Algorithmes au programme d'IPT

Table des matières

I. Recherches et calculs dans une liste 2
1) Recherche d'un élément 2
2) Recherche du minimum ou du maximum 2
3) Calcul de la valeur moyenne et de la variance 2
4) Recherche par dichotomie dans une liste triée 3

II. Recherche des zéros d'une application à valeurs réelles 4
1) Algorithme de dichotomie 4
a) Démarche intuitive 4
b) Fin de l'algorithme 5
c) Mise en oeuvre 5
2) Méthode de Newton 5
a) Démarche intuitive 6
b) Condition d'arrêt de l'algorithme 6
c) Mise en œuvre 7

III.Intégration d'une application 7
1) Méthode d'Euler 7
2) Méthode des trapèzes 8

IV.Systèmes différentiels et méthode d'Euler 9
1) Méthode d'Euler 1D 9
2) Méthode d'Euler 2D 9

V. Algorithmes de tri 11
1) Le tri bulle 11
2) Le tri par sélection 12
3) Tri par insertion 12
4) Tri par comptage 12
5) Tri fusion 13
6) Tri rapide (quick sort) 14

I. Recherches et calculs dans une liste

1) Recherche d'un élément

Soit `L` une liste : chaque élément de `L` peut être accédé en lecture comme en écriture par son indice `i`$\in [\![0, n-1]\!]$. Ainsi : `L[i]` représente l'élément d'indice `i`.

Nous souhaitons savoir si un objet x est élément de `L`. Pour cela, on peut parcourir toute la liste jusqu'à rencontrer (ou non) x. Voici ce que cela donne, sous la forme d'une fonction Python qui retourne `True` ou `False` selon que x appartient ou non à `L`. On suppose que `L` contient au moins 1 élément.

Recherche dans une liste

```
1  def appartientListe(L, x) :
2      n = len(L)
3      i = 0
4      while i < n :
5          if L[i] == x :
6              return True
7          i += 1
8      return False
```

2) Recherche du minimum ou du maximum

Faisons-le pour le maximum. On suppose que les éléments de la liste `L` obéissent à une relation d'ordre total. L'idée est de définir une variable `max` contenant initialement `L[0]` (premier élément de la liste). On parcourt ensuite toute la liste et, à chaque fois qu'on rencontre un élément `x > max`, on met sa valeur dans `max`.

Recherche du plus grand élément (On suppose que `L` contient au moins 1 élément)

```
1  def maxListe(L) :
2      max = L[0]
3      n = len(L)
4      for i in range(n) :
5          if L[i] > max :
6              max = L[i]
7      return max
```

Exercice : écrire la fonction de recherche du plus petit élément de `L`.

3) Calcul de la valeur moyenne et de la variance

Étant donné une liste de N nombres x_0, x_1, ..., x_{N-1}, nous pouvons définir la valeur moyenne m ainsi que la variance V par :

$$m = \frac{1}{N}\sum_{i=0}^{N-1} x_i \quad \text{et} \quad V = \frac{1}{N}\sum_{i=0}^{N-1}(x_i - m)^2$$

En développant le carré : $(x_i - m)^2 = x_i^2 - 2x_i m + m^2$ et en notant que :

$$\sum_{i=0}^{N-1}(x_i^2 - 2\,x_i m + m^2) = \left(\sum_{i=0}^{N-1} x_i^2\right) - 2Nm^2 + Nm^2 = \left(\sum_{i=0}^{N-1} x_i^2\right) - Nm^2$$

il vient :

$$\boxed{V = \frac{1}{N}\left(\sum_{i=0}^{N-1} x_i^2\right) - m^2}$$

Les deux fonctions `moyenne(L)` et `variance(L)` calculent et renvoient la valeur moyenne et la variance d'une liste de nombres qu'on lui passe en paramètre : `L = [x0, x1, ..., xN]`.

Calcul de la valeur moyenne

```
1  def moyenne(L) :
2     N = len(L)
3     somme = 0
4     for i in range(N) :
5        somme += L[i]
6     return somme/N
```

Calcul de la variance

```
1  def variance(L) :
2     N = len(L)
3     somme = 0
4     somme_carres = 0
5     for i in range(N) :
6        somme += L[i]
7        somme_carres += L[i]**2
8     moy = somme/N
9     return somme_carres/N - moy**2
```

4) Recherche par dichotomie dans une liste triée

Soit $L = [x_0, x_1, ..., x_{n-1}]$ une liste de nombres triés et rangés par valeurs croissantes : $x_0 \leqslant x_1 \leqslant ... \leqslant x_{n-1}$. Nous souhaitons savoir si un nombre x appartient à cette liste en utilisant le principe de la recherche dichotomique.

