

PILE
fact(0) = 1
fact(1) = 1*fact(0)
fact(2) = 2*fact(1)
fact(3) = 3*fact(2)

Chapitre 6

Révisions sur la récursivité

- Appel de `fact(3)` : Retourne `3*fact(2)`
- Appel de `fact(2)` : Retourne `2*fact(1)`
- Appel de `fact(1)` : Retourne `1*fact(0)`
- Appel de `fact(0)` : Retourne 1
- Les appels successifs sont stockés dans une pile d'exécution.

PILE
fact(0) = 1
fact(1) = 1*fact(0) = 1 * 1 = 1
fact(2) = 2*fact(1) = 3 * 2 = 6
fact(3) = 3*fact(2) = 3 * 2 = 6

6.1 Notion de récursivité

- Une fonction en python peut appeler toute fonction définie.
- En particulier une fonction (en python ou tout autre langage de programmation) peut s'appeler elle-même. **On parle alors de fonction récursive** : une fonction est récursive lorsqu'elle s'appelle elle-même.
- Notamment lorsque le calcul de `fonction(n)` appelle le calcul de `fonction(n-1)`, etc... jusqu'à résoudre celui de `fonction(0)`.
- Un exemple classique est le calcul de factorielle $n : n!$.

```
def fact(n):
    if (n == 0): # Initialisation
        return 1 # 0! = 1
    else:
        return n * fact(n-1) # Relation de recurrence
```

- Par exemple l'appel de `fact(3)` appelle `fact(2)` qui appelle `fact(1)` qui finalement appelle `fact(0)` qui retourne 1.
- Ainsi `fact(1)` retourne `1*fact(0)=1`, `fact(2)` retourne `2*fact(1)=2`, et finalement `fact(3)` retourne `3*fact(2)=6`.

- Appel de `fact(0)` : Retourne 1
- Appel de `fact(1)` : Retourne `1*fact(0) = 1`
- Appel de `fact(2)` : Retourne `2*fact(1) = 2`
- Appel de `fact(3)` : Retourne `3*fact(2) = 6`. C'est le résultat retourné.
- La complexité (ici!) est linéaire en temps et en espace.
- Avantages : élégant, concis, se prête bien à la récurrence, permet de résoudre facilement des problèmes compliqués.
- Inconvénients : complexité en espace (du fait de la pile d'exécution). En python le nombre d'appels récursifs est limité à 1000 : pile de capacité limitée.
- Autre désavantage : la récursivité peut facilement donner lieu à des boucles infinies. Par exemple dans le programme précédent l'appel de `fact(-1)` appelle `fact(-2)`, qui appelle `fact(-3)`, etc..., donnant lieu à une boucle infinie, et dans la pratique à une erreur de dépassement de capacité de la pile d'exécution.
- Amélioration du programme pour éviter ce phénomène : ou encore,

6.2 Récursivité : Pile d'exécution

- Les appels successifs sont stockés dans une pile d'exécution.

```
def fact(n):
    assert isinstance(n,int) and n >= 0
    if (n == 0): # Initialisation
        return 1 # 0! = 1
    else:
        return n * fact(n-1) # Relation de recurrence
```

- Terminaison et correction du programme :

- si l'argument n n'est pas un entier naturel : terminaison avec erreur.
- si n est un entier naturel : terminaison et correction de `fact(n)` se démontrent par un récurrence immédiate sur n : vrai si $n = 0$, et si `fact(n-1)` se termine et retourne $(n-1)!$, alors `fact(n)` se termine en retournant $n \cdot (n-1)! = n!$.

6.3 Exemples

6.3.1 L'algorithme d'exponentiation rapide

- Regardons un autre exemple, lui très utile en pratique :
- **Calcul d'une puissance x^n par multiplications successives**

Supposons que l'on s'interdise l'emploi de l'opération de mise en puissance (comme c'est le cas pour un microprocesseur), et que l'on veuille écrire une fonction prenant en paramètre un nombre x et un entier n et qui retourne la puissance x^n .

- Solution naïve non récursive, à l'aide d'une boucle `for` :

```
def puissance(x,n):
    result = 1
    for i in range(n):
        result = result * x
    return result
```

Complexité : linéaire (en $O(n)$) en temps dans le pire et le meilleur des cas. Bornée (en $O(1)$) en espace.

- Regardons un autre exemple, lui très utile en pratique :
- **Calcul d'une puissance x^n par multiplications successives**

Supposons que l'on s'interdise l'emploi de l'opération de mise en puissance (comme c'est le cas pour un microprocesseur), et que l'on veuille écrire une fonction prenant en paramètre un nombre x et un entier n et qui retourne la puissance x^n .

- Solution naïve, récursive :

```
def puissance(x,n):
    if n==0:
        return 1
    else:
        return x * puissance(x,n-1)
```

Complexité : linéaire (en $O(n)$) en temps dans le pire et le meilleur des cas. Linéaire (en $O(n)$) en espace.

- On peut aller beaucoup plus vite, en mettant à profit les propriétés de la mise en puissance :

$$x^0 = 1 \quad ; \quad x^{2p} = (x^2)^p \quad ; \quad x^{2p+1} = x \cdot x^{2p}$$

- Ainsi :

$$puissance(x,n) = \begin{cases} 1 & \text{si } n=0 \\ puissance(x^2, n/2) & \text{si } n \text{ est pair} \\ x \times puissance(x^2, (n-1)/2) & \text{si } n \text{ est impair} \end{cases}$$

- Code python :

```
def puissance(x,n):
    if n == 0:
        return 1
    elif n%2==0:
        return puissance(x**2, n//2)
    else:
        return x * puissance(x**2, n//2) #ici n//2=(n-1)/2
```

Vérifions sur un exemple que l'algorithme est beaucoup plus rapide :

Exemple calcul de x^{100} :

- 100 est pair. On obtient x^{100} en élevant x au carré, $x_1 = x^2$ (1ère mult.); on devra élever x_1 à l'exposant 50.
- 50 est pair. On obtient $(x_1)^{50}$ en élevant x_1 au carré, $x_2 = (x_1)^2$ (2ème mult.); on devra élever x_2 à l'exposant 25.
- 25 est impair. On obtient $(x_2)^{25}$ en élevant x_2 au carré, $x_3 = (x_2)^2$ (3ème mult.); on devra élever x_3 à l'exposant 12 puis multiplier par x_2 (4ème mult.).
- 12 est pair. On obtient $(x_3)^{12}$ en élevant x_3 au carré, $x_4 = (x_3)^2$ (5ème mult.); on devra élever x_4 à l'exposant 6.
- 6 est pair. On obtient $(x_4)^6$ en élevant x_4 au carré, $x_5 = (x_4)^2$ (6ème mult.); on devra élever x_5 à l'exposant 3.
- 3 est impair. On obtient $(x_5)^3$ en élevant x_5 au carré, (7ème mult.) puis en multipliant par x_5 (8ème mult.).

