Récursivité

Exemple n° 1

On dispose d'une poupée russe (appelons la `pp`) et on cherche à savoir combien il y a de poupées russes emboîtées les unes dans les autres. La méthode intuitive est décrite en première approche par cet algorithme :

```
compter_poupees (pp) :
    Ouvrir pp
    fille = poupée qui se trouve dans pp
    nb_poupees = 1 + compter_poupees(fille)
    Renvoyer nb_poupees
```

Néanmoins cet algorithme souffre d'une erreur : lorsqu'on arrive à la dernière poupée, il n'y a pas de fille à l'intérieur. Il faut donc distinguer ce cas spécifique (on parle du *cas de base*) pour lequel il suffit *basiquement* de renvoyer 1. Finalement :

```
compter_poupees (pp) :
    Ouvrir pp
    Si pp est vide :
        Renvoyer 1
    Si pp est pleine :
        fille = poupée qui se trouve dans pp
        nb_poupees = 1 + compter_poupees(fille)
        Renvoyer nb_poupees
```

Cet algorithme comporte une spécificité assez naturelle mais que nous n'avons jamais rencontrée jusqu'ici : l'algorithme s'appelle lui-même. Cela pourrait en théorie ne jamais s'arrêter (l'algorithme s'appelle lui-même qui s'appelle lui-même qui s'appelle lui-même qui …).
En fait on est ici convaincu que l'algorithme fonctionne car :
- il sait gérer le *cas de base* (lorsque la poupée que l'on cherche à décompter est vide),
- sinon il s'appelle lui-même sur une version plus petite du problème initial et on est certain d'aboutir au *cas de base*.

Cette façon naturelle de penser certains algorithmes qui s'y prêtent correspond à la notion de *récursivité*. Un grand nombre de langages de programmation – dont Python – savent gérer la récursivité. Voici l'analogue de l'algorithme ci-dessus pour des tableaux (de type `list`) qui seraient emboîtés les uns dans les autres, par exemple : `[ [ [[]] ] ]`.

```
def compter_poupees( pp ):
    if len(pp) == 0:
        return 1
    else:
        fille = pp[0]
        return 1 + compter_poupees(fille)
```

On aurait aussi pu écrire un algorithme *itératif* à l'aide d'une boucle while (voir ci-contre). Néanmoins cette version itérative est moins naturelle que la version récursive. Par ailleurs, pour certains problèmes "naturellement récursifs", une version itérative peut s'avérer compliquée alors que la version récursive restera très simple et élégante à coder.

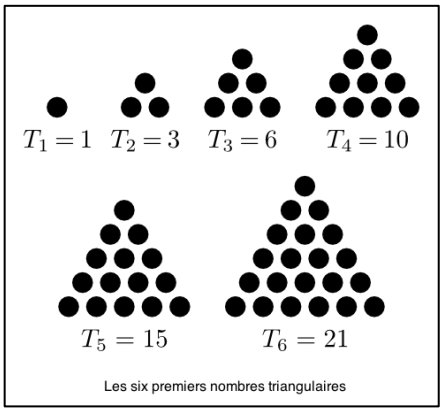
```
def compter_poupees_iteratif( pp ):
    nb_poupees = 1
    while len(pp) != 0:
        nb_poupees = nb_poupees + 1
        pp = pp[0]
    return nb_poupees
```

Exercice 1 :
On suppose désormais que chaque tableau peut soit être vide, soit contenir deux autres tableaux.
1) Modifier l'algorithme récursif `compter_poupees` pour qu'il s'adapte à ce cas là.
2) Modifier l'algorithme itératif `compter_poupees_iteratif` pour qu'il s'adapte à ce cas là.

La programmation récursive d'un algorithme consiste à utiliser une fonction qui effectue un appel à elle-même avec un argument différent. Pour que cela fonctionne il faut :
- que cette fonction renvoie une valeur particulière pour chaque cas de base (souvent un seul cas de base),
- que les arguments des appels successifs se "rapprochent" au fur et à mesure des cas de base de sorte qu'on soit certain de les atteindre.

Certains problèmes ne peuvent se résoudre simplement qu'avec des algorithmes récursifs alors que pour d'autres problèmes une version itérative est également possible.

Exemple n°2



source : www.bibmath.net

On souhaite calculer les nombres triangulaires. Appelons `triang(n)` la valeur renvoyée par l'algorithme calculant le $n^{ième}$ nombre triangulaire. Un rapide coup d'œil à la figure ci-contre nous montre que, par exemple, pour passer de `triang(5)` à `triang(6)`, il suffit de rajouter `6`. Autrement dit :

```
triang(n) = n + triang(n-1)
```

On a ici une formule *récursive* puisque la fonction / l'algorithme `triangulaire` s'appelle lui-même (ceux qui ont fait spécialité Maths en Première auront vu que *récursive* et *récurrente* ont la même racine).
Par ailleurs le cas de base va être le cas `triangulaire(1) = 1`.

