Feuille 3: Résolution de problèmes par "backtracking"
CPP 2A, Semestre 2 – 2018-2019

On étudie une méthode de résolution de problèmes appelée "*backtracking*" (ou "*retour sur trace*" en français). Cette méthode est applicable dès qu'on peut construire incrémentalement une "*solution partielle*" au problème : lorsqu'on s'aperçoit que cette solution partielle ne peut pas être complétée, on défait les dernières étapes de construction pour reprendre la construction d'une autre solution partielle (c'est une démarche par "essais/erreurs"). Dans cette méthode, *chercher une solution* est juste une variante de **l'algorithme d'énumération de l'ensemble des solutions** : on arrête simplement l'énumération à la première solution ! La récursivité est ici un moyen commode d'explorer un *arbre* des solutions partielles.

Les deux premiers exos préparent le terrain en établissant un lien entre l'exploration des branches d'un arbre et les problèmes d'énumération. L'exo 3 est un exo d'introduction à la résolution de problème par backtracking. L'exo 4 donne un exemple typique d'application de cette technique.

```
t = (
    ((), 1, ()), 2, ((), 4, ())),
    6,
    ((), 8, ((), 9, ()))
)
```

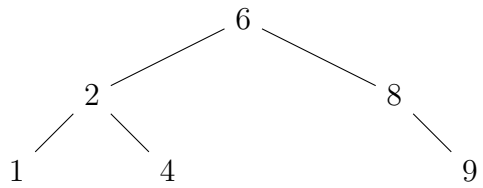

Figure 1: Un arbre binaire `t` en Python et sa représentation graphique (sans sous-arbre vide)

Exercice 1 La figure 1 rappelle l'exemple d'arbre binaire de la Feuille 2 avec sa représentation graphique. Dans un tel arbre, une *branche* est *par définition* une suite de descendants de la racine qui se termine sur une feuille (un noeud dont tous les fils sont vides). Par convention, on représente chaque branche comme la suite des éléments présents sur ses noeuds. Typiquement pour l'arbre `t` de la figure 1, ses 3 branches sous affichées ci-dessous :

```
[6, 2, 1]        [6, 2, 4]        [6, 8, 9]
```

Réciproquement, pour tout ensemble fini E de suites d'éléments, on peut construire un arbre T (pas forcément binaire) dont l'ensemble des branches est E. Chaque noeud de T correspond à un *préfixe* d'une suite de E. Dans le cas général, la racine de T sera un noeud

special correspondant au préfixe vide. En dehors de ce cas spécial (qui n'intervient pas ici), l'élément sur un noeud X est un préfixe de longueur 1 de l'ensemble des branches issues de ce noeud X. Ainsi, toute énumération se ramène peu ou prou à l'énumération des branches d'un certain arbre. D'où l'intérêt de l'exercice : *étudier comment énumérer les branches d'un arbre* (on se restreint au cas des arbres binaires pour simplifier).

Écrire une fonction récursive `enum_branch_rec(t, curr)` qui énumère l'ensemble des branches d'un arbre binaire non vide `t`, en préfixant l'affichage de ces branches par la liste `curr`. Intuitivement, le paramètre `curr` permet de mémoriser le préfixe de la branche associée à chaque fils de `t` lors des appels récursifs (on "étend" le préfixe courant `curr` en `curr+x` où `x` est l'élément à la racine de `t`).

Pour éviter les copies coûteuses de listes induites par l'opérateur `+` sur les listes lors des appels récursifs, on cherchera à modifier temporairement le paramètre `curr` en place. Typiquement, `curr.append(x)` *ajoute* `x` à la fin de la liste `curr`. Inversement, si `curr` est non vide, alors `curr.pop()` *supprime* l'élément à la fin de `curr`.

Exercice 2 *(examen avril 2015).* On considère le code ci-dessous

```
def enum_rec(alpha, curr):
    if len(curr) < len(alpha):
        for x in alpha:
            curr.append(x)
            enum_rec(alpha, curr)
            curr.pop()
    else:
        display(curr)
```

```
def display(arg):
    for x in arg:
        print(x, end="")
    print()


def enum(alpha):
    enum_rec(alpha, [])
```

1. Qu'affiche `enum(('a', 'b', 'c'))` ?

2. Proposer une modification de ce programme, de manière à ce que chaque ligne affichée soit numérotée, comme illustré ci-contre. On pourra utiliser une variable globale (c'est-à-dire dont la valeur persiste en mémoire entre les appels de fonctions), déclarée via le mot-clé `global`.

```
1 aaa
2 aab
3 ...
...
```

3. Proposer une fonction `enum_perms` paramétrée par une liste triée de lettres par ordre strictement croissant, et qui affiche l'ensemble des mots, par ordre alphabétique, qui sont permutations de cette liste de lettres. Typiquement `enum_perms(['a', 'b', 'c'])` doit afficher les lignes ci-contre.
   On pourra s'inspirer de `enum` en modifiant le paramètre `alpha` via des opérations 1) `alpha.pop(i)` qui supprime l'élément d'indice `i` et le retourne, à condition que `0 <= i < len(alpha)` ; 2) `alpha.insert(i, x)` qui insère `x` en position `i`, à condition que `0 <= i < len(alpha)` : les éléments d'indice supérieur ou égal à `i` sont décalés à l'indice suivant.

```
1 abc
2 acb
3 bac
4 bca
5 cab
6 cba
```

Exercice 3 On s'intéresse au problème du "*compte-et-bon simplifié*".