Au total : 8 multiplications, contre 100 pour l'algorithme naïf!

- Une variante de l'algorithme d'exponentiation rapide, utilisant plutôt :

$$x^0 = 1 \quad ; \quad x^{2p} = (x^p)^2 \quad ; \quad x^{2p+1} = x \cdot x^{2p}$$

- Ainsi :

$$puissance(x,n) = \begin{cases} 1 & \text{si } n=0 \\ (puissance(x, n/2))^2 & \text{si } n \text{ est pair} \\ x \times (puissance(x, (n-1)/2))^2 & \text{si } n \text{ est impair} \end{cases}$$

• Code python :

```
def puissance(x,n):
    if n == 0:
        return 1
    elif n%2==0:
        return puissance(x,n//2)**2
    else:
        return x * puissance(x,n//2)**2 #ici n//2=(n-1)/2
```

Exemple calcul de x^{100} :

- 100 est pair. On obtient x^{100} en élevant au carré x^{50} , soit une multiplication.
- 50 est pair. On obtient x^{50} en élevant au carré x^{25} , soit une multiplication.
- 25 est impair. On obtient x^{25} en élevant au carré x^{12} puis en multipliant par x , soit 2 multiplications.
- 12 est pair. On obtient x^{12} en élevant au carré x^6 , soit une multiplication.
- 6 est pair. On obtient x^6 en élevant au carré x^3 , soit une multiplication.
- 3 est impair. On obtient x^3 en élevant x au carré puis en multipliant par x .

Au total : 8 multiplications, contre 100 pour l'algorithme naïf!

• Complexité de l'exponentiation rapide

• Démontrons que l'algorithme d'exponentiation rapide a une complexité logarithmique (d'ordre $\Theta(\log(n))$).

I. Soit $n \in \mathbb{N}$ et $p = \lfloor \log_2(n) \rfloor$. Montrons que :

$$2^p \leq n < 2^{p+1}$$

↪ Soit $n \in \mathbb{N}$ et $p = \lfloor \log_2(n) \rfloor$.

Puisque : $2^{\lfloor \log_2(n) \rfloor} = n$, $\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1$, et $x \mapsto 2^x$ est strictement croissante, alors : $2^p \leq n < 2^{p+1}$.

II. Montrons par récurrence sur p que lorsque $2^p \leq n < 2^{p+1}$, alors il faut entre p et $2(p+1)$ multiplications pour le calcul de x^n par l'algorithme d'exponentiation rapide.

↪ Récurrence sur p :

Initialisation. Si $p = 0$: $1 \leq n < 2$, et le calcul de $x^n = x$ par l'algorithme d'exponentiation rapide nécessite une multiplication. Vrai au rang 0.

Hérédité. Supposons la propriété vraie au rang p et supposons que $2^{p+1} \leq n < 2^{p+2}$. L'algorithme obtient x^n en : élevant x au carré x^2 (1 multiplication), puis en élevant x^2 à la puissance $\lfloor n/2 \rfloor$, puis éventuellement en multipliant par x (≤ 1 multiplication).

Or $2^p \leq \lfloor n/2 \rfloor < 2^{p+1}$. Par hypothèse de récurrence l'élevation à la puissance $\lfloor n/2 \rfloor$ par exponentiation rapide nécessite entre p et $2(p+1)$ multiplications, donc celui de x^n nécessite entre $p+1$ et $2(p+1)+2 = 2(p+2)$ multiplications. cqfd.

III. En déduire que l'algorithme d'exponentiation rapide a pour complexité $\Theta(\log(n))$.

↪ Le nombre d'opérations effectuées par l'algorithme a même ordre que le nombre de multiplications effectuées (au plus 2 tests par multiplication).

D'après la question précédente, le calcul de x^n nécessite :

au moins : $p = \lfloor \log_2(n) \rfloor \geq \log_2(n) - 1$ multiplications,

au plus : $2(p+1) = 2(\lfloor \log_2(n) \rfloor + 1) \leq 2\log_2(n) + 2$ multiplications.

Donc la complexité est en $\Theta(\log(n))$. ■

Exemple : l'exponentiation rapide de x^{2^p} :

$$x^{2^p} = \underbrace{(\dots((x^2)^2)\dots)^2}_{p \text{ élévations au carré}}$$

nécessite p multiplications (une pour chaque élévation au carré).

• Version itérative de l'exponentiation rapide

Une version itérative de l'algorithme d'exponentiation rapide peut s'obtenir ainsi :

1. Le résultat sera contenu dans une variable `puissance` qui initialement vaut 1.
2. Si n est impair on multiplie `puissance` par x , sinon on ne fait rien.
3. puis on change x par x^2 et n par $n//2$.
4. On recommence tant que $n > 0$.

• **Version itérative de l'exponentiation rapide par divisions euclidiennes de n par 2 successives** :

```
def expRapide(x,n):
    puissance = 1
    while n > 0:
        if n%2 == 1:
            puissance = puissance * x
        x = x**2
        n = n//2
    return puissance
```

Terminaison : n est un variant de boucle (ses valeurs prises forment une suite strictement décroissante d'entiers naturels).

Complexité : le nombre d'opérations élémentaires est du même ordre que le nombre de passage dans la boucle ; c'est à dire $\log_2(n)$: complexité logarithmique $\Theta(\log(n))$.

Correction : Après k passages dans la boucle, notons n_k la valeur de n , p_k la valeur de puissance et x_k la valeur de x . On a pour invariant de boucle :

$$p_k = x_0^{r_k}$$

où n_k, r_k sont le quotient et le reste dans la division euclidienne de $n_0 = n$ par 2^k , et $x_k = x_0^{2^k}$.

