Culture algorithmique

4.1 Algorithmes dichotomiques

4.1.1 Introduction

Les méthodes de résolutions par un algorithme dichotomique font partie des algorithmes basés sur le principe de « diviser pour régner ». Elles utilisent la définition du terme **dichotomie** qui signifie diviser un tout en deux parties « opposées ». Certains algorithmes de tris sont basés sur ce principe de diviser pour régner.

Ce cours vous présente deux algorithmes dichotomiques :

- la recherche d'un élément dans une liste triée ;
- la détermination de la racine d'une fonction quand elle existe.

4.1.2 Recherche dichotomique dans une liste triée

Lorsque vous cherchez le mot « hippocampe » dans le dictionnaire, vous ne vous amusez pas à parcourir chaque page depuis la lettre a jusqu'à tomber sur le mot « hippocampe »...
Dans une liste triée, il y a plus efficace ! Par exemple dans le dictionnaire, vous ouvrez à peu près au milieu, et suivant si le mot trouvé est « inférieur » ou « supérieur » à « hippocampe » (pour l'ordre alphabétique), vous poursuivez votre recherche dans l'une ou l'autre moitié du dictionnaire.

Propriété –

On se donne une liste `L` de nombres de longueur `n`, triée dans l'ordre croissant, et un nombre `x0`.
Pour chercher `x0`, on va couper la liste en deux moitiés et chercher dans la moitié intéressante et ainsi de suite.
On appelle `g` l'indice de l'élément du début de la sous-liste dans laquelle on travaille et `d` l'indice de l'élément de fin.
Au début, `g = 0` et `d = n-1`
On souhaite construire un algorithme admettant l'invariant suivant :

si `x0` est dans `L` alors `x0` est dans la sous-liste `L[g:d]` (`g` inclus et `d` exclu).

On va utiliser la méthode suivante.

- On compare `x0` à « l'élément du milieu » : c'est `L[m]` où `m = (g+d)//2` son indice est `m = n//2` (division euclidienne)
- Si `x0 = L[m]`, on a trouvé `x0`, on peut alors s'arrêter.
- Si `x0 < L[m]`, c'est qu'il faut chercher dans la première moitié de la liste, entre `L[g]` et `L[m-1]` (`L[m]` exclu).
- Si `x0 > L[m]`, c'est qu'il faut chercher dans la seconde moitié de la liste, entre `L[m+1]` et `L[d]` (`L[m]` exclu).

On poursuit jusqu'à ce qu'on a trouvé `x0` ou lorsque l'on a épuisé la liste `L`.

Exemples d'application

Indiquer pour les deux exemples suivants les valeurs successives de `g` et `d` :

1. `x0` = 5 et `L` = | −3 | 5 | 7 | 10 | 11 | 14 | 17 | 21 | 30 |
2. `x0` = 11 et `L` = | −2 | 1 | 2 | 7 | 8 | 10 | 13 | 16 | 17 |

Cas 1 : $\begin{cases} g = 0 \\ d = 8 \\ m = 4, L[m] > x0 \end{cases}$

$\begin{cases} g = 0 \\ d = 3 \\ m = 1, L[m] = x0 \end{cases}$.

C'est fini, on a bien trouvé x_0 dans la liste.

Cas 2 : $\begin{cases} g = 0 \\ d = 8 \\ m = 4, L[m] < x0 \end{cases}$,

$\begin{cases} g = 5 \\ d = 8 \\ m = 6, L[m] > x0 \end{cases}$

$\begin{cases} g = 5 \\ d = 5 \\ m = 5, L[m] < x0 \end{cases}$ $\begin{cases} g = 6 \\ d = 5 \end{cases}$.

C'est fini, on a épuisé la liste `L` et on n'a pas trouvé $x0$.

Implémentation en Python

La fonction `recherche_dichotomie` d'arguments une liste `L` et un élément `x` renvoyant un booléen disant si `x` est dans la liste `L` est proposée :

```
def recherche_dichotomie(L:list, x:int)-> bool:
    n = len(L)
    g = 0 # c'est l'indice de gauche
    d = n - 1 # c'est l'indice de droite
    rep = False
    while g <= d and rep == False :
        # si x est dans L alors L[g] <= x <= L[d]     {invariant}
        m = (g+d) // 2
        if x == L[m]:
            rep = True
        elif x < L[m]:
            d = m - 1
        else:
            g = m + 1
        # si x est dans L alors L[g] <= x <= L[d]     {invariant}
    return(rep)
```

Remarque : La terminaison de l'algorithme est obtenue avec $d - g$ qui est un entier positif qui décroît strictement à chaque passage dans la boucle `while` et joue le rôle de variant.

