Le dernier algorithme de tri que nous verrons est le tri rapide. Il applique le principe de "diviser pour régner". Pour cela nous avons d'utiliser la notion récureivité, c'est à dire d'une fonction qui s'appelle elle-même.

2 La récursivité

Rappelons qu'une fonction en `python` peut appeler toute autre fonction qui lui est visible, c'est à dire dont la définition dans le code la précède, au sens large.

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 récurrence
```

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.

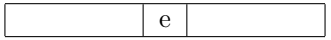
3 Le tri rapide

3.1 Principe

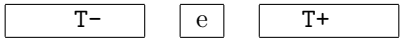
Le tri rapide :

- 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.

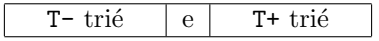
Choisir arbitrairement un élément `e` :



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



Recommencer <u>récursivement</u> avec les tableaux `T-` et `T+` puis reconstituer `T` :



Le tableau `T` est alors trié.

3.2 Exemple :

(1) Tri rapide à la main de `[7, 3, 6, 4, 2, 5, 1]` :

- Appel sur `T = [7, 3, 6, 4, 2, 5, 1]`
  `e = 4,  T1 = [3, 2, 1],  T2 = [7, 6, 5]`
- Appel récursif sur `T1 = [3, 2, 1]` :
  `e = 2,  T11 = [1],  T12 = [3]`

  $\Longrightarrow$ `T1` devient : `T11 + [2] + T12 = [1, 2, 3]`
- Appel récursif sur `T2 = [7,6,5]`
  `e = 6,  T21 = [5],  T22 = [7]`

  $\Longrightarrow$ `T2` devient : `T21 + [6] + T22 = [5,6,7]`

  $\Longrightarrow$ `T` devient : `T1 + [4] + T2 = [1,2,3,4,5,6,7]`

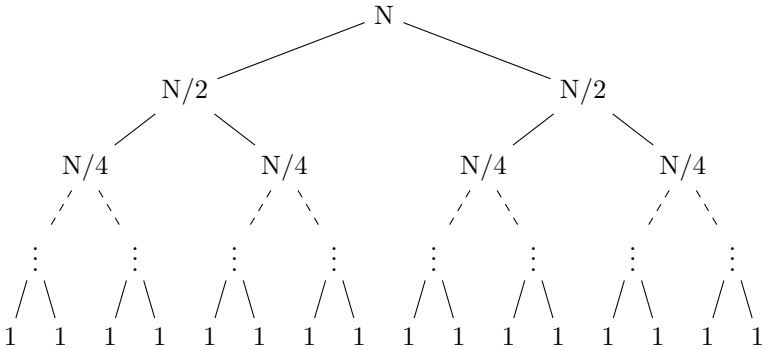
3.3 Algorithme

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]     # Elément 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.
```

3.4 complexité

On démontre que sa complexité dans le pire des cas est quadratique. Son intérêt vient du fait que sa complexité en moyenne est en $O(n\log(n))$, bien mieux que 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.

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

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).

3.5 Code python

```
Tri_rapide(T) :
    N = len(T)
    if N <= 1:    # Appel terminal
        return T
    e = T.pop(N//2)  # Retirer l'élément au milieu du 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.
```

3.6 Correction de l'algorithme

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 `Tri_rapide(T1)`, de `[e]` et de `Tri_rapide(T2)`. Les tableaux `T1` et `T2` sont de longueur inférieure strictement à N, et donc par hypothèse de récurrence `Tri_rapide(T1)` et `Tri_rapide(T2)` sont triés. Puisque par construction tous les éléments de `Tri_rapide(T1)` sont inférieurs à `e` et tous les éléments de `Tri_rapide(T2)` sont supérieurs à `e`, le tableau obtenu par concaténation est lui aussi trié. □