Par récurrence :

Initialisation. ($k = 1$). Après un passage dans la boucle p_1 vaut 1 si n_0 est pair et x_1 sinon ; n_1 vaut le quotient de n_0 par 2 ; $x_1 = x_0^2$.

Hérédité. Après k passage, n_k est le quotient de $n_0 = n$ par 2^k , r_k est le reste.

Si n_k est pair, $p_{k+1} = p_k$ et $r_{k+1} = r_k$ puisque :

$$n = n_k \cdot 2^k + r_k = 2n_{k+1} \cdot 2^k + r_k = n_{k+1} \cdot 2^{k+1} + r_k.$$

Ainsi $p_{k+1} = p_k = x_0^{r_k} = x_0^{r_{k+1}}$.

Si n_k est impair, $p_{k+1} = p_k \cdot x_k = x_0^{r_k} \cdot x_0^{2^k}$.

Or $r_{k+1} = r_k + 2^k$ puisque :

$$n_0 = 2^k \times n_k + r_k = 2^k \times (2n_{k+1} + 1) + r_k = 2^{k+1}n_{k+1} + 2^k + r_k$$

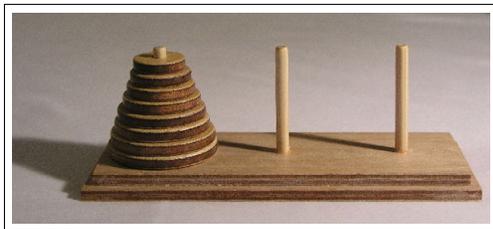
Ainsi $p_{k+1} = x_0^{r_{k+1}} = x_0^{r_{k+1}}$.

CQFD.

6.3.2 Le problème des tours d'Hanoï

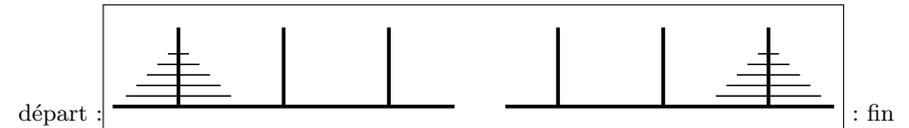
• Enoncé du problème

Le problème des tours de Hanoï est un jeu de réflexion inventé par le Mathématicien français Edouard Lucas en 1883 :



Des disque percés de diamètre décroissants sont empilés sur une colonne. Il y a trois colonnes. Il s'agit de déplacer les disques de la colonne de gauche à la colonne de droite –en un minimum de coût– et en respectant les règles suivantes :

1. On ne peut déplacer qu'un disque à la fois, et sur la colonne de son choix.
2. On ne peut empiler un disque que sur une colonne vide ou sur un disque plus grand.



• Résolution récursive

Ecrivons un programme qui résout le problèmes des tours de Hanoï pour n disques. Pour le résoudre simplement et élégamment nous allons procéder par récursivité, en remarquant :

• Si l'on sait résoudre le problème pour $n - 1$ disques alors il n'est pas difficile de le résoudre pour n disques :

1. Procéder aux mouvements permettant de résoudre avec les $n - 1$ disques de plus petits diamètres pour que la position d'arrivée soit la colonne du milieu.
2. Déplacer le disque de plus gros diamètre sur la colonne de droite.
3. Procéder aux mouvements de résolution avec les $n - 1$ disques de la colonne du milieu pour les déplacer sur la colonne de droite.

• On constitue une fonction :

`hanoi(n, Colonne_départ, Colonne_intermédiaire, Colonne_arrivée)`

• Schéma récursif de résolution (en pseudo-code) :

```
def hanoi(n, D, I, A):
    if n != 0:
        hanoi(n-1, D, A, I)
        déplacer le disque restant de D à A
        hanoi(n-1, I, D, A)
```

• Pour l'implémenter : on utilisera 3 piles pour D, L, A, de capacités illimités (c'est à dire 3 listes en python),

1. l'empilement s'obtient alors par `Pile.append(e)`,
2. le dépilement par `Pile.pop()`
3. le peek par `Pile[-1]`, la taille par `len(Pile)`, et "est vide?" par `len(Pile) == 0` (que l'on n'utilisera pas ici).

• Initialement :

```
D = [ n-k for k in range(n) ] # D = [ n, n-1, ..., 2, 1 ]
I = [ ]
A = [ ]
```

• Code

```
def hanoi(n, D, I, A): # Partie recursive
    if n!=0:
```

```

    hanoi(n-1,D,A,I)
    A.append(D.pop()) # empiler(A, depiler(D))
    hanoi(n-1,I,D,A)

def resoudreHanoi(n): # Fonction pour initialiser
    D = [ n-i for i in range(n)] # Initialisation Tours
    I = [ None ] * n # reserver place necessaire
    A = [ None ] * n # simplifie calcul complexite
    print(D,I,A) # Affichage etat au depart
    hanoi(n,D,I,A)
    print(D,I,A) # Affichage etat à l'arrivee

```

```

>>> resoudreHanoi(4)
[4, 3, 2, 1] [ ] [ ]
[ ] [ ] [4, 3, 2, 1]

```

• Complexité

Déterminons la complexité de cet algorithme :

L'appel de `resoudreHanoi(n)` effectue :

1. n opérations élémentaire pour constituer la liste D ,
 2. n opérations élémentaires pour créer la liste I de capacité n ,
 3. n opérations élémentaires pour créer la liste A de capacité n ,
 4. $3.n$ opérations élémentaires pour écrire D, I, A , 2 fois
 5. les opérations nécessaires à l'appel de `hanoi(n,D,I,A)`.
- Au total, si l'on appelle H_n le nombre d'opérations élémentaires nécessaires pour `hanoi(n,D,I,A)`, alors `resoudreHanoi(n)` nécessite H_n+9n opérations élémentaires.
 - Calculons l'ordre de H_n .

```

def hanoi(n, D, I, A): # Partie recursive
    if n!=0:
        hanoi(n-1,D,A,I)
        A.append(D.pop()) # empiler(A, depiler(D))
        hanoi(n-1,I,D,A)

```

On obtient la relation de récurrence :

$$H_0 = 1 \quad H_{k+1} = 1 + H_k + 2 + H_k = 2H_k + 3$$

(1 test, 2 appels de récursifs de coût H_k , et 2 opérations pour dépiler D et empiler A (l'espace ayant été réservé l'ajout en fin de liste se fait en $O(1)$).