Implémentation récursive

```
def recherche_dichotomique_recursive(L:[int], x0:int, g:int, d:int) -> bool:
    if g>d :
        return False # ou -1
    m = (g+d)//2
    if L[m] == x0 :
        return True # ou m
    elif L[m] > x0 :
        return recherche_dichotomique_recursive(L, x0, g, d-1)
    else :
        return recherche_dichotomique_recursive(L, x0, g+1, d)
```

Terminaison (Implémentation non récursive)

Prenons comme variant de boucle la longueur de la zone de recherche, à savoir, à la n^e itération, $\ell_n = d_n - g_n$.

À l'itération suivante, $m_{n+1} = \left\lfloor \frac{g_n + d_n}{2} \right\rfloor$

- ou bien `L[m]==x0` et l'alogrithme s'arrête ;

- ou bien $g_{n+1} = g_n$ et $d_{n+1} = m_{n+1} - 1$. Ainsi, $\ell_{n+1} = \left\lfloor \frac{g_n + d_n}{2} \right\rfloor - 1 - g_n$ et $\ell_{n+1} \leq \frac{g_n + d_n}{2} - 1 - g_n \leq \frac{g_n + d_n - 2 - 2g_n}{2} \leq \frac{\ell_n - 2}{2} < \ell_n$;
- ou bien $g_{n+1} = m_{n+1} + 1$ et $d_{n+1} = d_n$. Ainsi, $\ell_{n+1} = d_n - \left\lfloor \frac{g_n + d_n}{2} \right\rfloor - 1$ et $\ell_{n+1} \leq d_n - \frac{g_n + d_n}{2} - 1 \leq \frac{2d_n - g_n - d_n}{2} - 1 < \ell_n$.

La longueur ℓ_n est donc strictement décroissante. L'algorithme terminera donc.

Preuve de correction (Implémentation non récursive)

Montrons que `x` $\in$ `L` $\Rightarrow$ `x` $\in$ `[g:d+1]`.[1]

1: Cela signifie que $x \in [\![g, d]\!]$.

- En entrant dans la boucle, $g = 0$ et $d + 1 = n - 1 + 1 = n$. Si `x` $\in$ `L` alors, `x` $\in$ `[0:n]`. La propriété d'invariance est donc vérifiée.
- Considérons la propriété `x` $\in$ `L` $\Rightarrow$ `x` $\in$ `[g:d+1]` est vraie à la ^{e} itération.
- On calcule `m`.
- Si `x = L[m]`, on a bien `x` $\in$ `[g:d+1]`. La propriété d'invariance est vérifiée à la fin de l'itération et on sort de la boucle (car `rep` est `True`).
- Si `x < L[m]`, on a déjà vérifié que `x` était différent de `L[m]` et alors `d=m-1`. Donc $g = g$ et $d + 1 = m - 1 + 1 = m$. `x` $\in$ `L` alors `x` $\in$ `[g:m]`. La propriété d'invariance est donc vérifiée.
- Si `x > L[m]`, on a déjà vérifié que `x` était différent de `L[m]` et alors `g=m+1`. Donc $g = m + 1$ et $d + 1$ est inchangé. Donc si `x` $\in$ `L` alors `x` $\in$ `[m+1:d+1]`. La propriété d'invariance est donc vérifiée.

Dans chacun des cas, la propriété d'invariance est vérifiée à la fin de la ^{e} itération. C'est donc bien un invariant de boucle.

4.1.3 Analyse de la complexité

Dans le pire des cas, le terme recherché n'est pas dans la liste. On divise donc la taille du tableau par 2 tant que la taille est supérieure ou égale à 1.

On note n la taille du tableau. On cherche k le nombre de fois qu'on peut diviser la taille du tableau par 2 : $n \cdot \left(\frac{1}{2}\right)^k \geq 1 \Rightarrow \cdot \left(\frac{1}{2}\right)^k \geq \frac{1}{n} \Rightarrow k \ln\left(\frac{1}{2}\right) \geq \ln\left(\frac{1}{n}\right) \Rightarrow k \geq \frac{\ln\left(\frac{1}{n}\right)}{\ln\left(\frac{1}{2}\right)}$

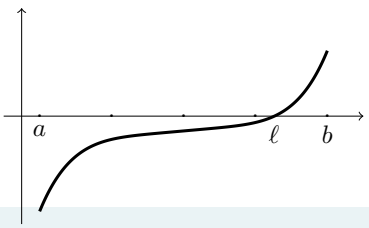
$\Rightarrow k \geq \frac{\ln n}{\ln 2}$. Il y aura donc dans le pire des cas $\log_2 n$ opérations. L'algorithme de recherche dichotomique est donc en $\mathcal{O}(\log_2(n))$.

