Dauphine | PSL UNIVERSITÉ PARIS

M2 ID
TRANSPARENCE DES ALGORITHMES
POUR LA DÉCISION

TD1 : INTRODUCTION AU LANGAGE PYTHON
PARTIE 2

Exercice 1 : Exercice sur les nombres

1. **Nombre Parfait**

   Ecrire une fonction qui décide si un nombre est parfait, c'est-à-dire s'il est égal à la somme de ses diviseurs (par exemple, 6 est un nombre parfait).

   Ecrire une fonction qui étant donné un entier n, calcule la liste des entiers parfaits inférieurs à n.

2. **Nombre heureux**

   Un nombre heureux est un nombre entier non nul qui, lorsqu'on ajoute les carrés de ses chiffres, puis les carrés des chiffres de ce résultat et ainsi de suite jusqu'à l'obtention d'un nombre à un seul chiffre, donne 1 pour résultat.

   Par exemple, 7 est un nombre heureux :

   - $7^2 = 49$
   - $4^2 + 9^2 = 97$
   - $9^2 + 7^2 = 130$
   - $1^2 + 3^2 + 0^2 = 10$
   - $1^2 + 0^2 = 1$

   Par contre, 2 n'est pas un nombre heureux :

   - $2^2 = 4$
   - $4^2 = 16$
   - $1^2 + 6^2 = 37$
   - $3^2 + 7^2 = 58$
   - $5^2 + 8^2 = 89$
   - $8^2 + 9^2 = 145$
   - $1^2 + 4^2 + 5^2 = 42$
   - $4^2 + 2^2 = 20$
   - $2^2 + 0^2 = 4$

   On détecte alors un cycle car on a déjà rencontré le nombre 4, donc on peut arrêter la vérification. Par conséquent 2 n'est pas heureux.

   (a) Écrire une fonction python `estHeureux(n)` qui indique si un nombre entier `n` est heureux.

   (b) Écrire une fonction python `listeNombreHeureuxInf(100)` qui retourne la liste des nombres heureux inférieurs ou égaux à 100.

Exercice 2 : Arbre binaire

Nous donnons ci-dessous l'exemple de l'arbre binaire `a1` ayant l'entier 4 comme racine, 10 nœuds dont 4 feuilles et 5 nœuds internes. Notons que les arbres vides sont représentés comme des dictionnaires vides, et qu'un arbre non vide contient

exactement les 3 clés suivantes : `'rac'` pour racine, `'g'` pour désigner le sous-arbre gauche, et `'d'` pour le sous-arbre droit.

```
a1 = { 'rac' : 4,
       'g' :{'rac' : 2,
             'g' : {},
             'd' : { 'rac' : 3,
                     'g' : {},
                     'd' : {}
                   }
           },
       'd' : {'rac' : 7,
              'g' : {'rac' : 5,
                     'g' : {},
                     'd' : {'rac' : 6,
                            'g' : {},
                            'd' : {}
                           }
                   },
              'd' : {'rac' : 10,
                     'g' : {'rac' : 9,
                            'g' : {},
                            'd' : {}
                           },
                     'd' : {'rac' : 14,
                            'g' : {},
                            'd' : {}
                           }
                   }
            }
     }
```

1. Écrire un prédicat `est_abr(a)` qui teste si `a` est un objet Python représentant un arbre binaire, conformément à la représentation choisie ci-dessus.
2. Écrire une fonction `build_abr(r,g,d)` qui à partir d'une valeur `r`, un arbre `g`, et un arbre `d`, construit un arbre de racine `r`, de sous-arbre gauche `g` et de sous-arbre droit `d`.
3. Écrire les fonctions d'observation des arbres `est_vide`, `racine`, `gauche`, `droit` qui respectivement teste si un arbre est vide, fournit la valeur de la racine d'un arbre non vide, fournit le sous-arbre gauche d'un arbre non vide, et enfin le sous-arbre droit.
4. Écrire les fonctions usuelles portant sur les arbres :
   (a) `hauteur` (longueur maximale du chemin de la racine à une feuille)
   (b) `est_feuille` prédicat testant si un arbre est réduit à une feuille, i.e. une racine avec des sous-arbres vides
   (c) `nombre_feuilles` comptant le nombre de feuilles d'un arbre
   (d) `nb_noeuds_internes` comptant le nombre de nœuds internes, i.e. les nœuds ayant au moins un sous-arbre non vide
5. Écrire les fonctions de parcours des arbres :
   (a) parcours infixe (le sous-arbre gauche est parcouru avant la racine, puis le sous-arbre droit. Avec l'arbre `a1`, cela donne : [2, 3, 4, 5, 6, 7, 9, 10, 14])

(b) parcours préfixe (la racine est parcourue d'abord, puis le sous-arbre gauche et enfin le sous-arbre droit. Avec l'arbre `a1`, cela donne : [4, 2, 3, 7, 5, 6, 10, 9, 14])

(c) parcours suffixe (les sous-arbres gauche et droit sont parcourus d'abord, puis la racine. Avec l'arbre `a1`, cela donne : [3, 2, 6, 5, 9, 14, 10, 7, 4])

(d) parcours en largeur (la racine est d'abord parcouru, puis les nœuds distants de 1 de gauche à droite, puis les nœuds distants de 2 de gauche à droite. Avec l'arbre `a1`, cela donne : [4, 2, 7, 3, 5, 10, 6, 9, 14])

6. On s'intéresse à partir de cette question, aux arbres binaires de recherche, c'est-à-dire vérifiant que pour chacun des nœuds, la racine est plus grande (resp. petite) que tous les noeuds du sous-arbre gauche (resp. droit), et que les sous-arbres gauche et droit sont eux-mêmes de recherche.

   Écrire une fonction `est_de_recherche` qui teste si un arbre binaire est un arbre binaire de recherche, et une fonction `member_recherche` qui teste si un élément appartient ou non à un arbre binaire de recherche.

7. Écrire une fonction `insertion_feuille` qui insère un nouvel élément à la racine de l'arbre en respectant la propriété "est de recherche"

8. Écrire une fonction `insertion_racine` qui insère un nouvel élément à l'une des feuilles de l'arbre en respectant la propriété "est de recherche"

9. Écrire une fonction `appartient` qui teste l'appartenance de l'élément x à l'arbre binaire de recherche a

10. Écrire une fonction `getMin` qui renvoie le plus petit élément de a en supposant que l'arbre n'est pas nul. Rappel : dans un arbre binaire de recherche, le plus petit élément se trouve tout en bas à gauche

11. Écrire une fonction `suppMin` qui supprime le plus petit élément de a en supposant que a n'est pas nul. La fonction ne modifiera pas l'arbre d'origine mais retournera un nouvel arbre.

12. Écrire une fonction `suppression` qui enlève un élément de l'arbre en préservant la propriété "est de recherche" (lorsque l'élément est rencontré, il est remplacé par l'élément le plus à droite du sous-arbre gauche, lorsqu'il n'est pas vide).