Alors H_n est une suite arithmético-géométrique, de terme général $H_n = 4 \times 2^n - 3$ (exercice).

La résolution du problème nécessite $4.2^n - 3 + 9n$ opérations élémentaires : du même ordre que 2^n . Donc la complexité est exponentielle, d'ordre $\Theta(2^n)$.

METHODE GENERALE POUR CALCULER LA COMPLEXITE D'UNE FONCTION RECURSIVE SUIVANT UNE RECURRENCE SIMPLE.

• Les tours de Hanoï : Affichage des mouvements

Deuxième tentative : pour afficher les états successifs.

L'affichage doit se faire au départ, puis après chaque changement.

C'est à dire avant l'appel initial (départ) et après chaque instruction `A.append(D.pop())`.

Le problème est que chaque appel récursif permute l'ordre des 3 tours.

Si l'on se contente de l'instruction `print(D,I,A)` les 3 tours ne seront pas affichées dans l'ordre exact.

Solution. Déclarer la liste $T = [D, I, A]$ comme une **variable globale**.

```

def hanoi(n,D,I,A):
    if n!=0:
        hanoi(n-1,D,A,I)
        A.append(D.pop())
        print(T) # Affichage des trois tours
        hanoi(n-1,I,D,A)

def resoudreHanoi(n):
    D = [ n-i for i in range(n)]
    I = [ ]
    A = [ ]
    # La variable T est declaree globale ,
    # on peut acceder à son contenu de partout :
    global T # D'abord declarer T comme globale
    T = [D, I, A] # Puis lui affecter une valeur
    print(T)
    hanoi(n,D,I,A)

```

```

>>> resoudreHanoi(3)
[[3, 2, 1], [], []]
[[3, 2], [], [1]]
[[3], [2], [1]]
[[3], [2, 1], []]
[[], [2, 1], [3]]
[[1], [2], [3]]
[[1], [], [3, 2]]
[[], [], [3, 2, 1]]

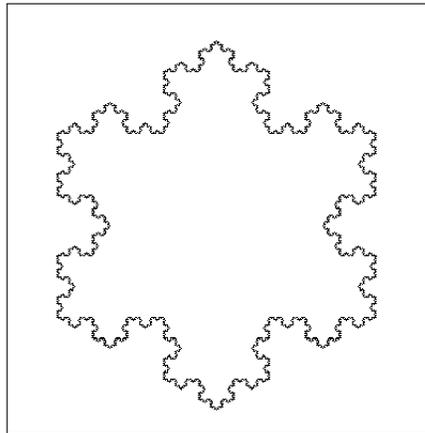
```

Démontrons que l'algorithme décrit résout bien le problème des Tours d'Hanoï en un minimum de déplacements :

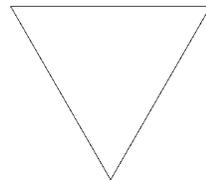
Esquisse : Par récurrence sur n .

Pour résoudre le problème : tôt ou tard il faudra déplacer le disque de plus gros diamètre n . Pour cela nécessairement les $n-1$ premiers disques devront être tous empilés dans le bon ordre sur une même colonne. Pour que le nombre de déplacements soit minimal il faut que le disque n soit déplacé d'entrée sur la colonne de droite.

6.3.3 Tracé du flocon de Von Koch

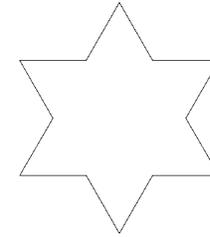


- Cette courbe est construite en partant d'un triangle équilatéral.



- Sur chaque segment :

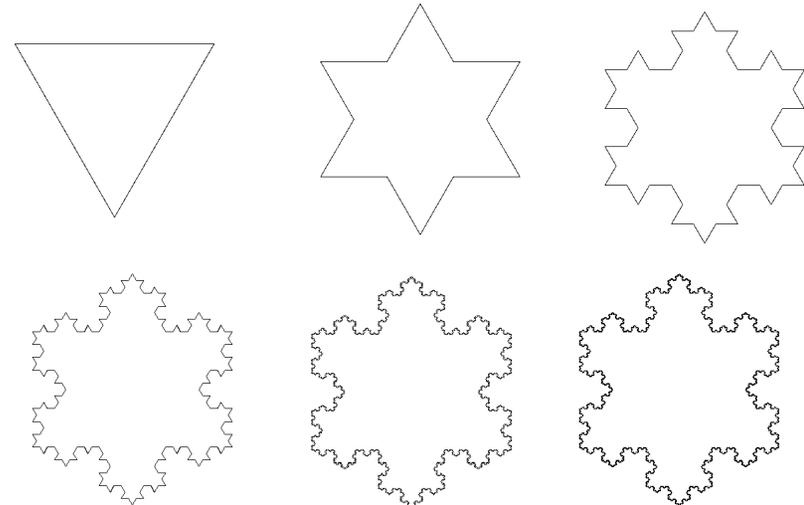
1. Diviser en 3 segments de mêmes longueurs.
2. Considérer le triangle équilatéral extérieur construit sur le segment du milieu.
3. Remplacer ce segment par les 2 autres côtés du triangle équilatéral.



- Sur chaque segment :

1. Diviser en 3 segments de mêmes longueurs.
2. Considérer le triangle équilatéral extérieur construit sur le segment du milieu.
3. Remplacer ce segment par les 2 autres côtés du triangle équilatéral.

- Recommencer avec chaque segment obtenu.



- **Tracé : graphiques avec le module turtle**

- Pour le tracé on utilise le module `turtle` (inspiré du `logo`, (80's)). On déplace une tortue dans le plan (rapporté à un repère orthonormé direct), qui peut tracer sur son passage.