4.1.4 Détermination de la racine d'une fonction par dichotomie

Principe théorique de la méthode par dichotomie

On considère une fonction f vérifiant :

$$f \text{ continue sur } [a, b] \text{ ; } f(a) \text{ et } f(b) \text{ de signes opposés.}$$



Le théorème des valeurs intermédiaires nous assure que f possède au moins un zéro ℓ entre a et b. La preuve, vue en cours de mathématiques, repose sur la méthode de dichotomie. Prenons le cas $f(a) < 0$ et $f(b) > 0$ et posons $g_0 = a$, $d_0 = b$.

On considère $m_0 = \dfrac{g_0 + d_0}{2}$ et on évalue $f(m_0)$:

- Si $f(m_0) \geq 0$, on va poursuivre la recherche d'un zéro dans l'intervalle $[g_0, m_0]$
  On pose donc : $g_1 = g_0 \quad ; \quad d_1 = m_0$
- Sinon, la recherche doit se poursuivre dans l'intervalle $[m_0, d_0]$
  On pose donc : $g_1 = m_0 \quad ; \quad d_1 = d_0$
- On recommence alors en considérant $m_1 = \dfrac{g_1 + d_1}{2}$...

Implémentation en Python et avec scipy

Écrivons une fonction `zero_dichotomie(f:callable, a:float, b:float, epsilon:float) -> float` d'arguments une fonction `f`, des flottants `a` et `b` (tels que `a < b`), et la précision voulue `epsilon` (flottant strictement positif). Cette fonction renverra une valeur approchée à `epsilon` près d'un zéro de `f`, compris entre `a` et `b`, obtenue par la méthode de dichotomie.

```
def zero_dichotomie(f:callable, a:float, b:float, epsilon:float) -> float:
    g = a # c'est un flottant
    d = b # c'est un flottant
    while d-g > 2*epsilon :
        m = (g + d) / 2
        if f(g)*f(m) <= 0:
            d = m
        else:
            g = m
    return ((g + d)/2)
```

Effectuons un test avec la fonction $f : x \mapsto x^2 - 2$ sur l'intervalle $[1, 2]$, avec une précision de 10^{-6} :

```
def f(x):
     return(x ** 2 - 2)
print (zero dichotomie(f, 1, 2, 10**(-6)))

# il s'affichera : 1.4142141342163086
```

Une telle fonction est déjà prédéfinie dans la bibliothèque `scipy.optimize`, la fonction `bisect`[2] :

2: La méthode de dichotomie s'appelle aussi la méthode de la *bisection*.

```
import scipy.optimize as spo
print (spo.bisect(f, 1, 2))
# il s'affichera : 1.4142135623724243
```

La précision est un argument optionnel (à mettre après `f`, `a` et `b`) et vaut 10^{-12} par défaut.

4.2 Intégration numérique

Hypothèse

$f : [a, b] \longrightarrow \mathbb{R}$ est une fonction continue sur $[a, b]$. On note $I = \int_a^b f(x)\mathrm{d}x$.

4.2.1 Principe des méthodes des rectangles

Définition –

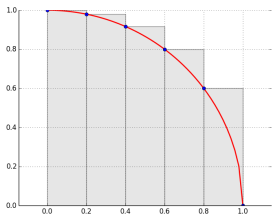
Dans cette méthode, la fonction à intégrer est interpolée par un polynôme de degré 0, à savoir une fonction constante. Géométriquement, l'aire sous la courbe est alors approximée par un rectangle. Plusieurs choix sont possibles.

Rectangles à gauche $I = \int_a^b f(x)\mathrm{d}x \simeq (b-a)\, f(a)$

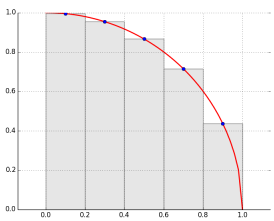
Point milieu $I = \int_a^b f(x)\mathrm{d}x \simeq (b-a)\, f\left(\frac{a+b}{2}\right)$

Rectangles à droite $I = \int_a^b f(x)\mathrm{d}x \simeq (b-a)\, f(b)$

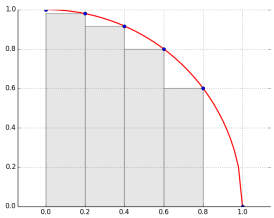
4.2.2 Interprétation graphique



Rectangles à gauche



Point milieu

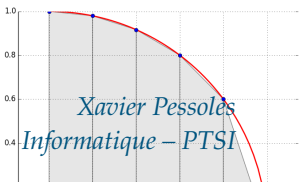


Rectangles à droite

4.2.3 Principe des méthodes des trapèzes

Définition –

Dans cette méthode, la fonction à intégrer est interpolée par un polynôme de degré 1, à savoir une fonction affine. Géométriquement, l'aire sous la courbe est alors approximée par un trapèze : $I = \int_a^b f(x)\mathrm{d}x \simeq (b-a)\,\frac{f(a)+f(b)}{2}$



