TP3

December 10, 2015

1 Récursivité

1.1 Quelques exercices pour commencer

Donner une implantation récursive pour chacune des fonctions suivantes.

In [2]:
```
"""
Retourne la somme des entiers de 1 à n
Si le paramètre n est inférieur à 1, la fonction retourne 0
Input :
    - n, un entier
"""
def sommeN(n):
    pass # remplacer par l'implantation
```

In [3]:
```
assert(sommeN(10) == 55)
assert(sommeN(20) == 210)
assert(sommeN(1) == 1)
assert(sommeN(0) == 0)
assert(sommeN(-5) == 0)
```

In [6]:
```
"""
Retourne la somme des entiers de i à n.
Si i > n, la fonction retourne 0
Input :
    - i, un entier
    - n, un entier
"""
def sommeIN(i, n):
    pass # remplacer par l'implantation
```

In [7]:
```
assert(sommeIN(1,10) == 55)
assert(sommeIN(1,20) == 210)
assert(sommeIN(-5,5) == 0)
assert(sommeIN(-2,4) == 7)
assert(sommeIN(-10,0) == -55)
```

In [3]:
```
"""
Retourne le nombre de chiffres dans l'écriture décimale du nombre n (positif)
Input :
    - n, un entier positif
"""
def chiffresPos(n):
    pass # remplacer par l'implantation
```

In [11]:
```
assert(chiffresPos(1235) == 4)
assert(chiffresPos(0) == 1)
assert(chiffresPos(9) == 1)
assert(chiffresPos(5642164132312431) == 16)
```

In [12]:
```
"""
Retourne le nombre de chiffres dans l'écriture décimale du nombre n (quelconque)
Input :
    - n, un entier
"""
def chiffres(n):
    pass # remplacer par l'implantation
```

In [13]:
```
assert(chiffres(1235) == 4)
assert(chiffres(-1235) == 4)
assert(chiffres(0) == 1)
assert(chiffres(9) == 1)
assert(chiffres(-9) == 1)
assert(chiffres(5642164132312431) == 16)
assert(chiffres(-5642164132312431) == 16)
```

In [15]:
```
"""
Retourne l'écriture binaire du nombre n (positif)
remarque : en python, l'addition des chaînes de caractères permet de les concaténer
Input :
    - n, un nombre entier positif
Output :
    une chaîne de caractère représentant n en écriture binaire
"""
def binairePos(n):
    pass # remplacer par l'implantation
```

In [19]:
```
assert(binairePos(4) == "100")
assert(binairePos(2) == "10")
assert(binairePos(58) == "111010")
assert(binairePos(27) == "11011")
assert(binairePos(0) == "0")
assert(binairePos(1) == "1")
```

In [20]:
```
"""
Retourne l'écriture binaire du nombre n (quelconque)
remarque : en python, l'addition des chaînes de caractères permet de les concaténer
Input :
    - n, un nombre entier
Output :
    une chaîne de caractère représentant n en écriture binaire
"""
def binaire(n):
    pass # remplacer par l'implantation
```

In [23]:
```
assert(binaire(4) == "100")
assert(binaire(-4) == "-100")
assert(binaire(2) == "10")
assert(binaire(-2) == "-10")
assert(binaire(58) == "111010")
```

```
assert(binaire(-58) == "-111010")
assert(binaire(27) == "11011")
assert(binaire(-27) == "-11011")
assert(binaire(0) == "0")
assert(binaire(1) == "1")
assert(binaire(-1) == "-1")
```

In [4]:

In [ ]:

In [ ]:

1.2 Génération récursive

La récursion est particulièrement utile lorsqu'il s'agit de *générer récursivement des listes d'objets*. Par exemple, je veux une fonction qui puisse me retourner la liste des mots binaires d'une taille donnée, c'est à dire *la liste de toutes les chaînes de caractères de taille n contenant uniquement les lettres "0" ou "1"*. Pour $n = 3$, une telle fonction me retournerait l'objet suivant :

```
In [10]: binaire3 = ["000","001","010","011","100","101","110","111"]
         print len(binaire3)
         print binaire3
```

```
8
['000', '001', '010', '011', '100', '101', '110', '111']
```