- Principales instructions :

<code>reset()</code>	Efface la fenêtre graphique, réinitialisation
<code>up()</code> , <code>down()</code>	Relève, abaisse le crayon
<code>forward(d)</code> , <code>backward(d)</code>	Avancer, reculer d'une distance <code>d</code>
<code>left(a)</code> , <code>right(a)</code>	Tourner à gauche, droite, d'un angle <code>a</code> en degrés
<code>goto(x,y)</code>	Se déplace au point de coordonnées <code>(x,y)</code>
<code>position()</code>	Retourne la position courante
<code>color(couleur)</code>	Détermine la couleur = 'black', 'blue', 'red', ...
<code>width(l)</code>	Détermine l'épaisseur du trait <code>l</code>
<code>fill(p)</code>	Remplir un contour fermé : <code>p=True</code> et <code>p=False</code> Pour délimiter la figure à remplir
<code>circle(r)</code>	Trace un cercle de rayon <code>r</code>
<code>undo()</code>	Annule la dernière commande

- Voir l'aide en ligne : <https://docs.python.org/3.4/library/turtle.html>

• Tracé par récursivité

Pour le tracé, on peut procéder par récursivité sur chacun des 3 segments du triangle initial :

`vonKoch(longueur, n)`

1. appeler `vonKoch(longueur / 3, n-1)`
2. Tourner à gauche de 60° : `tt.left(60)`
3. appeler `vonKoch(longueur / 3, n-1)`
4. Tourner à droite de 120° : `tt.right(120)`
5. appeler `vonKoch(longueur / 3, n-1)`
6. Tourner à gauche de 60° : `tt.left(60)`
7. appeler `vonKoch(longueur / 3, n-1)`



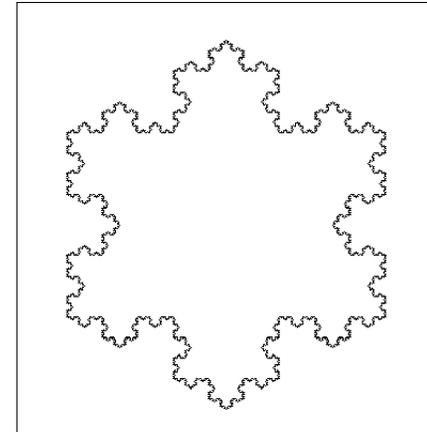
Code

```
def vonkoch(longueur, n):
    if n == 1:
        tt.forward(longueur)
    else:
        l = longueur / 3
        vonkoch(l, n - 1); tt.left(60)
        vonkoch(l, n - 1); tt.right(120)
```

```
vonkoch(l, n - 1); tt.left(60)
vonkoch(l, n - 1)}
```

```
def floconVonKoch(longueur, n):
    tt.pen(speed = 0) # Accélération du mouvement
    tt.hideturtle() # Pour ne pas tracer la tortue
    tt.up()
    # Départ en haut à gauche :
    tt.goto(-longueur/2, longueur/3)
    tt.down()
    for i in range(3):
        vonkoch(longueur, n); tt.right(120)
```

```
>>> floconVonKoch(300, 6)
```



Remarque. Le flocon de Von-Koch a un périmètre infini, qui délimite une aire finie. On peut vérifier que :

$$L_n = (3 \times \text{longueur}) \times \left(\frac{4}{3}\right)^{n-1} \quad \text{Aire}_{n+1} = \text{Aire}_n + \frac{3}{4} \times \left(\frac{4}{9}\right)^n \times \frac{\sqrt{3}}{2} \times L^2$$

6.4 Limitations de la récursivité

Ecrivons deux versions, l'une itérative, l'autre récursive, d'une fonction retournant le terme de rang n de la suite de Fibonacci :

$$u_0 = 0, \quad u_1 = 1, \quad \forall n \in \mathbb{N}, \quad u_{n+2} = u_{n+1} + u_n$$

```
def fibo_iter(n): # Version iterative
    if n==0:
        return 0
    u, v = 0, 1
    for i in range(n-1):
        u, v = v, u+v
    return v
```

```
def fibo_rec(n): # Version recursive
    if n==0:
        return 0
    if n==1:
        return 1
    return fibo_rec(n-1) + fibo_rec(n-2)
```

Comparons leurs temps d'exécution :

```
In [1]: %timeit fibo_iter(20)
1000000 loops, best of 3: 1.98 µs per loop
```

```
In [2]: %timeit fibo\_rec(20)
100 loops, best of 3: 4.6 ms per loop
```

Pour $n = 20$ La version itérative est plus de 2000 fois plus rapide!

```
In [3]: %timeit fibo_iter(30)
1000000 loops, best of 3: 2.84 µs per loop
In [4]: %timeit fibo_rec(30)
1 loops, best of 3: 565 ms per loop
```

Pour $n = 30$ La version itérative est près de 200 000 fois plus rapide!

```
In [9]: %timeit fibo_iter(40)
1000000 loops, best of 3: 3.68 µs per loop
In [8]: %timeit fibo\_rec(40)
1 loops, best of 3: 1min 9s per loop
```

Pour $n = 40$ La version itérative est près de 20 millions de fois plus rapide!
 Pour des valeurs de n plus grandes la version récursive ne répond plus!

• Explication

Modifions le code pour qu'il affiche les rangs calculés :

```
def fiborec(n):
    if n==0:
        print('rang 0')
```

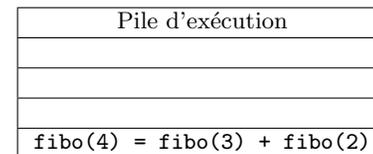
```
        return 0
    if n==1:
        print('rang 1')
        return 1
    print('rang',n)
    return fiborec(n-1) + fiborec(n-2)
```

```
In [19]: fiborec(4)
rang 4
rang 3
rang 2
rang 1
rang 0
rang 1
rang 2
rang 1
rang 0
```

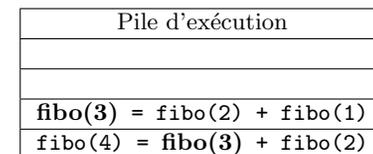
• Explication : usage de la mémoire

```
>>> fiborec(4)
```

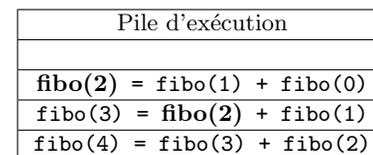
rangs : 4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0



