

6.1 Définitions

Définition – Fonctions récursives

Une fonction récursive est une fonction qui s'appelle elle-même.
On appelle récursion l'appel de la fonction à elle-même.

La programmation récursive est un paradigme de programmation au même titre que la programmation itérative. Un programme écrit de manière récursive peut être traduit de manière itérative, même si dans certains cas, cela peut s'avérer délicat.

Méthode –

- ▶ Une fonction récursive doit posséder une condition d'arrêt (ou cas de base).
- ▶ Une fonction récursive doit s'appeler elle-même (récursion).
- ▶ L'argument de l'étape de récursion doit évoluer de manière à se ramener à la condition d'arrêt.

Quelques avantages et inconvénients d'un algorithme récursif :

- ▶ simplicité de l'écriture récursive dans certains cas ;
- ▶ l'algorithme peut sembler plus aisé lors de sa lecture ;
- ▶ comme pour les algorithmes itératifs, il faut prêter attention à sa terminaison en utilisant un variant de boucle (à voir ultérieurement) ;
- ▶ comme pour les algorithmes itératifs, il est aussi nécessaire de vérifier la correction de l'algorithme en utilisant un invariant de boucle (à voir ultérieurement) ;
- ▶ les complexités algorithmiques temporelle et spatiale (à voir ultérieurement) d'un algorithme récursif peuvent être plus coûteuses que celles d'un algorithme itératif suivant l'algorithme retenu.

6.2 Suites définies par récurrence

Les suite définies par récurrence pour lesquelles $u_n = f(u_{n-1}, u_{n-2}, \dots)$ sont des cas d'application directs des fonctions récursives.

Par exemple, soit la suite u_n définie par récurrence pour tout $n \in \mathbb{N}^*$ par
$$\begin{cases} u_1 = 1 \\ u_{n+1} = \frac{u_n + 6}{u_n + 2} \end{cases} .$$

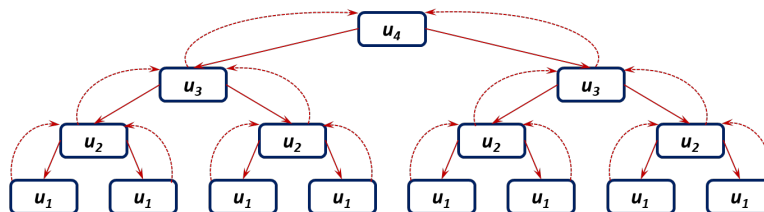
Il est possible de calculer le n^{e} terme par un algorithme itératif ou un algorithme récursif.

```
1 def un_it (n : int) -> float :
2     if n == 1 :
3         return 1
4     else :
5         u = 1
6         for i in range(2,n+1):
7             u = (u+6)/(u+2)
8         return u
```

```
1 def un_rec (n : int) -> float :
2     if n == 1 :
3         return 1
4     else :
5         return (un_rec(n-1)+6)/(un_rec(
        n-1)+2)
```

6.1 Définitions	1
6.2 Suites définies par récurrence	1
6.3 Slicing de tableau ou de chaînes de caractères	2
6.4 Algorithmes dichotomiques – Diviser pour régner	3
6.5 Tracer de figures définies par récursivité	4

La figure suivante montre que dans le cas de l'algorithme récursif proposé plusieurs termes sont calculés à plusieurs reprises ce qui constitue une perte de temps et d'espace mémoire.

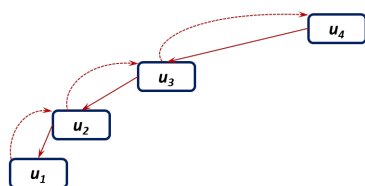


Une autre formulation de l'algorithme récursif permet très simplement de diminuer le nombre de termes calculés.

Remarque – Complexité

On peut estimer le coût temporel de la fonction `un_rec` ainsi : $C(n) = C(n-1) + C(n-1) + C_0$ car dans le `else` il y a deux appels de `un_rec` à l'itération $n-1$ et des opérations à temps constant (additions et division).

$C(n)$ est une suite arithmético géométrique de terme général $u_n = a^n (u_0 - r) + r$ avec $a = 2$. On a donc $C(n) = \mathcal{O}(2^n)$.



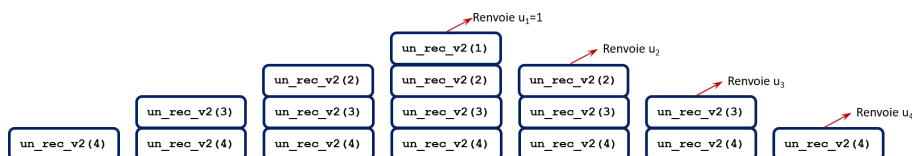
```
1 def un_rec_v2 (n : int) -> float :
2   if n == 1 :
3     return 1
4   else :
5     v = un_rec_v2(n-1)
6     return (v+6)/(v+2)
```

Remarque – Complexité

On peut estimer le coût temporel de la fonction `un_rec_v2` ainsi : $C(n) = C(n-1) + C_0$

$C(n)$ est une suite arithmético géométrique de terme général $u_n = a^n (u_0 - r) + r$ avec $a = 1$. On a donc $C(n) = \mathcal{O}(n)$.