Cette recherche est basée sur le principe suivant, en notant $L[p...q]$ une sous-liste de L qui commence à l'indice p et finit à l'indice q :

- Calculer l'indice $m = (p+q)//2$ situé au milieu de la sous-liste.
- Si $x == L[m]$, c'est gagné !
- Sinon si $x < L[m]$, chercher x dans la sous-liste $L[p...m-1]$.
- Sinon (dans ce cas $x > L[m]$ forcément), chercher x dans la sous-liste $L[m+1...q]$.

On peut rédiger cet algorithme de façon itérative ou récursive. Le voici programmé sous la forme de deux fonctions qui prennent en paramètre la liste triée `L`, les deux indices `p` et `q` ainsi que le nombre `x` dont on veut savoir s'il appartient à `L`. Ces fonctions renvoient `True` ou `False`. Si on veut faire la recherche dans toute la liste `L`, il suffit d'appeler ces fonctions avec `p = 0` et `q = n - 1`.

Recherche par dichotomie dans une liste triée (version itérative)

```
1   def recherche_dicho_it(L, p, q x) :
2      min = p
3      max = q
4      while min <= max :
5         m = (min + max)//2
6         if x == L[m] :
7            return True
8         elif x < L[m] :
9            max = m - 1
10        else :
11           min = m + 1
12     return False
```

En voici maintenant la version récursive :

Recherche par dichotomie dans une liste triée (version récursive)

```
1  def recherche_dicho_rec(L, p, q, x) :
2     if p > q :
3        return False
4     else :
5        m = (p + q)//2
6        if x == L[m] :
7           return True
8        elif x < L[m] :
9           return recherche_dicho_rec(L,p, m - 1, x) :
10       else :
11          return recherche_dicho_rec(L,m + 1,q, x) :
```

II. Recherche des zéros d'une application à valeurs réelles

Soit $f : I = [a, b] \longrightarrow \mathbb{R}$ une applications réelle $(a > b)$, continue sur I et admettant au moins une valeur $c \in]a, b[$ telle que $f(c) = 0$. Quitte à choisir correctement a et b, nous allons supposer qu'il existe **un et un seul réel** $c \in]a, b[$ tel que $f(c) = 0$: le réel c est un zéro de l'application f.

Deux algorithmes permettant de trouver c sont au programme des classes préparatoires : *l'algorithme de dichotomie* et *l'algorithme de Newton.*

1) Algorithme de dichotomie

a) Démarche intuitive

Prérequis : l'application f doit vérifier $f(a) \times f(b) < 0$, c'est à dire que $f(a)$ et $f(b)$ sont de signes opposés.

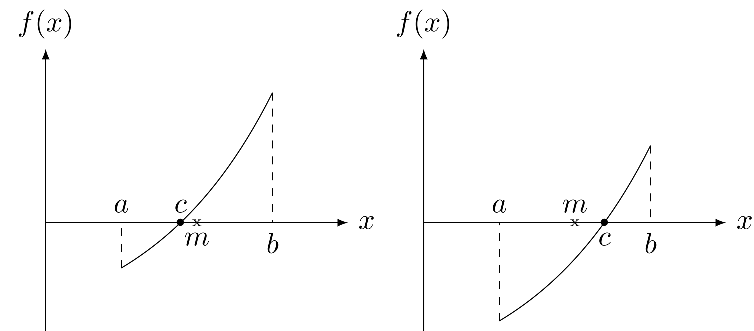


Figure 1 –

Considérons le point milieu $m = (a + b)/2$ de a et b.

- Sur la Figure 1 à gauche nous avons $f(a) \times f(m) < 0$ et nous resserrons l'intervalle de recherche en prenant comme nouvelle valeur de b : $b_1 = m$ sans rien changer à a : $a_1 = a$.
- Sur la Figure 1 à droite nous avons au contraire $f(a) \times f(m) > 0$ et c'est la valeur de a qui change et devient $a_1 = m$ tandis que celle de b ne change pas : $b_1 = b$.
- On recommence ensuite avec l'intervalle $[a_1, b_1]$ et on pose $m = (a_1 + b_1)/2$. À nouveau, si $f(a_1) \times f(m) < 0$ nous définissons $b_2 = m$ et $a_2 = a_1$ et si $f(a) \times f(m) > 0$ nous posons $a_2 = m$ et $b_2 = b_1$.