Appels : 4



Appels : 4 - 3



Appels : 4 - 3 - 2

Pile d'exécution
fib(1) = 1
fib(2) = fib(1) + fib(0)
fib(3) = fib(2) + fib(1)
fib(4) = fib(3) + fib(2)

Appels : 4 - 3 - 2 - 1

Pile d'exécution
fib(2) = 1 + fib(0)
fib(3) = fib(2) + fib(1)
fib(4) = fib(3) + fib(2)

Appels : 4 - 3 - 2 - 1

Pile d'exécution
fib(0) = 0
fib(2) = 1 + fib(0)
fib(3) = fib(2) + fib(1)
fib(4) = fib(3) + fib(2)

Appels : 4 - 3 - 2 - 1 - 0

Pile d'exécution
fib(2) = 1
fib(3) = fib(2) + fib(1)
fib(4) = fib(3) + fib(2)

Appels : 4 - 3 - 2 - 1 - 0

Pile d'exécution
fib(3) = 1 + fib(1)
fib(4) = fib(3) + fib(2)

Appels : 4 - 3 - 2 - 1 - 0

Pile d'exécution
fib(1) = 1
fib(3) = 1 + fib(1)
fib(4) = fib(3) + fib(2)

Appels : 4 - 3 - 2 - 1 - 0 - 1

Pile d'exécution
fib(3) = 2
fib(4) = fib(3) + fib(2)

Appels : 4 - 3 - 2 - 1 - 0 - 1

Pile d'exécution
fib(4) = 2 + fib(2)

Appels : 4 - 3 - 2 - 1 - 0 - 1 - 2

Pile d'exécution
fib(2) = fib(1) + fib(0)
fib(4) = 2 + fib(2)

Appels : 4 - 3 - 2 - 1 - 0 - 1 - 2

Pile d'exécution
fib(1) = 1
fib(2) = fib(1) + fib(0)
fib(4) = 2 + fib(2)

Appels : 4 - 3 - 2 - 1 - 0 - 1 - 2 - 1

Pile d'exécution
fib(0) = 0
fib(2) = 1 + fib(0)
fib(4) = 2 + fib(2)

Appels : 4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0

Pile d'exécution
fib(2) = 1
fib(4) = 2 + fib(2)

Appels : 4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0

- considérer le premier élément dans la partie du tableau non-triée
- L'insérer progressivement à sa bonne place dans la partie triée du tableau, en la faisant descendre d'une position tant que sa valeur reste inférieure à celle de l'élément qui le précède.
- Recommencer avec l'élément suivant du tableau non-trié.

La complexité temporelle du Tri par insertion est quadratique dans le pire des cas (du à l'insertion dans le tableau trié).

Dans le meilleur des cas (celui d'un tableau déjà trié), le Tri par insertion est linéaire (insertion immédiate). (\implies c'est un assez bon algorithme!)

On montre que sa complexité en moyenne est quadratique (facile : en probabilité uniforme le i -ème plus petit élément a autant de probabilités d'être proche de la position i que de la position $N - i$).

Sa complexité spatiale est bornée, en $O(1)$, car c'est un tri en place.

Peut-on faire mieux ?

- Dans tous les cas, (meilleur, pire, en moyenne), il est évident qu'on ne peut pas faire mieux qu'une complexité temporelle linéaire : en effet chaque élément du tableau devra au moins être lu.
- Les tris par sélection et par insertion sont tous deux quadratiques en moyenne et dans le pire des cas. A priori toutes les complexités, dans le pire des cas, ou en moyenne, entre linéaire et quadratique seraient envisageables.
- C'est naïf : on démontre que, dans le pire des cas et en moyenne, il n'est pas possible de faire mieux qu'en temps $\Theta(n \cdot \log(n))$. De tels algorithmes sont alors optimaux.
- De tels algorithmes optimaux sont basés sur le principe de **Diviser pour régner** :

Pour résoudre un problème sur une donnée de taille N le réduire récursivement à $K \geq 2$ résolutions du problème sur des données de taille N/K .

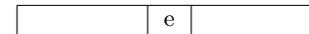
Applications pour le Tri : Ramener récursivement le tri d'un tableau aux deux tris de deux sous-tableaux de taille moitié. (Pour que ça fonctionne la fusion de deux sous-tableaux triés en un seul tableau trié doit être rapide).

6.5.2 Le tri rapide

Le tri rapide :

- Choisir arbitrairement un élément e dans le tableau T ,
- Décomposer les autres éléments du tableau en deux sous-tableaux :
 - T^- constitué des éléments inférieurs à e ,
 - T^+ constitué des éléments supérieurs à e .
- Après appels récursifs sur les deux sous-tableaux T^- et T^+ , le tableau T trié s'obtient en concaténant les sous-tableaux T^- trié, suivi de e , suivi de T^+ trié.

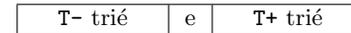
- Choisir arbitrairement un élément e :



- Constituer les sous listes T^- et T^+ (en temps linéaire) :



- Recommencer récursivement avec les tableaux T^- et T^+ puis reconstituer T :



- Le tableau T est alors trié.

Code du tri rapide

Algorithme : en pseudo-code :

```

Tri_rapide(Tableau T) :
  N ← longueur(T)
  SI N ≤ 1 ALORS :      # Appel terminal
    retourner T
  FIN SI
  e ← T[N//2]          # Élément au milieu du tableau
  Retirer e de T      # retiré de T
  T1, T2 sont 2 Tableaux vides
  POUR chaque élément x dans T : # Constitution de T1 et T2
    Si x ≤ e, ALORS :
      Insérer x à la fin de T1
    SINON :
      Insérer x à la fin de T2
  FIN SI
  FIN POUR
  Retourner Tri_rapide(T1) + [e] + Tri_rapide(T2) # Appel rec.

```

Algorithme : en python :

```

def Tri_rapide(T) :
  N = len(T)
  if N <= 1: # Appel terminal
    return T
  e = T.pop(N//2) # Retirer element milieu tableau
  T1, T2 = [], []
  for x in T : # Constitution de T1 et T2
    if x <= e:
      T1.append(x)
    else:
      T2.append(x)
  return Tri_rapide(T1)+[e]+Tri_rapide(T2) # Appel rec.