Dans la même idée que les graphes présentés ci-dessus, il est possible de représenter la pile des appels récursifs, c'est à dire la succession des appels qui vont être faits pour calculer le n ème terme d'une suite.



6.3 Slicing de tableau ou de chaînes de caractères

Il est possible d'agir par récursivité sur des listes ou sur des chaînes de caractère. Pour cela, il peut être nécessaire d'utiliser le slicing. Pour rappel, `a = ch[i:j]` permet d'affecter à `a` la liste (ou le chaîne de caractères) constitué des éléments de `i` (inclus) à `j` (exclus). En écrivant `a = ch[i:]` on affecte à `a` tous les éléments du i ème au dernier.

Ainsi, pour réaliser la somme des éléments d'une liste `L` par récursivité :

- commençons par déterminer le cas terminal : si la taille de la liste vaut 1, on renvoie $L[0]$;
- sinon (si la taille vaut n), on peut choisir de renvoyer $L[0] + \text{somme}(L[1] + L[2] + \dots + L[n-1])$.

Cela se traduit ainsi.

```
1 def somme(L:list) -> float :
2     if len(L)==1 :
3         return L[0]
4     else :
5         return L[0]+somme(L[1:])
```

Pour une chaîne de caractères, si on souhaite renvoyer son miroir (par exemple, le miroir de abc serait cba) :

- si la chaîne de caractère a une taille de 1, on renvoie le caractère ;
- sinon, on renvoie la concaténation de $\text{miroir}(ch[1:]) + ch[0]$.

```
1 def miroir(ch:str) -> str :
2     if len(ch)==1 :
3         return ch[0]
4     else :
5         # Attention le + désigne la concaténation
6         return miroir(ch[1:])+ch[0]
7
```

6.4 Algorithmes dichotomiques – Diviser pour régner

Les algorithmes dichotomiques se prêtent aussi à des formulations récursives. Prenons comme exemple la recherche d'un élément dans une liste triée. L'algorithme de gauche propose une version itérative. L'algorithme de droite une version récursive.

```
1 def appartient_dicho(e : int , t : list) -> bool:
2     """Renvoie un booléen indiquant si e est dans t.
3     Préconditions : t est un tableau de nombres trié par ordre croissant e est
4     un nombre"""
5     # Limite gauche de la tranche où l'on recherche e
6     g = 0
7     # Limite droite de la tranche où l'on recherche e
8     d = len(t)-1
9     # La tranche où l'on cherche e n'est pas vide
10    while g <= d:
11        # Milieu de la tranche où l'on recherche e
12        m = (g+d)//2
13        pivot = t[m]
14        if e == pivot: # On a trouvé e
15            return True
16        elif e < pivot:
17            # On recherche e dans la partie gauche de la tranche
18            d = m-1
19        else:
20            # On recherche e dans la partie droite de la tranche
21            g = m+1
22    return False
```

```
1 def appartient_dicho_rec(e : int , t : list) -> bool:
2     # Limite gauche de la tranche où l'on recherche e
3     g = 0
4     # Limite droite de la tranche où l'on recherche e
```

```

5     d = len(t)-1
6     # La tranche où l'on cherche e n'est pas vide
7     # Milieu de la tranche où l'on recherche e
8     m = (g+d)//2
9     if t==[]: #e n'est pas dans t
10        return False
11    elif t[m]==e: #e est dans t
12        return True
13    elif t[m]<e: #On recherche à droite
14        return appartient_dicho_rec(e, t[m+1:])
15    elif t[m]>e: #On recherche à gauche
16        return appartient_dicho_rec(e, t[:m])

```

6.5 Tracer de figures définies par récursivité

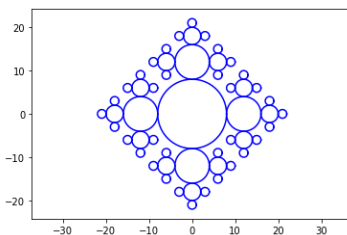
Un grand nombre de figures peuvent être tracées en utilisant des algorithmes récursifs (flocon de Koch, courbe de Peano, courbe du dragon *etc.*).

Ci-dessous un exemple de figure définie par récursivité où à chaque itération un cercle va se propager vers le haut, le bas, la gauche et la droite. À chaque itération, le rayon de cercle est divisé par 2.