Notion d'erreur d'intégration

Résultat –

Dans chaque cas, on intègre f sur n subdivisions régulières de I.
Erreur sur la méthode des rectangles à gauche et à droite
Soit f fonction dérivable sur $I = [a, b]$ et dont f' est continue sur I. Soit M_1 un majorant de f' sur I. L'erreur ε commise lors de l'intégration par la méthode des rectangles à droite ou à gauche est telle que $\varepsilon \leq \frac{M_1}{2n}$.
Erreur sur la méthode des rectangles – point milieu
Si de plus f est deux fois dérivables sur $I = [a, b]$ et f'' est continue sur I, on note M_2 un majorant de f'' sur I.L'erreur ε commise lors de l'intégration par la méthode des rectangles – point milieu est telle que $\varepsilon \leq \frac{M_2}{12n^2}$.
Erreur sur la méthode des trapèzes
L'erreur commiseε est telle qu'il existe un entier M tel que $\varepsilon \leq \frac{M}{12n^2}$.

Bibliothèque Python

Il est possible d'intégrer une fonction en utilisant les modules de la bibliothèque `scipy` :

```python
from scipy.integrate import quad
from math import sin
# Définition des bornes de gauche et de droite
g,d = -1,1
def f(x):
    return sin(x)

I,erreur = quad(f,g,d)
print(I,erreur)
```

4.2.4 Implémentation des algorithmes d'intégration

Méthode des rectangles à gauche

```python
def integrale_rectangles_gauche(f,a,b,nb):
    """
    Calcul de la valeur approchée de l'intégrale de f(x) entre a et b par la
    méthode des rectangles à gauche.
    Keywords arguments :
    f -- fonction à valeur dans IR
    a -- flt, borne inférieure de l'intervalle d'intégration
    b -- flt, borne supérieure de l'intervalle d'intégration
    nb -- int, nombre d'échantillons pour le calcul
    """
    res = 0
    pas = (b-a)/nb
    x = a
    while x<b:
        res = res + f(x)
        x = x + pas
    return res*pas
```

Méthode des rectangles à droite

```
def integrale_rectangles_droite(f,a,b,nb):
    """
    Calcul de la valeur approchée de l'intégrale de f(x) entre a et b par la
    méthode des rectangles à droite.
    Keywords arguments :
    f -- fonction à valeur dans IR
    a -- flt, borne inférieure de l'intervalle d'intégration
    b -- flt, borne supérieure de l'intervalle d'intégration
    nb -- int, nombre d'échantillons pour le calcul
    """
    res = 0
    pas = (b-a)/nb
    x = a+pas
    while x<=b:
        res = res + f(x)
        x = x + pas
    return res*pas
```

Méthode des rectangles – Point milieu

```
def integrale_rectangles_milieu(f,a,b,nb):
    """
    Calcul de la valeur approchée de l'intégrale de f(x) entre a et b par la m
    éthode du point milieu.
    Keywords arguments :
    f -- fonction à valeur dans IR
    a -- flt, borne inférieure de l'intervalle d'intégration
    b -- flt, borne supérieure de l'intervalle d'intégration
    nb -- int, nombre d'échantillons pour le calcul
    """
    res = 0
    pas = (b-a)/nb
    x = a+pas/2
    while x<b:
        res = res + f(x)
        x = x + pas
    return res*pas
```

Méthode des trapèzes pour le calcul approché d'une intégrale sur un segment

```
def integrale_trapeze(f,a,b,nb):
    """
    Calcul de la valeur approchée de l'intégrale de f(x) entre a et b par la m
    éthode des trapèzes.
    Keywords arguments :
    f -- fonction à valeur dans IR
    a -- flt, borne inférieure de l'intervalle d'intégration
    b -- flt, borne supérieure de l'intervalle d'intégration
    nb -- int, nombre d'échantillons pour le calcul
    """
    res = 0
    pas = (b-a)/nb
    x = a+pas
    while x<b:
```

```
14            res = res + f(x)
15            x = x + pas
16        res = pas*(res+(f(a)+f(b))/2)
17        return res
```

4.3 Tris

4.3.1 Introduction et objectifs

- algorithmique quadratique : tri par insertion, par sélection ;
- tri par partition-fusion ;
- tri par comptage.