On obtient donc :

```
def triang(n) :
    if n == 1:
        return 1
    else:
        return n + triang(n-1)
```

Voyons maintenant ce qui se passe lorsque Python exécute cet algorithme pour `n=4` :

Pile d'appels en cours	Représentation de l'arbre des appels
triang(4)	triang(4)
triang(4) triang(3)	triang(4) : return 4 + triang(3)

triang(4)
triang(3)
triang(2)
	triang(4) : return 4 + triang(3)
	      |
	     return 3 + triang(2)

triang(4)
triang(3)
triang(2)
triang(1)
	triang(4) : return 4 + triang(3)
	      |
	     return 3 + triang(2)
	       |
	      return 2 + triang(1)

triang(4)
triang(3)
triang(2)
triang(1)
	triang(4) : return 4 + triang(3)
	      |
	     return 3 + triang(2)
	       |
	      return 2 + triang(1)
	        |
	        return 1

triang(4)
triang(3)
triang(2)
	triang(4) : return 4 + triang(3)
	      |
	     return 3 + triang(2)
	       |
	      return 2 + 1

triang(4)
triang(3)
	triang(4) : return 4 + triang(3)
	      |
	     return 3 + 3

triang(4)	triang(4) : return 4 + 6
-	10

Il y a ici deux points importants à comprendre concernant la récursivité :
- lors de l'exécution, il faut que Python garde en mémoire les différents appels successifs qui ont été effectués (essentiellement, Python a en mémoire l'arbre des appels),
- il y a des phases de "descente" vers le cas de base lorsque la pile d'appels devient plus longue et des phases de "remontée" lorsque certains appels ont (enfin) renvoyé leur valeur et que la pile d'appels se réduit.

Exercice 2 :
Ce cas se transpose facilement en itératif. Donner une version itérative de cet algorithme en remarquant que cela revient à effectuer la somme des `n` premiers entiers.

Exemple n°3

Pour mettre à son service une valeureuse scientifique, un roi lui propose le marché suivant :

"Chaque jour, je te paierai ce que je t'avais payé la veille plus ce que je t'avais payé l'avant-veille"

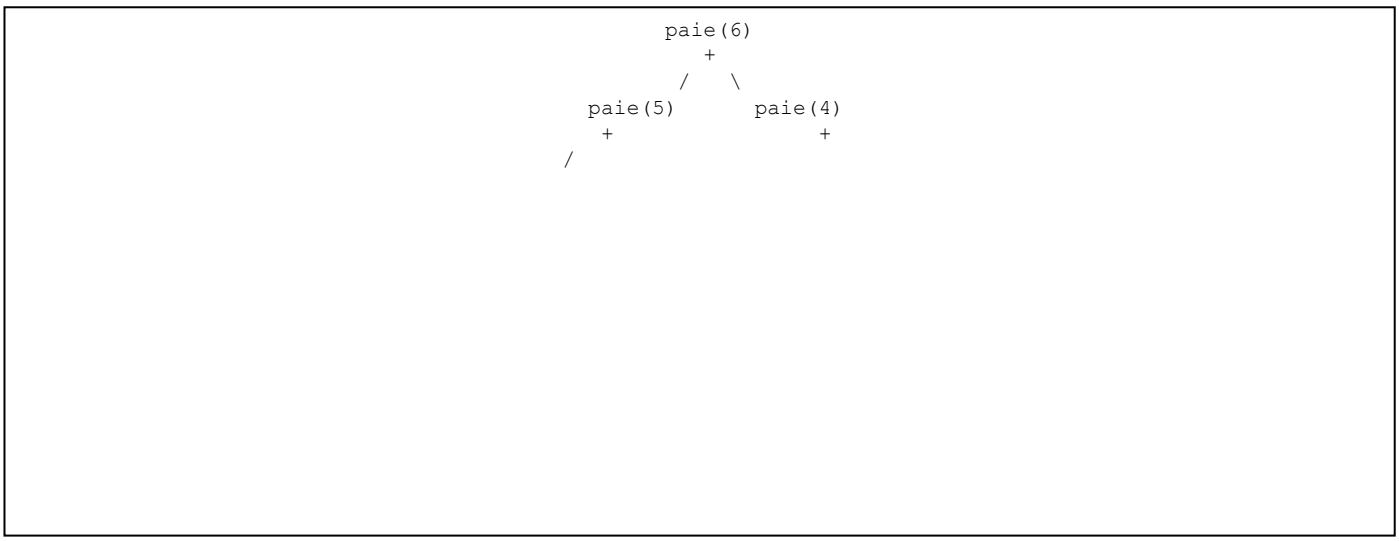
Bien entendu la scientifique lui demande combien elle sera payée au début, ce à quoi le roi lui répond qu'il la paiera 1 centime le premier jour (`n=1`).
La scientifique lui dit que cela ne lui suffit pas à savoir combien elle sera payée, le roi lui répond alors qu'il la paiera également 1 centime le second jour (`n=2`).

On note `paie(n)` la valeur de la paie du jour `n` (`paie` est donc la fonction ou l'algorithme qui calcule cette valeur).
1) Quelle est la *formule récursive* vérifiée par l'algorithme ?