On voit que cela permet de définir deux suites (a_n) et (b_n) qui encadrent c et se rapprochent progressivement de sa valeur. En d'autres termes :

$$\lim a_n = \lim b_n = c$$

De façon générale, le principe de calcul des suites (a_n) et (b_n) se fait par récurrence selon le schéma :

$$m = \frac{a_n + b_n}{2} \quad \text{et} \quad (a_{n+1}, b_{n+1}) = \begin{cases} (a_n, m) & \text{si} \quad f(a_n) \times f(m) < 0 \\ (m, b_n) & \text{si} \quad f(a_n) \times f(m) > 0 \end{cases}$$

b) Fin de l'algorithme

Si $f(m) == 0$ c'est terminé mais cela est très improbable avec des nombres flottants.

La condition d'arrêt se fait grâce à une précision représentée par un flottant $\varepsilon > 0$. Il y a deux façons d'écrire cette condition d'arrêt :

1. Soit la condition porte sur la fonction f elle-même. En effet, la suite (c_n) telle que $\forall\, n \in \mathbb{N},\ c_n = (a_n + b_n)/2$ est convergente de limite c. La continuité de f implique alors que la suite $(f(c_n))$ converge vers $f(c) = 0$.

   Nous arrêtons donc le calcul des a_n et b_n à partir du rang N qui vérifie :

   $$\left| f\left(\frac{a_N + b_N}{2}\right) \right| < \varepsilon$$

2. Soit la condition porte sur la longueur de l'intervalle $[a_n, b_n]$ et le processus s'arrête à partir du rang N tel que :

   $$|b_N - a_N| < \varepsilon$$

c) Mise en oeuvre

Recherche par dichotomie du zéro d'une fonction

```
def rechercheDichoZero(f, a, b, epsilon) :
    min = a
    max = b
    while (max - min) > epsilon :
        m = (max + min)/2
        if f(m) == 0 :
            return m
        elif f(min)*f(m) > 0 :
            min = m
        else :
            max = m
    return (min+max)/2
```

2) Méthode de Newton

On suppose que l'application f est de classe C^1 sur un intervalle $I = [a, b]$ de $\mathbb{R}$ $(b > a)$ et on se place dans le cas où il existe un unique réel $c \in]a, b[$ tel que :

$$f(c) = 0 \quad \text{et} \quad f'(c) \neq 0$$

Quitte à restreindre l'intervalle I, nous allons supposer de plus que la dérivée f' garde un signe constant sur I.

a) Démarche intuitive

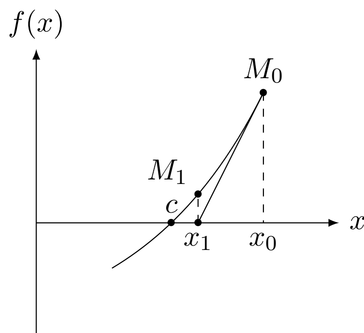


Figure 2 –

Prenons un point M_0 d'abscisse x_0 et traçons la tangente en ce point à la courbe représentative de f. Cette droite tangente coupe l'axe des abscisses en x_1 qui est l'abscisse du point M_1. Nous constatons que x_1 s'est rapproché de c (Figure 2).

En recommençant la même opération avec le point M_1, nous traçons la tangente à la courbe en M_1 et celle-ci coupe l'axe des abscisses en x_2 (non dessiné sur la Figure 2 pour que celle-ci soit lisible) qui est encore plus proche de c.

De proche en proche, nous pouvons construire ainsi une suite (x_n) telle que x_{n+1} soit l'intersection avec l'axe des abscisses de la tangente à la courbe au point M_n d'abscisse x_n. Cette suite (x_n) est alors convergente, de limite c.