Définition 1 *Étant donné une liste* `l` *d'entiers* strictement positifs *et* `n` *un entier positif ou nul, on appelle* solution *au problème du "*compte-et-bon simplifié*" sur* `(l, n)` *un ensemble d'indices de* `l` *tel que la somme des éléments de* `l` *sur ces indices fasse* `n`.

Par exemple, si `l=[1, 9, 2, 3]` et `n=7`, il n'y a pas de solution.

Pour `l=[5, 1, 1, 3, 9, 2, 3]` et `n=7`, il y a exactement six solutions données chacune par une ligne ci-contre (où les indices commencent à 0 et les valeurs associées à ces indices apparaissent entre parenthèses).

```
0(5) 1(1) 2(1)
0(5) 5(2)
1(1) 2(1) 3(3) 5(2)
1(1) 2(1) 5(2) 6(3)
1(1) 3(3) 6(3)
2(1) 3(3) 6(3)
```

Pour chacun des traitements ci-dessous, programmer une fonction en Python qui l'effectue récursivement, comme éventuellement un cas particulier d'un traitement plus général (avec des paramètres en plus). Pour chacun de ces traitements, on précisera donc bien l'appel initial à la fonction récursive en fonction des paramètres `l` et `n`. Ci-dessous, "*afficher une solution*" signifie qu'il faut appeler une procédure `display_sol(l, curr)` où `curr` est un tableau d'indices de `l` strictement croissant qui représente la solution.

Voilà la liste des traitements :

1. décider si le problème a une solution ;
2. calculer le nombre de solutions au problème du "*compte-et-bon simplifié*" ;
3. *afficher toutes les solutions* ;
4. *afficher une seule solution.*

Exercice 4 Étant donné une constante `RN` $\in \mathbb{N}$ et `N` $= \texttt{RN}^2$, on s'intéresse maintenant à la résolution de Sudoku `N` $\times$ `N`. L'état initial d'un puzzle de Sudoku est représenté par une matrice carrée d'ordre `N` comme celle de la figure 2 où les "`0`" représente un nombre absent. Pour programmer la résolution des Sudoku par backtracking, on manipule une grille `grid` qui représente une solution partielle correcte de Sudoku, au travers des 3 fonctions ci-dessous (à coût constant).

```
# Retourne la valeur de la grille en position (i,j) dans 0..N
def value(grid, i, j)

# Essaie d'inserer le nombre 1+k en position (i,j)
def insert(grid, i, j, k)

# Supprime le nombre en position (i,j) sur la grille
def remove(grid, i, j)
```

Précisons le fonctionnement de `insert` et `remove`.

- `insert` *requiert* que les entiers `i`, `j` et `k` soient dans `0..N-1`. Si il y a déjà un nombre (non-nul) en position `(i,j)` dans la grille ou si insérer `1+k` en position `(i,j)` dans la grille violerait les règles du Sudoku, alors `insert` *retourne* `False` *sans modifier* la grille. Sinon, `insert` *modifie* la grille en *insérant* `1+k` en position `(i,j)` et *retourne* `True`.

- `remove` *requiert* que les entiers `i` et `j` soient dans `0..N-1` et que la grille contienne un nombre non-nul en position `(i,j)`. Dans ce cas, elle *modifie* la grille en *effaçant* ce nombre (valeur mise-à-zéro).

On dispose en outre d'une fonction `mkgrid(puzzle)` qui retourne une nouvelle grille à partir d'un `puzzle` comme celui de la figure 2 et d'une fonction `display(grid)` pour afficher une grille. Dans le cadre de l'exercice ci-dessous, **on utilisera toutes ces fonctions de manipulation de la grille sans chercher à les programmer**.

Écrire une procédure capable d'afficher toutes les solutions d'un puzzle de Sudoku. Étudier les variantes suivantes de la procédure précédente : numéroter les solutions ; s'arrêter sur la deuxième solution (pour vérifier que le puzzle a une et une seule solution).

Puzzle

		2	
			4
1		**4**	
	3		

```
puzzle = [
    [0, 0, 2, 0],
    [0, 0, 0, 4],
    [1, 0, 4, 0],
    [0, 3, 0, 0]
]
```

Solution

3	4	**2**	1
2	1	3	**4**
1	2	**4**	3
4	**3**	1	2

Figure 2: Un puzzle de Sudoku 4×4, sa représentation en Python, et son unique solution.