Objectif

Un algorithme de tri est un algorithme permettant d'organiser une liste d'éléments selon un ordre fixé. On peut dire que les éléments à trier feront partie d'un ensemble E muni d'une relation d'ordre total noté $\leq$.
Les ensembles $\mathbb{N}$, $\mathbb{R}$... sont munis de l'ordre $\leq$.
L'ensemble des chaînes de caractères peut être muni de l'ordre lexicographique (ordre du dictionnaire). Ainsi en Python, `'a'<'b'`, `'aa'<'b'`, `'A'<'a'` (les lettres majuscules sont avant les lettres minuscules).
Les principales capacités développées ici sont :

- comprendre un algorithme de tri et expliquer ce qu'il fait ;
- s'interroger sur l'efficacité algorithmique temporelle d'un algorithme ;
- programmer un algorithme dans un langage de programmation moderne et général.

Définition – Effet de bord

On dit qu'une fonction est à effet de bord lorsqu'elle modifie une variable en dehors de son environnement local. C'est par exemple le cas lorsqu'on donne une liste (objet mutable, passé par référénce) comme argument d'une fonction.
Conséquence sur les algorithmes de tri : une fonction de tri ne renvoie rien la plupart du temps (on peut donc l'appeler une procédure). La liste passée en argument en entrée sera triée et cela, même en dehors du scope de la fonction.

Définition – Tri en place

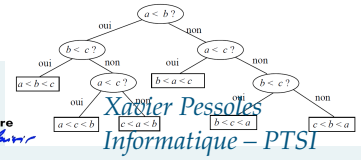
Un tri est effectué en place lorsque la liste à trier est modifiée jusqu'à devenir triée. Dans le cas contraire, la fonction de tri pourra renvoyer une novelle liste contenent les mêmes éléments, mais triés.

Définition – Tri stable

Un algorithme est dit stable si les positions relatives de deux éléments égaux ne sont pas modifiées par l'algorithme : c'est-à-dire que si deux éléments x et y égaux se trouvent aux positions i_x et i_y de la liste avant l'algorithme, avec $i_x < i_y$, alors c'est également le cas de leurs positions après l'algorithme.

Définition – Tri comparatif

Un tri est dit comparatif lorsqu'il s'appuie uniquement sur la comparaison deux à deux des éléments de la liste et pas sur la valeur ce ces éléments.



Exemple –

Un **algorithme comparatif** (à opposer au non-comparatif comme le tri baquet ou par paquet) est basé sur des comparaisons successives entre les données pour déterminer la permutation correspondant à l'ordre croissant des données. On va chercher ici à évaluer la complexité théorique des tris comparatifs en se basant sur le nombre de comparaisons.
Cet algorithme correspond au fonctionnement du tri insertion que nous verrons plus loin.

4.3.2 Tri par insertion

Principe

Définition – Principe du tri par insertion

Le **tri par insertion** est le tri que l'on effectue naturellement, par exemple pour trier un jeu de cartes. On trie les premières puis à chaque nouvelle carte on l'ajoute à l'ensemble déjà trié à la bonne place.
Ce tri s'effectue en place, c'est-à-dire qu'il ne demande pas d'autre tableau que celui que l'on trie.
Son coût en mémoire est donc constant si on ne compte pas la place occupée par les données.

Implémentation

- On stocke les données dans un tableau L entre les indices 0 et $n-1$.
- On utilise deux variables notées i,j et une variable `clef` (la clef) du même type que les données de la liste.
- On réalise une première boucle qui va permettre de parcourir tous les éléments $L[i]$ du tableau à trier. Chacun de ces éléments sera alors successivement la clef.
- La deuxième boucle (`while`) va nous permettre pour chaque valeur de la clef de tester successivement les valeurs déjà triées jusqu'à trouver une valeur inférieure. On aura ainsi déterminé la place de la clef.
- A chaque fois qu'une valeur est plus grande que la clef on la décale vers la droite pour laisser la place d'écrire la clef. Dès que l'on arrive sur une valeur inférieure on a trouvé la bonne place pour écrire la clef.

```
def tri_insertion(L:list) -> list:
    """
    Tri par insertion d'une liste d'entiers L.
    """
    n = len(L)
    for i in range(1,n):
        # Invariant : en entrée de boucle L[0:i-1] est triée.
        j = i
        v = T[i]
        while j>0 and v<L[j-1] :
            L[j] = L[j-1]
            j = j-1
        L[j] = v
        # Invariant : en fin de boucle L[0:i] est triée.
    return L
```

Terminaison

Pour la boucle `while`, la quantité `j` est un variant de boucle. En effet :

- avant d'entrer dans la boucle, `j` est une quantité entière strictement positive ;
- `j` est décrémenté de 1 à chaque itération, donc `j` décroit strictement à chaque itération.
- `j` est donc un variant de boucle.

La boucle `for` termine au bout de `n-1` itérations. L'algorithme de tri par insertion termine donc.