Quelle est alors la relation générale entre x_{n+1} et x_n ? L'équation

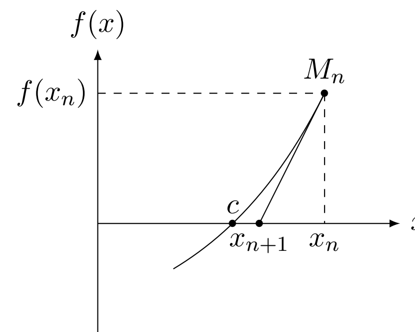


Figure 3 –

de la tangente en M_n est :

$$y = f'(x_n)(x - x_n) + f(x_n)$$

Celle-ci coupe l'axe des abscisses lorsque $x = x_{n+1}$, solution de :

$$0 = f'(x_n)(x_{n+1} - x_n) + f(x_n) \iff \boxed{x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}}$$

b) Condition d'arrêt de l'algorithme

Il existe deux façons d'arrêter l'algorithme, lorsque x_n sera suffisamment proche de c grâce à un nombre flottant ε qui donne la précision du calcul :

1. Si une suite (x_n) converge, alors elle vérifie le critère de Cauchy qui donne comme cas particulier :

$$\forall \varepsilon \in \mathbb{R}_+^*, \ \exists N \in \mathbb{N}, \ \forall n \geqslant N, \ |x_{n+1} - x_n| < \varepsilon$$

   En se basant sur ce schéma, on se donne un flottant $\varepsilon > 0$ et on décide d'arrêter l'algorithme dès qu'on a atteint le plus petit entier naturel N vérifiant $|x_{N+1} - x_N| < \varepsilon$.

2. Une autre façon de raisonner est de considérer la suite $f(x_n)$ qui tend vers 0. En effet, f étant continue et (x_n) ayant pour limite c, la suite $[f(x_n)]$ est convergente, de limite $f(c) = 0$. Ainsi, pour tout réel $\varepsilon > 0$, il existe un entier naturel N tel que :

$$\forall\, n \geqslant N, |f(x_n)| < \varepsilon$$

Cela permet de trouver un autre critère de terminaison de l'algorithme : on se donne $\varepsilon > 0$ et on arrête l'algorithme dès qu'on a trouvé un entier naturel N tel que $|f(x_N)| < \varepsilon$.

c) Mise en œuvre

`f` représente la fonction et `fd` sa dérivée.

```
1 def newton(f, fd, x0, epsilon) :
2    x = x0
3    y = x0 - f(x0)/fd(x0)
4    while abs(y - x) > epsilon :
5       x = y
6       y = x - f(x)/fd(x)
7    return y
```

ou bien

```
1 def newton(f, fd, x0, epsilon) :
2    x = x0
3    while abs( f(x) ) > epsilon :
4       x = x - f(x)/fd(x)
5    return y
```

III. Intégration d'une application

Soit $f : I = [a, b] \longrightarrow \mathbb{R}$ une application à valeurs réelles, continue sur l'intervalle fermé borné $I = [a, b]$ $(b > a)$. On sait que f est alors intégrable sur I. On expose dans cette section deux méthodes au programme des CPGE qui permettent de calculer :

$$J = \int_a^b f(x)\,\mathrm{d}x$$

de façon approchée : la **méthode d'Euler** et la **méthode des trapèzes**.

Nous commençons par créer une subdivision régulière $(a_i)_{0\leqslant i\leqslant n}$ de l'intervalle $[a, b]$, de sorte que :

$$a_i = a + i\,\frac{b-a}{n}$$

Nous aurons donc $a_0 = a$ et $a_n = b$.

1) Méthode d'Euler

Dans la méthode d'Euler, sur chaque intervalle fermé $[a_i, a_{i+1}]$ on remplace f par l'application constante $g_i : x \longmapsto f(a_i)$. On calcule :

$$\int_{a_i}^{a_{i+1}} f(x)\,\mathrm{d}x \approx f(a_i) \times (a_{i+1} - a_i) = f(a_i)\,\frac{b-a}{n}$$

Une valeur approchée de J est donc :

$$\boxed{J = \sum_{i=0}^{n-1} \int_{a_i}^{a_{i+1}} f(x)\,\mathrm{d}x \approx \frac{b-a}{n} \sum_{i=0}^{n-1} f(a_i)}$$

Mise en oeuvre : on suppose que $n > 2$:

```
1  def intEuler(f, a, b, n) :
2     somme = 0
3     for i in range(n) :
4        x = a + i*(b - a)/n
5        somme = somme + f(x)
6     J = (b - a)/n * somme
7  return J
```

2) Méthode des trapèzes

Dans la méthode des trapèzes, sur chaque intervalle fermé $[a_i, a_{i+1}]$, on remplace f par l'application affine g_i qui relie les points $(a_i, f(a_i))$ et $(a_{i+1}, f(a_{i+1}))$.