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3 def cercle(x,y,r):
4     theta = np.linspace(0, 2*np.pi, 100) #des points régulièrement espacés
5     # dans l'intervalle [0,2pi]
6     X = [x+r*np.cos(t) for t in theta]      #abscisses de points du cercle C((
7     Y = [y+r*np.sin(t) for t in theta]      #ordonnées de points du cercle C
8     plt.plot(X,Y,"b")                      #tracé sans affichage

```



```

1 def bubble(n, x, y, r, d):
2     cercle(x, y, r)
3     if n > 1:
4         if d != 's':
5             bubble(n-1, x, y+3*r/2, r/2, "n")
6         if d != 'w':
7             bubble(n-1, x+3*r/2, y, r/2, "e")
8         if d != 'n':
9             bubble(n-1, x, y-3*r/2, r/2, "s")
10        if d != 'e':
11            bubble(n-1, x-3*r/2, y, r/2, "w")
12    bubble(4, 0, 0, 8, "")
13    plt.axis("equal")
14    plt.show()

```

Applications

Multiplication

Question 1 Ecrire une fonction itérative `mult(n:int, p:int)->int` permettant de multiplier deux entiers (sans utiliser `*`).

Question 2 Ecrire une fonction récursive `mult_rec(n:int, p:int)->int` permettant de multiplier deux entiers (sans utiliser `*`).

Exponentiation

Question 1 Ecrire une fonction itérative `expo(x:int, n:int)->int` permettant de calculer x^n (sans utiliser `**`).

Question 2 Ecrire une fonction récursive `expo_rec(x:int, n:int)->int` permettant de calculer x^n (sans utiliser `**`).

Une autre méthode, celle de l'exponentiation rapide consiste à remarquer que $x^n =$

$$\begin{cases} (x^2)^{n/2} & \text{si } n \text{ pair} \\ x \times (x^2)^{(n-1)/2} & \text{si } n \text{ impair} \end{cases}$$

Question 3 Ecrire une fonction itérative `expo_rap_it(x:int, n:int)->int` permettant de calculer x^n (sans utiliser `**`).

Question 4 Ecrire une fonction récursive `expo_rap_rec(x:int, n:int)->int` permettant de calculer x^n (sans utiliser `**`).

Somme récursive

Question 1 Écrire une fonction récursive `somme(L)`, de paramètre une liste `L` de nombres, qui calcule et renvoie la somme des éléments de `L`. On précise que l'exécution de `somme([])` doit renvoyer la valeur 0 et que la liste `L` n'est pas modifiée par la fonction (elle peut être modifiée momentanément mais, à la fin de l'exécution de la fonction, on doit retrouver la liste initiale).

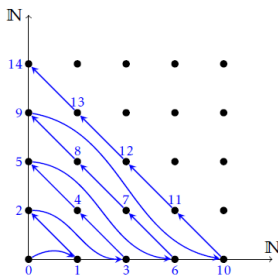
Maximum récursif

Question 1 Écrire une fonction récursive `maximum(L)`, de paramètre une liste `L` de nombres (supposée non vide), qui détermine et renvoie le maximum de `L`. On précise que la liste `L` n'est pas modifiée par la fonction et qu'il ne doit bien sûr pas être fait usage de la fonction native `max` pour les listes.

```

000000
00000
0000
000
00
0
0
00
000
0000
00000
000000

```



Dessins récursifs

Question 1 Écrire une fonction récursive `triangle_1(n)`, de paramètre un entier naturel n , qui ne renvoie rien mais qui produit à l'aide de la fonction `print` la figure ci-contre (dans le cas bien sûr où la fonction est évaluée en 6).

Question 2 Faire de même avec une fonction récursive `triangle_2(n)` qui produit ce même « triangle rectangle » mais pointe en haut (ci-contre la figure obtenue dans le cas où la fonction est évaluée en 6).

Dénombrément de couples

On démontre que sur l'ensemble $\mathbb{N} \times \mathbb{N}$ est dénombrable en numérotant chaque couple $(x, y) \in \mathbb{N}^2$ suivant le procédé suggéré par la figure ci-contre.

Question 1 Rédiger une fonction récursive qui retourne le numéro du point de coordonnées (x, y) .

Question 2 Rédiger la fonction réciproque, là encore de façon récursive.

Exercice

Question 1 Prouver la terminaison de la fonction G de Hofstadter, définie sur \mathbb{N} de la façon suivante :

```

1 def g(n) :
2     if n == 0 :
3         return 0
4     return n - g(g(n-1))

```

On suppose donné un tableau $t[0, \dots, n-1]$ (contenant au moins trois éléments) qui possède la propriété suivante : $t_0 \geq t_1$ et $t_{n-2} \leq t_{n-1}$. Soit $k \in [1, n-2]$; on dit que t_k est un minimum local lorsque $t_k \leq t_{k-1}$ et $t_k \leq t_{k+1}$.

Question 1 Justifier l'existence d'un minimum local dans t .

Question 2 Il est facile de déterminer un minimum local en coût linéaire : il suffit de procéder à un parcours de tableau. Pourriez-vous trouver un algorithme récursif qui en trouve un en réduisant le coût logarithmique ?

Tracé de cercles

On suppose disposer d'une fonction `circle([x, y], r)` qui trace à l'écran un cercle de centre $(x; y)$ de rayon r .

Question 1 Définir deux fonctions récursives permettant de tracer les dessins présentés figure suivante (chaque cercle est de rayon moitié moindre qu'à la génération précédente).