Correction

Montrons que la propriété `L[0:i] est triée` est un invariant de boucle.

Initialisation En entrant dans la boucle : `i=1` ; donc `L[0:1]` ne contient qu'un élément. `L[0:1]` est triée.

Hypothèse : considérons qu'au début de l'itération `i`, `L[1:i-1]` est triée.

Montrons qu'à la fin de l'itération `i`, `L[1:i]` est triée.

- `v = L[i]` est la valeur à insérer dans `L[0:i-1]`.
- **ou bien** `v>=L[i-1]`, on n'entre pas dans la boucle `while` et `L[i]=v` ; donc `L[0:i]` est triée.
- **ou bien** `v<L[i-1]` et on entre dans la boucle `while`.`j` décrémente alors et tous les éléments sont « décalés vers la droite » jusqu'à ce que `v>=L[j-1]`. On insère alors `v` en `L[j]`.

À la fin du `n-1` tour de boucle, `L[0:n]` est donc triée.

Complexité

Propriété – Complexité

La complexité est dans le pire des cas quadratique : $C(n) = \mathcal{O}(n^2)$.
La complexité est dans le meilleur des cas linéaire : $C(n) = \mathcal{O}(n)$.
L'efficacité du tri par insertion est excellente lorsque le tableau est déjà trié ou « presque trié » ($C(n) = \mathcal{O}(n)$).
Il surpasse alors toutes les autres méthodes de tri qui sont au mieux en $\mathcal{O}(n \times \ln(n))$.

Au vu de notre algorithme, le pire des cas que l'on peut rencontrer est celui où la liste `L` est triée dans l'ordre décroissant. En effet, dans ce cas, à l'itération `i`, la boucle while devra être réalisée `i` fois.

Comptons le nombre de comparaisons `j>0` réalisées par l'algorithme pour une liste de `n` éléments :

- à l'itération 1 : il y a 2 comparaisons ;
- à l'itération 2 : il y a 3 comparaisons ;
- ...
- à l'itération `n-1` : il y a `n` comparaisons.

Au final le nombre de comparaisons est égal à $C(n) = 2 + 3 + ... + n = \sum_{i=1}^{n}(i) - 1 = \frac{1}{2}n(n+1) - 1 = \mathcal{O}(n^2)$.

4.3.3 Tri rapide (ou « quicksort »)

Principe

Définition – Tri rapide

L'algorithme de tri rapide fait partie de la catégorie des algorithmes « diviser pour régner ».
À chaque appel de la fonction tri on choisit une valeur « **pivot** », par exemple le premier élément. On effectue une partition des élément à trier. Un premier groupe est constitué de valeurs inférieures au pivot et un deuxième avec les valeurs supérieures. Le pivot est alors placé définitivement dans le tableau.
On traite alors chacun des groupes de façon indépendante. On peut les traiter avec le même algorithme.

Implémentation

Pour trier une liste, on procède ainsi :

- si la liste est vide, on renvoie la liste vide ;
- sinon :
  - on choisit un « pivot », à savoir une valeur de la liste. Dans notre cas, nous prendrons le dernier élément,
  - on crée une liste `el_inf` constituée des éléments strictements inférieurs au pivot,
  - on crée une liste `el_sup` constituée des éléments supérieurs ou égaux au pivot (en excluant la dernière valeur du tableau, ie le pivot),
  - on renvoie la concatanation du tri rapide appliqué à `el_inf`, du pivot ry du tri rapide appliqué à `el_sup`.

```
def elts_inf(L : list, p : int) -> list :
    """
    Fonction renvoyant la liste des éléments de L stictements inférieurs à  p.
    """
    res = []
    for e in L :
        if e<p :
            res.append(e)
    return res

def elts_sup(L : list, p : int) -> list :
    """
    Fonction renvoyant la liste des éléments de L supérieurs ou égaux à  p.
    """
    res = []
    for e in L :
        if e>=p :
            res.append(e)
    return res

def tri_rapide(L : list) -> list :
    print(L)
    if len(L) == 0 :
        return []
    else :
        p = L[-1]
```

```
27          ei = elts_inf(L,p)
28          es = elts_sup(L[:-1],p)
29          return tri_rapide(ei) + [p] + tri_rapide(es)
```

Terminaison

Prenons comme variant la longueur n de la liste à trier. Nous cherchons à trier des listes de tailles supérieure à n.

Initialement, `L` est de taille `n` supérieur ou égal à 1. `n` est un entier strictement positif.

Considérons qu'au début du ième appel de `tri_rapide` `Li` est de taille `ni` strictement positif.

`ei` est de taille strictement inférieure à `Li` car elle ne contient que les valeurs strictement inférieures au pivot.