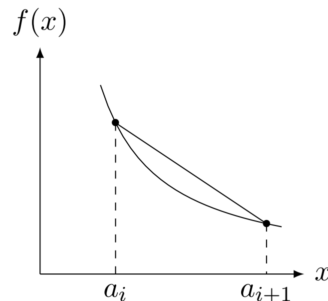


Figure 4 –

On calcule facilement :

$$\int_{a_i}^{a_{i+1}} g_i(x)\,\mathrm{d}x = \frac{f(a_i) + f(a_{i+1})}{2} \times (a_{i+1} - a_i) = \frac{f(a_i) + f(a_{i+1})}{2}\,\frac{b-a}{n}$$

(il s'agit en fait de l'air du trapèze). Une valeur approchée de J est donc :

$$\overline{J} = \sum_{i=0}^{n-1} \int_{a_i}^{a_{i+1}} g_i(x)\,\mathrm{d}x = \frac{b-a}{n} \sum_{i=0}^{n-1} \frac{f(a_i) + f(a_{i+1})}{2}$$
$$= \frac{b-a}{n} \left[\frac{f(a) + f(b)}{2} + f(a_1) + \dots + f(a_{n-1}) \right]$$

On peut démontrer le résultat suivant, qui donne un majorant de l'erreur commise en remplaçant J par $\overline{J}$:

Majoration de l'erreur

M désignant un majorant de $|f''|$ sur $[a, b]$ nous avons :

$$|J - \overline{J}\,| \leqslant \frac{M\,(b-a)^3}{12n^2}$$

Mise en oeuvre : on suppose $n > 2$

```
1  def intTrapeze(f, a, b, n) :
2     somme = ( f(a) + f(b) ) / 2
3     for i in range(1, n) :
4        x = a + i*(b-a)/n
5        somme = somme + f(x)
6     J = (b - a)/n * somme
7     return J
```

IV. Systèmes différentiels et méthode d'Euler

On cherche la solution approchée d'un système différentiel de la forme :

$$\begin{cases} y'(t) & = & f(t, y(t)) \\ y(t_0) & = & y_0 \end{cases}$$

où y' désigne la dérivée d'une application y. Il s'agit d'un système différentiel du premier ordre puisque seule la dérivée première y' intervient.

1) Méthode d'Euler 1D

Nous supposons que nous cherchons une solution $y : t \longrightarrow y(t)$ définie sur un intervalle $I = [t_0, t_F]$ ($t_F > t_0$), de sorte que :

$$\forall\, t \in I,\ y'(t) = f(t, y(t)) \quad \text{et} \quad y(t_0) = y_0 \ \ \text{(condition initiale)}$$

où y_0 est donné.

Tout comme dans la section "Intégration", on considère une subdivision régulière $(t_i)_{0 \leqslant i \leqslant n}$ de l'intervalle I de sorte que :

$$t_i = t_0 + ih$$

où h est le **pas** de cette subdivision.

En intégrant l'équation du système différentiel sur l'intervalle $[t_{i-1}, t_i]$ (en supposant bien sûr que $i \geqslant 1$), nous pouvons écrire :

$$y(t_i) - y(t_{i-1}) = \int_{t_{i-1}}^{t_i} y'(t)\,\mathrm{d}t = \int_{t_{i-1}}^{t_i} f(t, y(t))\,\mathrm{d}t$$

Dans la méthode d'Euler explicite, on calcule l'intégrale de l'équation de façon approchée en remplaçant $f(t, y(t))$ par sa valeur sur la borne inférieure de l'intégrale, c'est à dire en t_{i-1}. On obtient alors :

$$y(t_i) - y(t_{i-1}) = f(t_{i-1}, y(t_{i-1})) \times (t_i - t_{i-1}) = f(t_{i-1}, y(t_{i-1})) \times h$$

On transforme donc le système différentiel en une équation de récurrence en posant $y_i = y(t_i)$ et $y_0 = y(t_0)$:

$$\boxed{y_i = y_{i-1} + f(t_{i-1}, y_{i-1}) \times h}$$

Mise en oeuvre : la fonction `euler1D` prend en paramètre la fonction `f` caractéristique du système différentiel (à définir dans le fichier du programme) et renvoie deux listes :

1. La liste `Dates = [t0, t1, ..., tn]` où `t`$_F$ `- h` $<$ `tn` $\leqslant$ `t`$_F$.
2. La liste des valeurs approchées `Y = [y0, y1, ..., yn]` du système différentiel aux dates t_0, t_1, ...,t_n.

```
1  def euler1D(f, y0, t0, tf, h) :
2    Dates, Y = [t0], [y0]
3    t, y = t0, y0        # Initialisation de la récurrence
4    while t + h <= tf :
5      y = y + f(t, y) * h
6      Y.append(y)
7      t = t + h
8      Dates.append(t)
9    return Dates, Y
```