Voici les fonctions qui génèrent les mots de taille 0, 1 et 2 :

```
In [21]: def getBinaire0():
             return [""]

         def getBinaire1():
             return ["0", "1"]

         def getBinaire2():
             result = []
             for word in getBinaire1():
                 result.append(word + "0")
                 result.append(word + "1")
             return result
```

```
In [22]: print getBinaire0()
         print getBinaire1()
         print getBinaire2()
```

```
['']
['0', '1']
['00', '01', '10', '11']
```

Sur le même modèle, écrivez une fonction *getBinaire3*.

```
In [13]: """
         Retourne la liste des mots binaires de taille 3
         """
         def getBinaire3():
             pass # remplacer par l'implantation
```

```
In [14]: binaire3 = ["000","001","010","011","100","101","110","111"]
         assert(set(binaire3) == set(getBinaire3()))
```

Maintenant que vous avez compris le principe, **écrivez une fonction qui génère les mots binaires de taille n**. Remarque : votre fonction doit être *all inclusive*, n'utilisez pas les fonctions *getBinaireX* écrites précédemment.

```
In [26]: """
         Retourne la liste des mots binaires de taille n
         Input :
             - n, un entier
         """
         def getBinaire(n):
             pass # remplacer par l'implantation
```

```
In [27]: binaire3 = ["000","001","010","011","100","101","110","111"]
         assert(set(binaire3) == set(getBinaire(3)))
         assert(len(set(getBinaire(4))) == 16)
         assert(len(set(getBinaire(5))) == 32)
```

Remarque pour ceux qui connaissent bien python : les tests ont été écrits pour passer aussi si vous retourner un générateur (avec yield) au lieu d'une liste.

Sur le même modèle, implanter la fonction suivante. (un peu plus dur)

Posez-vous la question de cette façon : si j'ai un mot de taille n qui termine par 0 et qui contient k 1, combien de 1 contenait le mot taille n-1 à partir duquel il a été créé ? De même s'il termine par 1.

```
In [29]: """
         Retourne la liste des mots binaires de taille n contenant exactement k fois 1
         Input :
             - n, un entier, la taille des mots
             - k, un entier, le nombre de 1
         """
         def getBinaireK1(n, k):
             pass # remplacer par l'implantation
```

```
In [32]: assert(len(set(getBinaireK1(0,0))) == 1)
         assert(len(set(getBinaireK1(0,1))) == 0)
         assert(set(getBinaireK1(1,1)) == {"1"})
         binaire3_2 = ["011","101","110"]
         assert(set(getBinaireK1(3,2)) == set(binaire3_2))
         assert(len(set(getBinaireK1(7, 3))) == 35)
```

Et pour finir (si vous ne voyez pas comment faire, passez à la suite...)

```
In [2]: """
        Retourne la liste des mots binaires de taille n contenant k fois 1
        tel que, dans chaque préfixe, le nombre de 1 soit toujours supérieur ou égal au nombre de 0
        Input :
            - n un entier, la taille
            - k un entier, le nombre de 1
        Remarque : en particulier, si n-k > k, on retourne la liste vide.
        """
        def catalan(n, k):
            pass # remplacer par l'implantation
```

```
In [ ]: assert(len(set(catalan(0,0))) == 1)
        assert(len(set(catalan(0,1))) == 0)
        assert(len(set(catalan(1,0))) == 0)
        assert(set(catalan(1,1)) == {"1"})
        catalan3_2 = ["110","101"]
        assert(set(catalan(3,2)) == set(catalan3_2))
        assert(len(set(catalan(10,5))) == 42)
        assert([len(set(catalan(2*n,n))) for n in xrange(6)] == [1,1,2,5,14,42])
```

Exécutez la ligne suivante et copiez la liste de nombre obtenu dans google, vous pouvez vous instruire en lisant les résultat. Êtes vous capable d'écrire un programme qui calcule ces nombres plus rapidement ?

```
In [ ]: [len(set(catalan(2*n, n))) for n in xrange(8)]
```

```
In [ ]:
```

1.3 Le problème du porte monnaie

Le problème est le suivant : je possède un certain ensemble de pièces s, puis-je rendre la somme N en utilisant mes pièces ?