`es` est de taille strictement inférieure à `Li` car elle ne contient que les valeurs supérieures ou égales au pivot, à l'exception du pivot lui-même.

L'appel à `tri_rapide` se fait alors sur des listes de tailles strictement inférieures à `Li`.

`n` est donc un variant de boucle et la fonction termine.

Correction

Montrons la propriété suivante : « si `L` une liste de taille `inférieure ou égale` à n alors `tri_rapide(L)` renvoie une liste triée ».

Initialisation La propriété de récurrence est vraie pour une liste vide et pour une liste de taille 1.

Hérédité Soit une liste de taille `n+1`. Le pivot est alors `L[-1]`. `ei` et `es` ont une taille inférieure ou égale à n. D'après la proprité de récurrence, `tri_rapide(ei)` et `tri_rapide(es)` trient donc `ei` et `es`.

De plus, `tri_rapide(ei) + [p] + tri_rapide(es)` renvoie une liste triée.

La propriété de récurrence est donc vraie pour une liste de taille `n+1`.

Complexité

Propriété – Efficacité de l'algorithme

- Cette méthode de tri est très efficace lorsque les données sont distinctes et non ordonnées. La complexité est alors globalement en $\mathcal{O}(n \ln(n))$. Lorsque le nombre de données devient petit (<15) lors des appels récursifs de la fonction de tri, on peut avantageusement le remplacer par un tri par insertion dont la complexité est linéaire lorsque les données sont triées ou presque. **[Beynet]**
- D'autre part si le tableau est déjà trié avec le code mis en place on tombe sur la complexité « dans le pire des cas ». Une solution simple consiste à ne pas choisir systématiquement le premier élément du segment comme pivot, mais plutôt un élément au hasard. On peut par exemple choisir la valeur de façon aléatoire. **[wack]**

Meilleur des cas : $n \log n$	Pire des cas n^2

Pour un tableau de longueur n :

- on crée une liste `ei` avec k éléments et une liste `es` avec $n-k-1$ éléments (le dernier élément étant le pivot) ;
- la création de chacune de ces listes fait appel à `n` comparaisons (boucles `for` qui itèrent sur chaque élément de `L`) ;
- on appelle alors récursivement la fonction `tri_rapide` avec les listes de tailles précitées.

Le nombre de comparaisons à l'itération n est donc approximativement de : $C(n) = 2n + C(k) + C(n-k-1)$.

Dans le pire des cas, $k = 0$ (ou $k = n-1$). En conséquences, $C(n) = 2n + C(0) + C(n-1)$. Pour une liste de taille 0 ou 1, l'algorithme de tri est réalisé en un tant constant $C(0) = C(1) = 1$.

On a donc $C(n) = 2n+1+C(n-1) \Leftrightarrow C(n) - C(n-1) = 2n+1$. De même à l'itération i, on a $C(i) - C(i-1) = 2i+1$.

Par sommation, on a alors : $\sum_{i=1}^{n}(C(i) - C(i-1)) = \sum_{i=1}^{n}(2i+1) \Rightarrow C(n) - C(0) = 2 \times \frac{1}{2}n(n+1) + n \Rightarrow C(n) = n^2 + 2n - 1 = \mathcal{O}(n^2)$.

4.3.4 Tri fusion – *Merge sort*

Principe

Cet algorithme fait aussi partie des algorithmes « diviser pour régner ».

Le principe consiste à couper le tableau de départ en deux. On trie chacun des groupes indépendamment. Puis on fusionne les deux groupes en utilisant le fait que chacun des groupe est déjà ordonné.

Il est possible pour réaliser l'ordonnancement de chacun des groupes d'utiliser à nouveau l'algorithme de tri de façon récursive.

4.3.5 Implémentation

Le tri fusion d'une liste se base sur le principe suivant :

1. on sépare la liste en 2 listes de longueurs quasi-égales (à un élément près) ;
2. on trie ces deux listes en utilisant le tri fusion (par un appel récursif) ;
3. à partir de deux listes triées, on les fusionne en une seule liste en conservant l'ordre croissant.

```
def separe(L: list) -> tuple[list,list]:
    return L[:len(L) // 2], L[len(L) // 2:]

def fusion(L1: list, L2: list) -> list:
    """
    Fusion de deux listes triées.
    """
    if not L1 or not L2: # si l'une des listes est vide (éventuellement les 2)
        return L1 or L2 # alors on renvoie l'autre (éventuellement vide aussi)
    else:
        a, b = L1[0], L2[0]
        if a < b : # sinon on compare leurs premiers éléments
            return [a] + fusion(L1[1:], L2) # on place le plus petit en tête
    et on fusionne le reste
```

```
        else:
            return [b] + fusion(L1, L2[1:])

def tri_fusion(L: list) -> list:
    if len(L) < 2: # cas d'arrêt
        return L
    L1, L2 = separe(L) # sinon on sépare
    return fusion(tri_fusion(L1), tri_fusion(L2)) # et on fusionne les sous-
    listes triées
```

Terminaison

Terminaison de la fusion Soit la suite définie par la somme successive des longueur de `L1` et `L2`.