2) Méthode d'Euler 2D

Nous allons développer l'exemple suivant :

$$y''(t) + y'(t)\sin[\,y(t)\,] = t$$

où $y : t \longrightarrow y(t)$ est une application définie sur $[t_0, t_f]$. Nous recherchons donc la solution de cette équation différentielle vérifiant les conditions initiales $y(t_0) = y_0$ et $y'(t_0) = v_0$ données. Par la suite

nous allons poser $v = y'$ (fonction vitesse), ce qui permet de définir le système différentiel suivant :

$$\left\{ \begin{array}{lcl} y'(t) & = & v(t) \\ v'(t) & = & t - v(t)\sin[\,y(t)\,] = f(t, y(t), v(t)) \end{array} \right. \qquad (*)$$

On introduit à nouveau une subdivision régulière $(t_i)_{0 \leqslant i \leqslant n}$ de l'intervalle $[t_0, t_f]$, avec un pas de temps h, de sorte que :

$$t_i = t_0 + ih$$

En intégrant le système différentiel $(*)$ sur l'intervalle $[t_{i-1}, t_i]$ nous pouvons écrire :

$$y(t_i) - y(t_{i-1}) = \int_{t_{i-1}}^{t_i} v(t)\,\mathrm{d}t \quad \text{et} \quad v(t_i) - v(t_{i-1}) = \int_{t_{i-1}}^{t_i} f(t, y(t), v(t))\,\mathrm{d}t$$

Tout comme dans la section (1), la méthode d'Euler explicite consiste à évaluer les deux intégrales de façon approchée en remplaçant les fonctions sous les intégrales par leur valeur sur les bornes inférieures. En posant $y_i = y(t_i)$ et $v_i = v(t_i)$ on obtient :

$$\boxed{y_i = y_{i-1} + v_{i-1} \times h \quad \text{et} \quad v_i = v_{i-1} + f(t_{i-1}, y_{i-1}, v_{i-1}) \times h}$$

Connaissant l'expression de $f : (t, y, v) \longmapsto f(t, y, v)$, les valeurs de y_i et v_i au rang i se calculent de proche en proche à l'aide d'une récurrence croisée donnée par les deux équations ci-dessus et compte-tenu des conditions initiales $y(t_0) = y_0$ et $v(t_0) = v_0$.

```python
from math import sin

def f(t,y,v) :
    return t - v * sin(y)

def euler2D(f,y0,v0,t0,tf,h) :
    Dates,Y,V = [t0], [y0], [v0]
    t,y,v = t0,y0,v0
    while t + h <= tf :
        y, v = y + v*h, v + f(t,y,v)*h
        t = t + h
        Dates.append(t)
        Y.append(y)
        V.append(v)
    return Dates, Y, V
```

V. Algorithmes de tri

Les algorithmes de tris sont fondamentaux. On dispose d'une liste L de n éléments qui peuvent être des entiers, des réels ou, plus généralement, toute collection de valeurs sur lesquelles on peut définir une relation d'ordre totale.

Entrée : la liste `L`
Sortie : une liste contenant les mêmes éléments que `L` mais ordonnés du plus petit au plus grand (ou, alternativement,T du plus grand au plus petit).

Définitions

On dit que le tri est :

- **en place** si et seulement s'il lit et écrit directement dans la liste `L`. Ce type de tri utilise peu de mémoire supplémentaire.
- **stable** si et seulement si lorsqu'il rencontre des éléments égaux dans la liste `L`, il les restitue dans le même ordre dans la liste triée.
- **interne** (resp. **externe**) si et seulement si les données à trier ne sont chargées que dans la mémoire vive de l'ordinateur (resp. la mémoire de masse, disque dur par exemple, est utilisée pour stocker les donnée lorsqu'elle sont trop volumineuses).

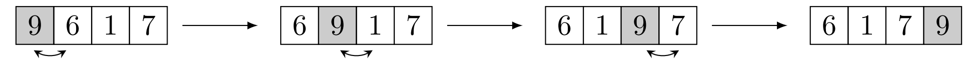
1) Le tri bulle

La liste `L` est parcourue à partir de son premier élément. Chaque élément est comparé à son successeur et, s'il est plus grand, on les échange. On continue jusqu'à atteindre la fin de la liste. À la fin de cette étape, le plus grand élément de la liste se trouve donc en dernière position.