Par exemple, si je possède les pièces $s = \{5, 5, 2, 2, 2\}$, je peux rendre les sommes 9 et 14 mais pas 8 et 15. Notez que s représente *exactement* les pièces que j'ai et pas uniquement la valeur de mes pièces (je ne peux pas utiliser une 3ème pièce 5 pour rendre le nombre 15)

Ce problème est typiquement récursif.

On cherche à implanter la fonction *rendMonnaie* qui prend en paramètre une liste de pièces et la valeur à rendre. **Lire d'abord l'aide et les rappels syntaxiques**

Aide : le principe de la récursion est le suivant, on choisis une pièce p de s et on répond aux deux questions suivantes

1. puis-je rendre la monnaie en utilisant p ? Si oui, alors je peux rendre la monnaie
2. Si non, puis-je rendre la monnaie sans utiliser p ?

Si aucune des deux possibilités n'est vérifiée, alors je ne peux pas rendre la monnaie.
Réfléchissez aux questions suivantes puis implantez l'algorithme.

- Que signifie le cas $n = 0$? que dois-je répondre ?
- Que signigifie le cas où $n \neq 0$ et s est vide ? que dois-je répondre ?
- Si je ne suis dans aucun de ces deux cas alors : $n \neq 0$ et s n'est pas vide. Je peux choisir une pièce p de s et tester 1. et 2., à chaque fois, quelles seront les nouvelles valeurs de n et s ?

** Rappels sur la syntaxe des listes ** (cf ci-dessous)

Attention : lorsqu'une une liste est passée en paramètre d'une fonction et *modifiée* la liste d'origine est aussi modifiée ! (passage par référence)

```
In [ ]: L = [1,2,3]
        print L
        p = L.pop()
        print p
        print L
        L.append(p)
        print L
```

```
In [ ]: def lesListesSontPasseesParRef(l):
            l.append(3)

        L = [1,2,3]
```

```
print L
lesListesSontPasseesParRef(L)
print L
```

** La fonction à implanter : **

In [3]:
```
"""
Retourne vrai s'il est possible de rendre le nombre n à partir des pièces contenues dans s
Input :
    - n, un entier positif (la valeur)
    - s, une liste d'entiers (les pièces)
"""
def rendreMonnaie(n, s):
    pass # remplacer par l'implantation
```

In [5]:
```
s = []
assert(rendreMonnaie(0,s))
assert(not rendreMonnaie(10,s))
s = [1]
assert(rendreMonnaie(1,s))
assert(not rendreMonnaie(2,s))
s = [5,5,2,2,2]
assert(not rendreMonnaie(8,s))
assert(rendreMonnaie(9,s))
assert(rendreMonnaie(14,s))
assert(not rendreMonnaie(15,s))
```

On adapte un peu la fonction de départ car on veut récupérer la solution sous la forme de la liste des pièces utilisées. **Implanter la variante ci-dessous**.

In [7]:
```
"""
Retourne une solution au problème du rendu de monnaie ou 'None' s'il n'y a pas de solution.
Input :
    - n, un entier positif (la valeur)
    - s, une liste d'entiers (les pièces)
OUTPUT :
    - la liste des pièces effectivement utilisées.
"""
def rendreMonnaieSolution(n, s):
    pass # remplacer par l'implantation
```

In [9]:
```
s = []
assert(rendreMonnaieSolution(0,s) == [])
assert(rendreMonnaieSolution(10,s) is None)
s = [1]
assert(rendreMonnaieSolution(1,s) == [1])
assert(rendreMonnaieSolution(2,s) is None)
s = [5,5,2,2,2]
assert(rendreMonnaieSolution(8,s) is None)
L = rendreMonnaieSolution(9,s)
assert(L is not None and sum(L)==9)
L = rendreMonnaieSolution(14,s)
assert(L is not None and sum(L)==14)
assert(rendreMonnaieSolution(15,s) is None)
```