À chaque itération on appelle `fusion` à des listes dont la taille d'une d'entre elle est diminuée de 1. La somme des longueurs est donc une suite d'entiers strictement décroissante.

La somme des longueurs est donc un vairant de boucle et la fonction termine.

Terminaison du tri

Le tri fusion termine si les listes ont 0 ou 1 éléments. On note (u_n) la suite définie par $u_0 = n$, taille de la liste à trier et u_i longueur de la plus grande des listes. À chaque étape, la taille des listes est divisé par 2 ; donc $u_{i+1} = \left\lfloor \frac{u_i + 1}{2} \right\rfloor$. En conséquence, $u_{i+1} < u_i$ tant que $u_i > 1$. La fonction `tri_fusion` termine donc.

Correction

Complexité

Propriété – Efficacité de l'algorithme

La complexité est $n \log(n)$ dans le meilleur des cas et dans le pire des cas.

Complexité du tri fusion – Petites remarques Quel est le coût de l'algorithme de fusion présenté précédemment ? En Python, concaténer deux listes nécessite de copier ces deux listes dans une nouvelle liste. Cette opération est donc en $\mathcal{O}(n)$.

Par ailleurs, la taille des listes à concaténer diminue de 1 à chaque itération. Il y aura donc n appels récursifs.

L'algorithme de fusion présenté est donc en $\mathcal{O}(n^2)$.

La séparation quant à elle est complexité $\mathcal{O}\log_2(n)$ (en effet, le nombre de fois que fois qu'on peut diviser notre liste de taille n en 2 listes de taille $n//2$ est à peu près de $\log(n)$.

L'algorithme proposé est donc en $\mathcal{O}\left(n^2 \log(n)\right)$... et pas en $n \log(n)$!

Il faudrait donc rechercher un algorithme de fusion de complexité linéaire...

```python
def fusion(L1, L2):
    L = []
    i, j = 0, 0
    while i < len(L1) and j < len(L2):
        if L1[i] < L2[j]:
            L.append(L1[i])
            i += 1
        else:
            L.append(L2[j])
            j += 1
    L += L1[i:]
    L += L2[j:]
    return L
```

4.3.6 Dans Python : sort et sorted

Les listes Python ont une méthode native `list.sort()` qui modifie les listes elles-mêmes et renvoie `None`.

4.3.7 Conclusion et méthodes dans Python

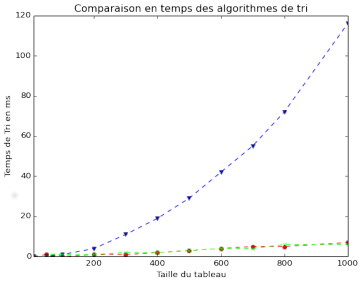
Propriété –

Distinguer par leurs complexités deux algorithmes résolvant un même problème : La première étape consiste à vérifier que les algorithmes résolvent exactement le même problème.

- On compare la complexité en temps dans le pire des cas (préférable a l'utilisation du module Time comme ci dessous car indépendant de la machine).
- Il faut vérifier que le pire des cas peut arriver en situation réelle
- Penser à comparer la complexité en espace qui peut permettre de départager des algorithmes équivalents

Exemple – Illustration de la comparaison

Pour comparer les temps globaux sur une même machine (Ici processeur Intel Core i7-4510U Haswell (2 GHz, TDP 15W),Mémoire vive 4 Go) on génère des tableaux de dimensions différentes avec des nombres aléatoires. On utilise le module time pour déterminer les temps. Ici c'est un temps global dépendant fortement de la machine utilisée. On fait varier la taille du tableau et on compare les temps mis par chacun des algorithmes (en rouge tri rapide, en bleu tri insertion, en vert tri fusion).



Bibliographie

[1] Irène Charon Olivier Hudry, Algorithmes de tri, cours de Télécom ParisTech, Avril 2014.

[2] B. Wack, S. Conchon, J. Courant, M. de Falco, G. Dowek, J.-C. Filliâtre, S. Gonnord, Informatique pour tous en classes préparatoires aux grandes écoles. Manuel d'algorithmique et programmation structurée avec Python. Nouveaux programmes 2013. Voies MP, PC, PSI, PT, TPC et TSI, Eyrolles, 2013.

[3] Beynet Patrick, Cours d'informatique publié sur le site de l'UPSTI.