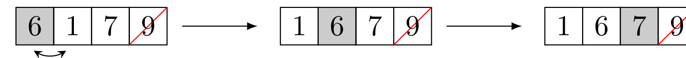
On recommence tout le processus précédent avec la liste `L[0:n-1]` ne comprenant que les $n-1$ premiers éléments, ..., et ainsi de suite.

Exemple avec `L = [9,6,1,4]`

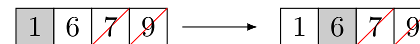
Étape 1 :



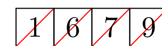
Étape 2 :



Étape 3 :



Sortie :



Son implémentation est simple et fait appel à deux boucles imbriquées.

```
  def tri_bulle(L) :
1   n = len(L)
2   for i in range(n-1) : # n-1 étapes
3     for j in range(n-1-i) :
4       if L[j+1] < L[j] :
5         L[j], L[j+1] = L[j+1], L[j] # échange éléments
```

Ce tri est en place et il est stable (ce dernier point étant dû à l'inégalité stricte dans `if L[j+1] < L[j]`).

Complexité temporelle :

L'instruction ligne 1 est $O(n)$. En supposant que les lignes 4 et 5 s'exécutent à temps constant c, la complexité temporelle des deux

boucles for est :

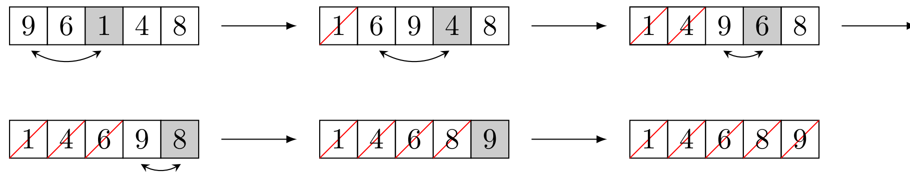
$$\sum_{i=0}^{n-2}\sum_{j=0}^{n-2-i} c = c \times \sum_{i=0}^{n-2}(n-1-i) = c \times \left\{(n-1)^2 - \frac{(n-1)(n-2)}{2}\right\}$$
$$\underset{\infty}{\sim} \frac{c}{2}n^2 = O(n^2)$$

En conclusion, la complexité de ce tri est $O(n^2)$.

2) Le tri par sélection

On recherche le plus petit élément dans une liste `L` et on le place en première position. On recherche ensuite le plus petit élément de la liste `L` privée du premier élément et on le place en position suivante ... et ainsi de suite jusqu'au dernier élément.

Exemple de fonctionnement avec la liste `L = [9,6,1,4,8]` :



```
def tri_selection(L) :
    n = len(L)
    for i in range(n-1) :
        indice_mini = i
        for j in range(i+1,n) :
            if L[indice_mini] > L[j] :
                indice_mini = j
        L[indice_mini], L[i] = L[i], L[indice_mini]
```

Ce tri est en place et il est stable.

3) Tri par insertion

Précondition : $n > 1$ # Au moins 2 éléments dans la liste

```
1  def TRI_INSERTION(L) :
2      n = len(L)
3      for i in range(1,n) :
4          temp = L[i]
5          j = i        # j est l'indice du trou dans la liste
6          while ( j > 0 and L[j - 1] > temp ) :
7              L[j] = L[j - 1]
8              j = j - 1
9          L[j] = temp
```

Intérêt : Le tri se fait sur place c'est à dire à l'intérieur même de la liste. Pas besoin de réserver un espace mémoire supplémentaire.

4) Tri par comptage

Le tri par comptage prend en argument une liste `L` d'entiers à trier, dont on connaît un minorant m et un majorant M. L'idée est de compter le nombre d'occurence dans `L` de chaque entier compris entre m et M, puis de construire une seconde liste `L2` de même taille que `L` : le plus petit élément de `L` est placé dans `L2` autant de fois qu'il apparaît dans `L`, et ainsi de suite.

Le comptage se fait grâce à un dictionnaire dont les clés sont les éléments `e` de `L` et dont les valeurs sont le nombre de fois que `e` est présent dans `L`.