2) Compléter le cadre suivant pour obtenir une version récursive de l'algorithme `paie` :

```
def paie(n) :
```

3) Compléter l'arbre des appels ci-dessous.



4) Quel est ici l'avantage de la programmation récursive ? Quel est l'inconvénient ?

Exercice 3 : Ce cas se transpose en itératif. Donner une version itérative de cet algorithme (on pourra utiliser une variable `veille`)

Nous sommes ici dans un cas pour lequel l'arbre des appels est *vraiment arborescent* : la version récursive de l'algorithme est très lente lorsque le nombre de jours `n` augmente. Puisqu'il existe une version itérative (ce qui n'est souvent pas le cas, confère exemple n°1 : exercice 1), il serait sur cet exemple préférable d'utiliser la version itérative.

L'arbre des appels de l'exemple n°2 était quant à lui *linéaire*,* dans un tel cas la version récursive est donc aussi performante que la version itérative (en première approche).
Exemple n°2 en récursif, `n = 30` : 30 appels.
Exemple n°3 en récursif, `n = 30` : 1 664 079 appels !

* : on peut tracer une ligne "qui ne se divise jamais" qui va du premier appel jusqu'au dernier appel.

Exercices

Exercice 4 :

On rappelle que la méthode `append` permet d'ajouter des éléments à un tableau (de type `list`).

```
>>> tab = [1, 3, 5]
>>> tab.append(7)
>>> tab.append(9)
>>> tab
[1, 3, 5, 7, 9]
```

1) La méthode `append` respecte-t-elle le paradigme de la programmation fonctionnelle ? Justifier brièvement.

2) On considère les trois algorithmes ci-dessous. Pour chacun d'entre eux :
   a) Dessiner l'évolution de la pile des appels et du remplissage du tableau lorsqu'on exécute les deux instructions :
   ```
   mon_tab = []
   algo_x(4, mon_tab)
   ```

   b) Indiquer quelle est alors la valeur de `mon_tab` à la fin de l'exécution des deux instructions.

Algorithme n°1	Algorithme n°2	Algorithme n°3

`def algo_1(n, tab):`
`  if n == 0:`
`  pass`
`  else:`
`  tab.append(n)`
`  tab.append(n)`
`  algo_1(n-1, tab)`
	`def algo_2(n, tab):`
	`  if n == 0:`
	`  pass`
	`  else:`
	`  tab.append(n)`
	`  algo_1(n-1, tab)`
	`  tab.append(n)`
		`def algo_3(n, tab):`
		`  if n == 0:`
		`  pass`
		`  else:`
		`  algo_1(n-1, tab)`
		`  tab.append(n)`
		`  tab.append(n)`

3) (***) Proposer une modification de l'algorithme n°2 pour qu'il respecte le paradigme fonctionnel. On utilisera alors l'algorithme avec les deux instructions suivantes :
   ```
   tab_depart = []
   tab_resultat = algo_x(4, tab_depart)
   ```

Exercice 5 :

La factorielle de l'entier 7 est : $7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$ et se note 7 ! De même on définit la factorielle de n'importe quel entier strictement positif.
Pour la factorielle de zéro on a la convention suivante : $0! = 1$.

On note `factorielle` une fonction / un algorithme permettant de calculer la factorielle de tout nombre entier positif ou nul.

1) Ecrire une relation de récursivité vérifiée par `factorielle`.
2) Implémenter cet algorithme en version récursive.
3) Peut-on facilement en obtenir une version itérative ?

Exercice 6 :

1) Ecrire une fonction récursive `compter_777(tab, k)` permettant de compter le nombre d'éléments du tableau `tab` ayant un indice inférieur ou égal à `k` et qui sont égaux à 777.

2) En déduire une fonction récursive permettant de compter le nombre d'éléments égaux à 777 dans un tableau `tab`.

Exercice 7 (**) :

On dispose de `n` inégalités disposées dans un tableau `inegs`.
On dispose de `n+1` nombres entiers triés dans un tableau `nbs`.
Ecrire un algorithme (en Python ou en langage naturel) qui permet d'obtenir un tableau `nbs_ok` contenant tous les nombres de `nbs` dans un ordre tel qu'ils respectent les inégalités de `inegs`. Voici un exemple :

```
Inegs = [ '<', '<', '>', '>', '<']
nbs = [ 3, 13, 13, 17, 19, 23]
```
`nbs_ok = [13, 17, 19, 13, 3, 23]` convient puisqu'on a bien : `13 < 17 < 19 > 13 > 3 < 19`.

Exercice 8 (***):

Lors d'un championnat, chacune des n=26 équipes `A, B, C, D … Z` a rencontré chacune des n-1 = 25 autres équipes et a soit perdu soit gagné la rencontre. Déterminer un algorithme permettant de ranger les équipes dans un tableau `equipes` de taille n=26 de sorte que pour tout indice `i` entre 0 et 24, on ait `equipes[i]` qui a gagné le match contre `equipes[i+1]`.
Exemple : A a gagné contre B, A a perdu contre C, A a perdu contre D, B a perdu contre C, B a perdu contre D, C a gagné contre D, `equipes = [C, D, A, B]` convient