```

• L'algorithme s'arrête puisque à chaque appel récursif chacun des deux tableau est de longueur strictement inférieure. Lorsqu'un tableau est de longueur inférieure ou égale à 1, l'appel récursif s'achève.

• Montrons que le tableau obtenu est trié : par récurrence forte sur sa taille N .

Initialisation. Pour un tableau de longueur 0 ou 1, l'algorithme retourne le même tableau, qui est trié.

Hérédité. Pour un tableau de longueur N . L'algorithme retourne la concaténation de $\text{Tri_rapide}(T1)$, de $[e]$ et de $\text{Tri_rapide}(T2)$. Les tableaux $T1$ et $T2$ sont de longueur inférieure strictement à N , et donc par hypothèse de récurrence $\text{Tri_rapide}(T1)$ et $\text{Tri_rapide}(T2)$ sont triés. Puisque par construction tous les éléments de $\text{Tri_rapide}(T1)$ sont inférieurs à e et tous les éléments de $\text{Tri_rapide}(T2)$ sont supérieurs à e , le tableau obtenu par concaténation est lui aussi trié. \square

Complexité du tri rapide

Pour l'appel pour un tableau de taille N , les seules opérations en temps non-bornées sont :

- $T.\text{pop}(N//2)$: prend un temps linéaire d'ordre $\Theta(N/2)$.
- La boucle **for** (constitution des deux sous-tableaux) prend un temps linéaire $\Theta(N)$.
- L'appel récursif sur deux tableaux de taille $K - 1$ et $N - K$.
- La concaténation $T1 + [e] + T2$ prend un temps linéaire $\Theta(N)$.

Ainsi si $C(N)$ compte le nombre d'opérations pour un tableau de longueur N :

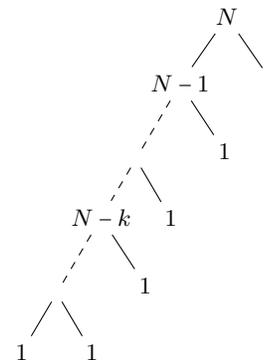
$$C(N) = C(K - 1) + C(N - K) + \Theta(N)$$

• Si $C(N)$ compte le nombre d'opérations pour un tableau de longueur N :

$$C(N) = C(K - 1) + C(N - K) + \Theta(N)$$

$$\begin{aligned}
 C(N) &= C(N_1) + C(N_2) + \Theta(N) && \text{avec } N_1 + N_2 = N - 1 \\
 &= \Theta(N_1) + \Theta(N_2) + \Theta(N) \\
 &\quad + C(N_{11}) + C(N_{12}) + C(N_{21}) + C(N_{22}) && N_{11} + N_{12} + N_{21} + N_{22} \approx N \\
 &= \Theta(N) + \Theta(N) \\
 &\quad + C(N_{11}) + C(N_{12}) + C(N_{21}) + C(N_{22}) \\
 &= \Theta(N) + \Theta(N) + \Theta(N) \\
 &\quad + \dots \\
 &\vdots \\
 &= \underbrace{\Theta(N) + \dots + \Theta(N)}_{\text{au plus } N \text{ fois}} \\
 &= O(N^2)
 \end{aligned}$$

Dans le pire des cas, lorsque l'élément choisi est toujours un élément extrême (min ou max) :

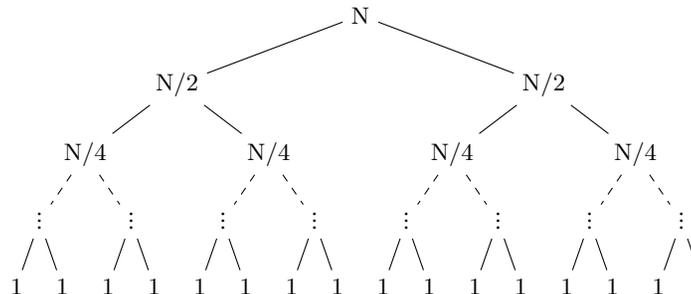


La profondeur de l'arbre est N . De l'ordre de $N - k$ opérations à chaque "étage" k .

$$\implies C(N) = \Theta\left(\sum_{n=1}^N n\right) = \Theta\left(N \times \frac{(N+1)}{2}\right) = \Theta(N^2)$$

complexité quadratique

Dans le meilleur des cas, lorsque l'élément choisi est médian :



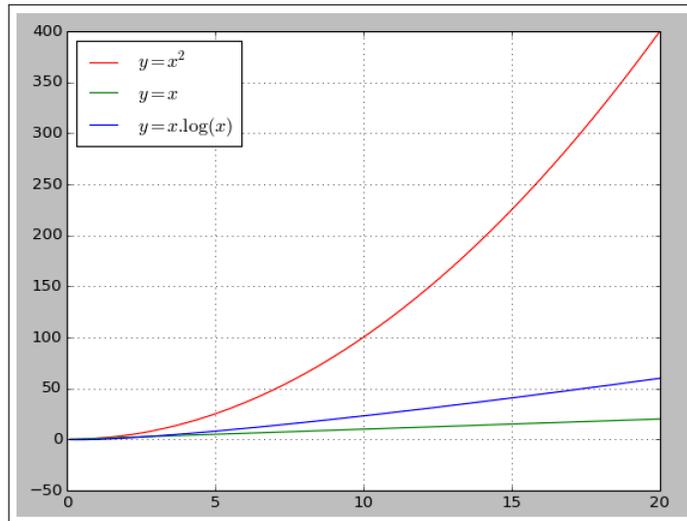
La profondeur de l'arbre est $\log_2(N)$ puisque : $N = 2^{\log_2(N)}$. De l'ordre de N opérations à chaque étage.

$$\implies C(N) = \Theta\left(\sum_{k=0}^{\log_2(N)} 2^k \times \frac{N}{2^k}\right) = \Theta((\log_2(N) + 1) \times N) = \Theta(N \log(N))$$

Dans le meilleur des cas la complexité est en $N \log(N)$. (c'est mieux que quadratique!)

On démontre qu'en moyenne la complexité est aussi d'ordre $N \log(N)$.

D'où le nom de TRI RAPIDE (en anglais QUICK SORT).