On utilise une fonction auxiliaire `comptage(L:list)` $\to$ `dict` qui renvoie le dictionnaire.

```
   def comptage(L) :
1    dico = {}
2    for e in range(L) :
3      if e in dico.keys() :
4        dico[e] += 1
5      else :
6        dico[e] = 1
7    return dico

   def tri_comptage(L,m,M) :
1    D = comptage(L)
2    L2 = []
3    for x in range(m,M+1) :
4      if x in D.keys() :
5        for i in range(D[x]) :
6          L2.append(x)
7    return L2
```

Ce tri n'est pas en place puisqu'il nécessite la création d'un dictionnaire qui occupe une place supplémentaire non négligeable dans la mémoire. Par contre il est stable.

5) Tri fusion

Le tri fusion est un exemple du paradigme diviser pour régner (on devrait mieux dire *diviser pour résoudre* dans le cas qui nous occupe). Cela consiste à prendre un problème et à le diviser en deux sous - problèmes plus simples à résoudre. On utilise ensuite les deux solutions pour résoudre le problème initial.

Supposons que nous disposions de deux listes `L1` et `L2`, de tailles respectives `n1` et `n2`, chacune triée par valeurs croissantes, c'est à dire que les éléments de `L1` sont triés les uns par rapport aux autres et ceux de `L2` aussi.

On souhaite construire une liste `L` de taille `n1 + n2` dont tous les éléments soient triés par ordre croissant. C'est le travail réalisé par la fonction `fusion` écrite ci-dessous :

Fusion de deux listes triées séparément

```
    def fusion(L1,L2) :
1     n1,n2 = len(L1),len(L2)
2     if n1 == 0 :
3       return L2
4     elif n2 == 0 :
5       return L1
6     else :
7       L = []
8       i1,i2 = 0,0
9       while i1 < n1 and i2 < n2 :
10        if L1[i1] <= L2[i2] :
11          L.append(L1[i1])
12          i1 += 1
13        else :
14          L.append(L2[i2])
15          i2 += 1
    # à ce stade soit i1 == n1, soit i2 == n2
16      if i1 < n1 :
17        L = L + L1[i1:]
18      if i2 < n2 :
19        L = L + L2[i2:]
20      return L
```

L'algorithme du tri fusion est ensuite écrit sous la forme d'une

fonction récursive.

```
   def tri_fusion(L) :
1    n = len(L)
2    if n <= 1 : # si liste de 0 ou 1 élément ...
3       return L # pas la peine de trier
4    else :
5       m = n//2
6       L1 = tri_fusion(L[0:m])
7       L2 = tri_fusion(L[m:])
8       return fusion(L1,L2)
```

Complexité temporelle :

On peut montrer que la complexité temporelle $T(n)$ de ce tri est $O(n\lg(n))$ en moyenne.

6) Tri rapide (quick sort)

Le tri rapide est très souvent utilisé, bien que ce ne soit pas le meilleur en terme de complexité temporelle. Comme le tri par fusion, il fonctionne lui aussi sur le principe diviser pour régner (résoudre) et sur une approche récursive.

Algorithme `tri_rapide(L)`

- Si `L` ne contient que 0 ou 1 élément (elle est déjà triée) : renvoyer `L`. Ceci est la condition de terminaison de la récursivité.
- Sinon :
  - choisir dans `L` un élément particulier `p` appelé *pivot* (à priori on peut prendre n'importe quel élément de `L`) et le retirer de `L` ;
  - créer deux listes `Lg` et `Ld`, parcourir tous les éléments de `L` et placer ceux qui sont inférieurs à `p` dans `Lg` et ceux qui sont qui sont strictement supérieurs à `p` dans `Ld` ;
  - renvoyer `tri_rapide(Lg) + [p] + tri_rapide(Ld)`.

En voici une version qui prend pour pivot le dernier élément de la liste.

```
   def tri_rapide(L) :
1    if len(L) <= 1 :
2       return L    # terminaison de la récursivité
3    else :
4       p = L.pop()    # retirer le pivot de la liste
5       Lg,Ld = [],[]    # contruction de Lg et Ld
6       for e in L :
7          if e <= p :
8             Lg.append(e)
9          else :
10            Ld.append(e)
11      return tri_rapide(Lg) + [p] + tri_rapide(Ld)
```

Tel qu'il est présenté ici, ce tri n'est pas en place (la construction des deux listes `Lg` et `Ld` occupe un espace mémoire supplémentaire significatif) mais il est stable.

Complexité temporelle :

On peut montrer que la complexité temporelle $T(n)$ de cette fonction est $O(n^2)$ dans le pire des cas et $O(n\lg(n))$ dans le meilleur des cas.