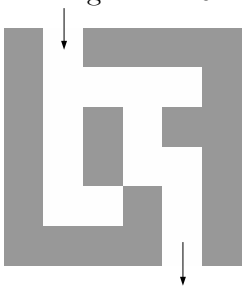
Écrire une unique fonction qui contrôle et évalue (à l'aide d'une pile !) une expression arithmétique postfixée fournie sous forme d'une chaîne de caractères. Ladite chaîne contiendra les différents symboles (nombres et opérateurs), séparés par des espaces.

Pour extraire les symboles, on peut parcourir la chaîne à la recherche des espaces, ou bien utiliser la méthode `split` de Python, qui — appliquée à une chaîne — renvoie la liste des constituants de ladite chaîne, constituants délimités par un ou des séparateurs que l'on peut préciser parmi les paramètres (cf. l'aide de Python). Mais les espaces font partie des séparateurs par défaut, ainsi `'3 4 +'.split()` renvoie `['3','4','+']`.

Pour convertir en nombre une chaîne représentant un nombre, il suffit de lui appliquer le *transtypage* : `int('7')` renvoie le nombre 7...

4. *Exploration d'un labyrinthe*

On s'intéresse au problème de la recherche d'une sortie dans un labyrinthe. On considérera des labyrinthes bâtis à partir d'une grille de 0 et de 1 comme ci-dessous :



0	1	0	0	0	0
0	1	1	1	1	0
0	1	0	1	0	0
0	1	0	1	1	0
0	1	1	0	1	0
0	0	0	0	1	0

Un labyrinthe est une grille de taille $n \times n$ où :

- les 0 représentent les murs et les 1 représentent les couloirs du labyrinthe ;
- l'entrée se fait par la case $(0, 1)$ et la sortie par la case $(n-1, n-2)$;
- le labyrinthe est entouré de murs, à l'exception de ces deux cases.

L'objectif est d'écrire un programme qui détermine s'il existe un chemin de l'entrée vers la sortie en se déplaçant vers le haut, le bas, la gauche ou la droite (mais pas en diagonale).

On pourra stocker le tableau `laby` sous forme d'une liste de listes, l'accès à la case `(i,j)` se faisant alors par `laby[i][j]`.

a) Écrire une fonction `casesVoisines(laby,i,j)` prenant en arguments un tableau `laby` et deux entiers `i` et `j`, et renvoyant la liste des cases adjacentes à la case `(i,j)` qui ne sont pas des murs.

Dans l'exemple ci-dessus, `casesVoisines(laby,1,1)` doit renvoyer `[(0,1),(1,2),(2,1)]`.

Attention à gérer les cases du bord !

b) *Parcours du labyrinthe*

On va explorer tous les couloirs de proche en proche jusqu'à trouver la sortie. Pour cela :

- on représente les cases par des couples d'indices (i, j) ;
- on stocke les cases à explorer dans une pile `p` ;
- on marque les cases visitées.

Algorithme :

- **Initialisation :** aucune case visitée, pile vide
- **Première étape :** on marque la case d'entrée, on la met dans la pile
- **Itération :** tant que la pile n'est pas vide, on dépile une case de la pile `p`, et l'on traite chacune des cases adjacentes qui n'est pas un mur
  - si elle est déjà marquée, on ne fait rien ;
  - si elle n'est pas marquée, on la marque et on l'empile dans `p`.
- **Fin :** les cases marquées sont exactement les cases accessibles depuis l'entrée.

Implémenter la fonction `parcours(laby)` prenant en argument le tableau `laby` représentant le labyrinthe, renvoyant un booléen indiquant s'il existe un chemin de l'entrée jusqu'à la sortie et plaçant 2 à la place de 1 dans les cases de `laby` accessibles depuis l'entrée.

Évaluer la complexité de cette fonction.

N.B. Pour afficher graphiquement le labyrinthe associé au tableau `laby`, on peut utiliser la commande `matshow` de `matplotlib`. On peut même choisir les couleurs associées aux valeurs entières présentes dans `laby` en définissant une *palette de couleurs* personnalisée grâce à la fonction `ListedColormap` du module `colors` de `matplotlib`. Par exemple :

```
plt.matshow(laby, cmap = colors.ListedColormap(['black', 'white', 'blue']))
```

5. *Application du parcours de labyrinthe au problème de percolation en milieu aléatoire*

La *percolation* est l'écoulement d'un liquide à travers un milieu poreux (eau à travers de la mouture de café...). L'objectif est de décider si l'eau peut traverser un milieu donné.

- **Modélisation :** on considère un milieu où les pores sont disposés aléatoirement. On le modélise par une grille dont chaque case est aléatoirement pleine ou vide. L'eau s'écoule en passant dans les cases vides.

- **Réduction au problème du labyrinthe :** étant donné un tableau rempli de 0 (cases pleines) et de 1 (cases vides) modélisant un milieu poreux `m`, on veut obtenir une instance `laby` du problème du labyrinthe telle que la fonction `parcours` appliquée à `laby` réponde au problème de la percolation pour `m`. Pour cela, il suffit de :
  - **encadrer le milieu :** le tour du milieu sera rempli de 0 ;
  - **ajouter une entrée :** la ligne d'indice 0 est bien remplie de 0 sauf un 1 sur la deuxième case ;
  - **ajouter une ligne d'écoulement à l'entrée :** la ligne d'indice 1 est remplie de 1 sauf les deux extrémités ;
  - **ajouter une ligne d'écoulement à la sortie :** la ligne d'indice -2 est remplie de 1 sauf les deux extrémités ;
  - **ajouter une sortie :** la ligne d'indice -1 est bien remplie de 0 sauf l'avant-dernière case.

a) *Construction du milieu poreux* : écrire une fonction `creerMilieu(n,p)` prenant en arguments un entier `n` et un flottant `p` de $[0, 1]$, et renvoyant un tableau `milieuPoreux` de dimension `n`×`n` représentant le milieu poreux. Ce tableau sera rempli comme suit :
- les deux premières lignes, les deux dernières lignes, la première et la dernière colonne seront comme expliqué précédemment ;
- les autres cases seront chacunes vides avec probabilité p (on utilisera le test `random.random() < p` qui est vrai avec probabilité p).

b) *Affichage de l'écoulement* : écrire une fonction `afficherEcoulement(n,p)` qui crée un milieu en appelant `creerMilieu(n,p)` et qui, ensuite, affiche en noir les cases pleines, en bleu les cases vides où l'eau s'écoule et en blanc les autres.

c) *Estimation de la probabilité d'écoulement* : écrire une fonction `estimerProba(n)` réalisant une estimation numérique de la probabilité P que l'eau traverse le milieu en fonction de la probabilité p que chaque case soit vide, en faisant plusieurs itérations pour chaque valeur.

d) *Seuil critique* : on observe un effet de seuil, en-dessous d'une certaine valeur critique p_c, l'eau ne traverse presque jamais. Cet effet de seuil est d'autant plus marqué que n est grand. Écrire une fonction `valCritique(n,eps)` prenant en argument la taille n du milieu et un flottant $\varepsilon > 0$ et calculant une valeur approchée de p_c à ε près à l'aide d'un algorithme dichotomique.

e) *Illustration du seuil* : Tester la fonction `afficherEcoulement(n,p)` avec $n = 200$ et successivement $p = 0.95 * p_c$, $p = p_c$ et $p = 1.05 * p_c$.