Tracé des courbes d'équations $y = x$, $y = x \cdot \log(x)$ et $y = x^2$ sur $]0, 20]$.

- Le tri rapide a une complexité temporelle (pour un tableau de taille N) :
 - quadratique, $O(N^2)$ dans le pire des cas,
 - en $N \log(N)$ dans le meilleur des cas,
 - On démontre que sa complexité en moyenne est en $N \log(N)$.

De plus en pratique il s'avère très rapide, d'où son nom. C'est le tri implémenté par de nombreuses fonctions de tri, notamment par la méthode `sort()` sur les listes en python.

Il est possible (mais pas trivial) de le faire "en place", c'est à dire en n'utilisant que le tableau passé en argument, pour économiser la mémoire (complexité en mémoire en $O(1)$ contre $O(N^2)$ dans le pire des cas pour cette version).

Application : calcul de la médiane

Pour calculer la médiane d'une liste de nombres :

1. On pourrait appliquer un tri rapide puis récupérer l'élément de position médiane dans le tableau obtenu.
2. Mais : au cours du tri rapide il suffit de se consacrer à la seule partie contenant plus de la moitié des éléments :

On se donne un nombre s . Pour un tableau de N élément, lorsque $s = \left\lceil \frac{N}{2} \right\rceil$ on retournera la médiane ; plus généralement le $(s+1)$ -ième élément dans le tableau ordonné :

Par tri rapide (pseudo-code) :

```
recherche(T,s):
    Soit e=T.pop(len(T)//2)
    Soient T1, T2 obtenu par tri rapide avec pivot e.
    Si len(T1)==s : renvoyer e.
    Sinon, si len(T1)>s : renvoyer recherche(T1,s)
    Sinon (si len(T1)<s) : renvoyer recherche(T2,s-len(T1)-1)
```

```
def recherche(T,s):
    e = T.pop(len(T)//2)
    T1, T2 = [], []
    for x in T:
        if x<=e:
            T1.append(x)
        else:
            T2.append(x)
    if len(T1) == s:
        return e
    elif len(T1) > s:
        return recherche(T1,s)
    else:
        return recherche(T2,s-len(T1)-1)

def mediane(L):
    T = L[:] # Copie superficielle pour pas modifier L
    return recherche(T,len(T)//2)
```

Complexité $O(N \log N)$; il existe un algorithme de complexité linéaire.

6.5.3 Le tri fusion

Le tri fusion est un autre exemple d'algorithme de tri basé sur le principe de "Diviser pour régner".

Il est basé sur le fait que fusionner deux tableaux triés en un seul tableau trié $T1, T2$ se fait en temps linéaire (sur le nombre total d'éléments, dans le pire et le meilleur des cas) :

- Pseudo-code pour fusionner deux tableaux ordonnés :

```

i1, i2 = 0
T = tableau vide
TANT QUE l'on n'a pas atteint la fin d'un des tableaux:
  SI T1[i1] <= T2[i2] ALORS
    Insérer T1[i1] à la fin de T
    incrémenter i1
  SINON
    Insérer T2[i2] à la fin de T
    incrémenter i2
FIN TANT QUE
Insérer les éléments restants du tableau non vide en fin de T
    
```

• Code de la fusion de deux tableaux ordonnés.

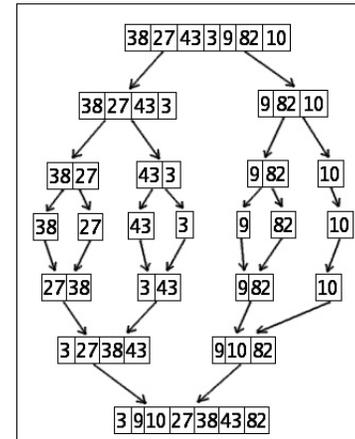
```

def fusion(T1,T2):
  n1, n2 = len(T1), len(T2) # Longueurs de T1, T2
  i1, i2 = 0,0 # Indices
  T = [ ] # Contendra le tableau fusionne
  # Tant que T1, T2 sont non vides
  while (i2 < n2) and (i1 < n1):
    if T1[i1] <= T2[i2]:
      T.append(T1[i1])
      i1 += 1
    else:
      T.append(T2[i2])
      i2 += 1
  while (i1 < n1): # Tant que T1 non fini
    T.append(T1[i1])
    i1 += 1
  while (i2 < n2): # Tant que T2 non fini
    T.append(T2[i2])
    i2 += 1
  return T
    
```

• Il met en oeuvre la récursivité :

- Décomposer le tableau en deux sous-tableaux de même longueur (± 1)
- Appliquer l'algorithme récursivement sur chacun des 2 sous-tableaux
- Puis fusionner les 2 sous-tableaux triés en un tableau trié.

Il est possible, mais compliqué et pénalisant de l'effectuer en place (on ne le verra pas).



(source du graphique : wikipedia)

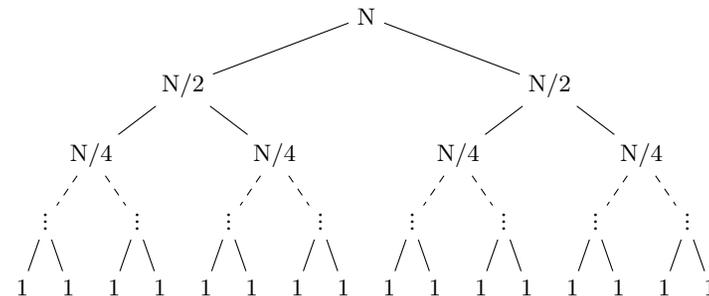
Code du Tri fusion

```

def tri_fusion(T):
  n = len(T)
  if n == 1: # Cas terminal
    return T
  T1 = tri_fusion(T[:n//2]) # Appel rec. 1er tableau
  T2 = tri_fusion(T[n//2:]) # Appel rec. 2eme tableau
  return fusion(T1,T2) # Retourner la fusion des 2
    
```

• **Correction** : (esquisse) par récurrence forte sur la longueur du tableau : après chaque fusion le tableau obtenu est trié.

• **Complexité dans le pire et le meilleur des cas** :



La profondeur de l'arbre est $\log_2(N)$. A chaque étage $\Theta(N)$ opérations :

Complexité optimale : $\Theta(N \log(N))$