Combien d'appels récursifs sont nécessaires pour résoudre le problème ? **Utiliser une variable globale et modifiez la fonction précédente pour compter le nombre d'appels lors des différents exemples.** (pour utiliser une variable globale *var* dans une fonction, la charger par l'instruction "global *var*")

Quelle est la complexité de l'algorithme dans le pire des cas ? *Pour écrire votre réponse, sélectionner le mode Markdown dans le ménu déroulant lorsque vous cliquez sur une cellule.*

In [ ]:

In [ ]:

On cherche à présent à retourner une solution *optimale*, c'est-à-dire en utilisant le moins de pièces possible. Pour cela, on utilise d'abord un algorithme dit *glouton* qui choisit sytématiquement la plus grande pièce et vérfie s'il existe une solution. Si c'est le cas, *on ne teste pas les autres solutions.*

Implanter l'algorithme glouton

In [2]:
```
"""
Retourne la solution obtenue par l'algorithme glouton ou 'None' s'il n'y a pas de solution
Input :
    - n, un entier positif (la valeur)
    - s, une liste d'entiers (les pièces)
OUTPUT :
    - la liste des pièces effectivement utilisées.
"""
def rendreMonnaieGlouton(n, s):
    pass # remplacer par l'implantation
```

In [3]:
```
s = [5,2,2,2,1]
assert(rendreMonnaieGlouton(13,s) is None)
assert(rendreMonnaieGlouton(5,s) == [5])
assert(set(rendreMonnaieGlouton(6,s)) == {5,1})
assert(rendreMonnaieGlouton(4,s) == [2,2])
```

Malheureusement, l'algorithme glouton ne donne pas toujours la solution optimale. Observer l'exemple ci-dessous.

In [ ]:
```
s = [4,3,3,1,1]
rendreMonnaieGlouton(6,s)
```

Pour obtenir une solution optimale à tous les coups, on doit choisir une pièce p et tester la taille de la solution *avec* p et *sans* p et retourner la meilleure des deux.

Implanter l'algorithme rendreMonnaieOptimal ci-dessous.

In [1]:
```
"""
Retourne la solution optimale au rendu de monnaie ou 'None' s'il n'existe pas de solution
Input :
    - n, un entier positif (la valeur)
    - s, une liste d'entiers (les pièces)
OUTPUT :
    - la liste des pièces effectivement utilisées.
"""
def rendreMonnaieOptimale(n, s):
    pass # remplacer par l'implantation
```

In [3]:
```
s = [5,2,2,2,1]
assert(rendreMonnaieOptimale(13,s) is None)
```

```
assert(rendreMonnaieOptimale(5,s) == [5])
assert(len(rendreMonnaieOptimale(6,s)) == 2)
assert(len(rendreMonnaieOptimale(4,s)) == 2)
s = [4,3,3,1,1]
assert(len(rendreMonnaieOptimale(6,s)) == 2)
```

Quelques questions bonus :

- Traiter le cas où les pièces sont données comme un ensemble de valeurs et où l'on peut donc réutiliser plusieurs fois les mêmes valeurs.
- Peut-être avez vous remarqué que vos algorithmes font les mêmes calculs plusieurs fois, en les oubliant au fur et à mesure... Pas convaincu ? Faites un dessin de *l'arbre des appels*! Pour gérer ce problème il faut mémoriser les calculs intermédiaires, par exemple dans un tableau où l'entrée `[i][j]` contiendra `None` ou ce qui est retourné par `votre_fonction(i, j)`.
- Plus généralement, ce problème est un exemple typique d'une classe d'algorithmes que nous n'avons pas vu en cours et qu'on appelle *programmation dynamique*. **Informez-vous sur la programmation dynamique et cet algorithme en particulier et essayez de l'implanter**

In [ ]:

In [ ]:

In [ ]:

In [ ]: