Informatique (Semestre 1)

CM 6
Zéro d'une fonction, Tris simples

La Prépa, Grenoble INP, UGA

2023-2024

Sommaire

Sommaire

Résolution numérique d'équation : zéro d'une fonction

- f fonction continue sur $[a, b]$, $f(a) \leq 0$, $f(b) > 0$
- Un *zéro* de f est un x tel que $f(x) = 0$ (ici $x \in [a, b[$)
- Soit $\varepsilon > 0$, on cherche x tel que $f(x) \leq 0$ et $f(x + \varepsilon) > 0$

Solution 1 : Recherche linéaire

On part de a et on teste successivement $[a, a + \varepsilon[$, $[a + \varepsilon, a + 2\varepsilon[$, ..., $[a + i\varepsilon, a + (i + 1)\varepsilon[$.

```
def f(x):
    return math.sin(x) - 0.5

def zero_lineaire(a, b, epsilon):
    assert f(a) <= 0 and f(b) > 0
    i = 0
    while True:
        # invariant: f(a + i * epsilon) <= 0
        if ( a + (i+1) * epsilon >= b or
             f(a + (i+1) * epsilon) > 0 ):
            return a + i * epsilon
        i = i + 1
        # invariant: f(a + i * epsilon) <= 0
```

démo : zero.ipynb

Résolution numérique d'équation : preuve (terminaison)

Puisque $b > a$ et $\varepsilon > 0$, $\exists i \in \mathbb{N}$, $a + (i+1)\,\varepsilon \geq b$.
Donc la boucle termine.

- Dans le meilleur cas : 1 itération
- Dans le pire cas : $\lceil \frac{b-a}{\varepsilon} \rceil$ itérations

Soit $x \in \mathbb{R}$

- La partie entière par défaut, ou partie entière, de x est l'unique $n \in \mathbb{Z}$ tel que $n \leq x < n+1$. Notation : $n = E(x) = \lfloor x \rfloor$.
- La partie entière par excès de x est l'unique $n \in \mathbb{Z}$ tel que $n - 1 < x \leq n$. Notation : $n = \lceil x \rceil$.

Exemples :

- $\lfloor 23 \rfloor = \lceil 23 \rceil = 23$
- $\lfloor 2.5 \rfloor = 2$; $\lceil 2.5 \rceil = 3$
- $\lfloor -2.5 \rfloor = -3$; $\lceil -2.5 \rceil = -2$

Résolution numérique d'équation : preuve (correction)

Supposant l'invariant $f(a+i\varepsilon) \leq 0$ vrai au début de la boucle, il est toujours vrai à la fin la boucle, sinon on serait sorti de la boucle.

En sortie de boucle, on a donc :

- Soit $a+(i+1)\,\varepsilon \geq b$, donc $b-(a+i\varepsilon) \leq \varepsilon$, or $f(a+i\varepsilon) \leq 0$ (invariant) et $f(b) > 0$ (hypothèse initiale), donc $a+i\varepsilon$ est un zéro à ε près
- Soit $f(a+(i+1)\,\varepsilon) > 0$, or $f(a+i\varepsilon) \leq 0$ (invariant), donc $a+i\varepsilon$ est un zéro à ε près

On a donc une bonne solution dans tous les cas.

Note : On n'a pas imposé f croissante, donc il peut y avoir plusieurs solutions, et on n'est même pas sûr d'avoir la première, car f peut devenir positive puis négative de nouveau sur $]a+i\varepsilon, a+(i+1)\varepsilon[$.

Recherche dichotomique

Solution 2 : Recherche dichotomique

On peut aller beaucoup plus vite !

- Couper l'intervalle $[a, b[$ en $[a, c[$ et $[c, b[$, avec $c = \dfrac{a+b}{2}$.
- Regarder de quel côté on est sûr d'avoir une solution
- Recommencer.

```
def f(x):
    return math.sin(x) - 0.5
def zero_dichotomie(a, b, epsilon):
    while b - a > epsilon:
        pivot = (a + b) / 2
        value = f(pivot)
        if value <= 0:
            a = pivot
        else:
            b = pivot
    return a
```

démo : zero.ipynb

Recherche dichotomique : preuve

On note a_0 et b_0 les valeurs initiales de a et b.

Terminaison

Au début de la i-ième itération (en comptant à partir de $i = 1$), on a $b - a = \frac{b_0 - a_0}{2^{i-1}}$, donc quelque soit ε, il existe un i à partir duquel $b - a \leq \varepsilon$.

Correction

Invariant : $f(a) \leq 0$, $f(b) > 0$.

L'invariant est initialement vrai si les hypothèses sont vérifiées. S'il est vrai à l'itération i, alors le nouveau choix de a ou b maintient l'invariant.

En d'autres termes, il existe toujours un zéro de f entre a et b.

En fin de programme, $b - a \leq \varepsilon$ donc a est au plus à ε d'un zéro de f.

Recherche dichotomique : complexité

On obtient la complexité à partir de la preuve de terminaison : on cherche le premier i pour lequel

$$b - a = \frac{b_0 - a_0}{2^{i-1}} \leq \varepsilon$$

$$2^{i-1} \geq \frac{b_0 - a_0}{\varepsilon}$$

$$i = \left\lceil \log_2\left(\frac{b_0 - a_0}{\varepsilon}\right) + 1 \right\rceil = O\left(\log_2\left(\frac{b_0 - a_0}{\varepsilon}\right)\right)$$

Rappel : $\log_2$ est la fonction réciproque de $x \mapsto 2^x$.
$\log_2(1) = 0$, $\log_2(2) = 1$, $\log_2(4) = 2$, $\log_2(8) = 3$, $\log_2(16) = 4$, ...
Le nombre de bits nécessaires pour la représentation binaire de x est $\lfloor \log_2(x) \rfloor$.

Complexité : Meilleur cas = pire cas = $O\left(\log_2\left(\frac{b_0 - a_0}{\varepsilon}\right)\right)$

Recherche dichotomique : complexité (2)

En d'autres termes, on calcule 1 bit significatif par itération.
Pour calculer 15 décimales :

- un million de milliard d'itérations avec la recherche linéaire
- 50 itérations ($2^{50} > 10^{15}$) avec la recherche dichotomique.

Exercice

Que se passe-t-il si :

- $a = b$?
- $f(a) > 0$?
- $f(b) < 0$?
- ε est plus petit que les erreurs d'arrondis ?

Exercice

Que se passe-t-il si :

- $a = b$?
  On ne rentre pas dans la boucle while car $b - a \leq \varepsilon$, donc le résultat est a
- $f(a) > 0$?
- $f(b) < 0$?
- ε est plus petit que les erreurs d'arrondis ?

Exercice

Que se passe-t-il si :

- $a = b$?
- $f(a) > 0$?
  Le calcul renvoie a
- $f(b) < 0$?
- ε est plus petit que les erreurs d'arrondis ?

Exercice

Que se passe-t-il si :

- $a = b$?
- $f(a) > 0$?
- $f(b) < 0$?
  Le calcul renvoie un nombre entre $b - \varepsilon$ et b
- ε est plus petit que les erreurs d'arrondis ?

Exercice

Que se passe-t-il si :

- $a = b$?
- $f(a) > 0$?
- $f(b) < 0$?
- ε est plus petit que les erreurs d'arrondis ?
  On peut avoir une boucle infinie. Exemple : boucle infinie sur `zero_dichotomie(-1, math.pi, 1e-16)`

Sommaire

Comment trier ?

- Comment trier par ordre alphabétique une centaine de copies d'examen ?
- Comment trier une main de cartes au tarot ?

Tri simple

Tri par insertion :

- on prend le premier élément de la pile non triée
- on le met *à sa place* dans la pile triée
- jusqu'à ce que la pile non triée soit vide

Tri par sélection :

- on prend le plus petit (ou le plus grand) élément de la pile non triée
- on le met au-dessus de la pile triée
- jusqu'à ce que la pile non triée soit vide

Tri simple

Tri par insertion :

- on prend le premier élément de la **liste** non triée... *facile*
- on le met *à sa place* dans la **liste** triée... *trouver la bonne place* $\rightsquigarrow$ *boucle*
- jusqu'à ce que la **liste** non triée soit vide... $\rightsquigarrow$ *boucle principale*

Tri par sélection :

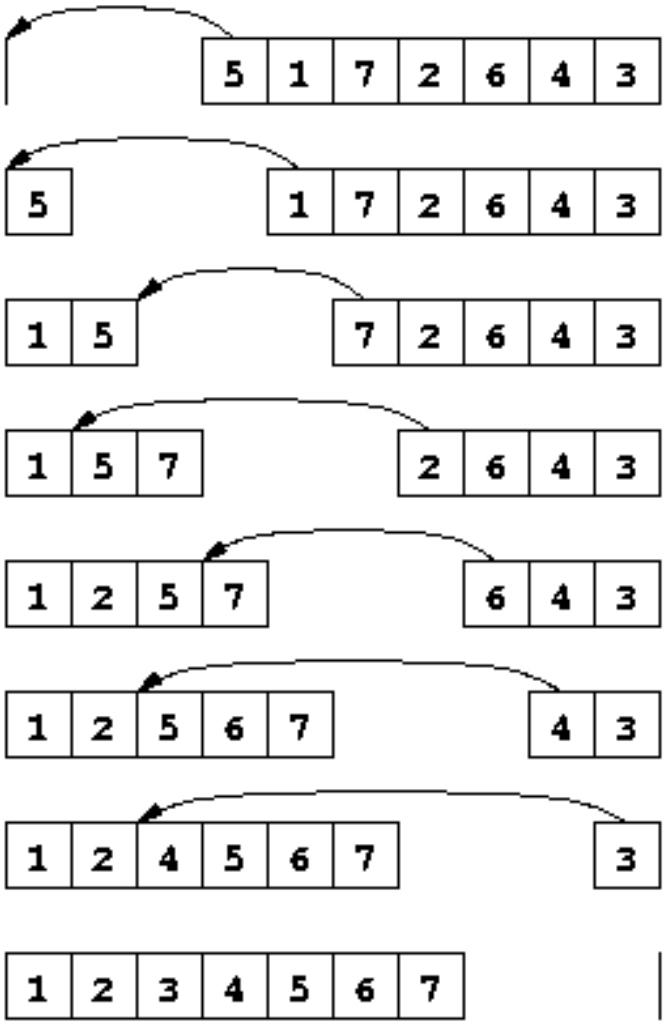
- on prend le plus petit (ou le plus grand) élément de la **liste** non triée... *trouver où il se trouve, le supprimer* $\rightsquigarrow$ *boucle*
- on le met au-dessus de la **liste** triée... *facile*
- jusqu'à ce que la **liste** non triée soit vide... $\rightsquigarrow$ *boucle principale*

D'autres idées d'algos de tri ?

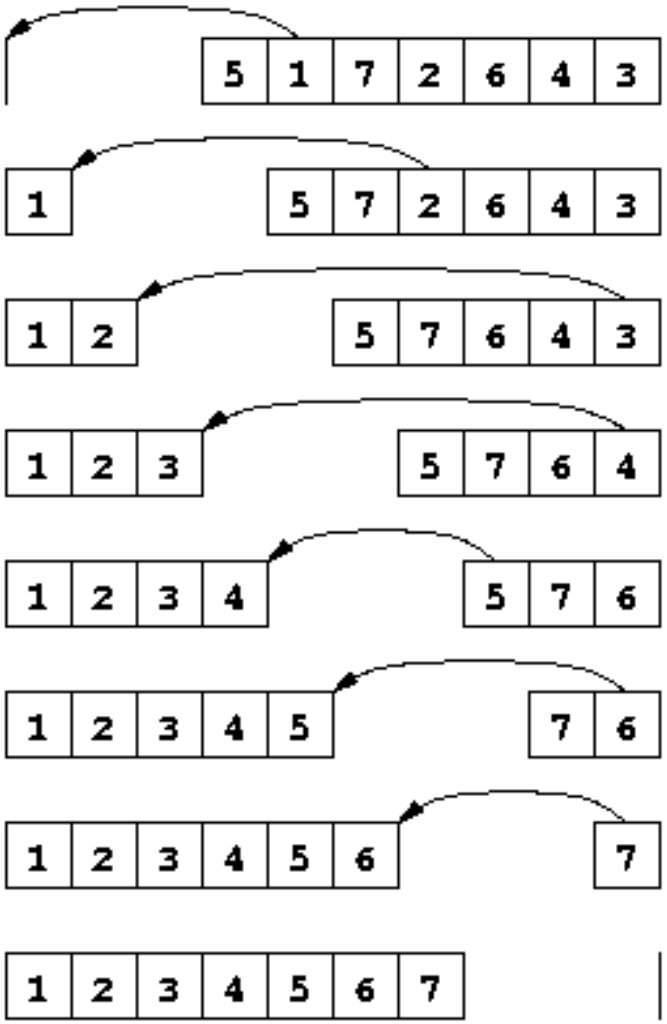
`https://visualgo.net/bn/sorting`
`https://imgur.com/gallery/voutF`
`https://www.toptal.com/developers/sorting-algorithms`

Tri simple

Tri par insertion :



Tri par sélection :



Tri par insertion « en place » §13.1

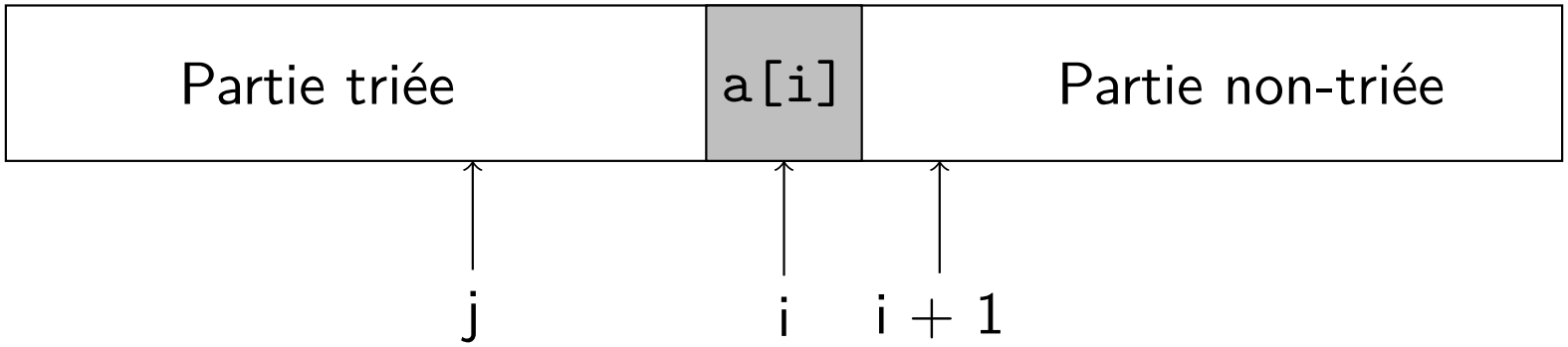
- On met la liste non triée et la liste triée dans une unique liste, dont la taille ne change pas
- À l'itération principale i, le tableau a est constitué de deux parties :
  - les éléments non triés à la fin du tableau (de l'index i à l'index $n-1$)
  - les éléments triés au début du tableau (de l'index 0 à l'index $i-1$)
- Itération principale : insérer l'élément `v = a[i]` à sa place dans la partie du tableau déjà triée `a[0...i-1]`
- On note qu'on commence en $i = 1$: le premier élément `a[0]`, pris seul, est déjà un tableau trié.

Par exemple, en partant d'un tableau non trié $a = [5, 2, 3, 1, 4]$:

- La partie triée est constituée du premier élément, $a = [\underline{5}, 2, 3, 1, 4]$
- On insère 2 dans $a[0..0] \rightarrow a = [\underline{2, 5}, 3, 1, 4]$
- On insère 3 dans $a[0..1] \rightarrow a = [\underline{2, 3, 5}, 1, 4]$
- On insère 1 dans $a[0..2] \rightarrow a = [\underline{1, 2, 3, 5}, 4]$
- On insère 4 dans $a[0..3] \rightarrow a = [\underline{1, 2, 3, 4, 5}]$

Tri par insertion « en place »

Au début de chaque itération, on a (invariant de boucle) :



La fonction ressemblera donc à (on commence en $i = 1$) :

```
def tri_insertion(a):
    for i in range(1,len(a)):
        # invariant:
        # les éléments 0 à i-1 du tableau sont triés
        # itération:
        # insérer l'élément i à sa place (indice j)
```

Tri par insertion « en place » : itération

Pour insérer l'élément *i* à sa place, il faut le comparer avec le précédent, et tant qu'il est inférieur au précédent (tri croissant) il faut l'échanger avec le précédent.

Mais il n'est pas nécessaire d'échanger. Il suffit de décaler l'élément vers la droite et de conserver la valeur à insérer, puis de la mettre à sa place finale une fois tous les éléments décalés.

```
# itération: insérer l'élément i à sa place
v = a[i] # la valeur à insérer
j = i
while 0 < j and v < a[j-1]:
    # décaler l'élément a[j-1] en j
    a[j] = a[j-1]
    j = j-1
# j contient l'index où mettre v
a[j] = v
```

démo : `tri_insertion.ipynb`

Tri par insertion « en place » : coût

L'opération "coûteuse" est l'insertion d'un élément à sa place, qui se fait en moyenne avec $i/2$ comparaisons et affectations, donc au total et en moyenne :

$$C(n) = \sum_{i=1}^{n-1} \frac{i}{2} = \frac{1}{2} \sum_{i=1}^{n-1} i = \frac{1}{2} \frac{(n-1)n}{2} = \frac{1}{4} n^2 - \frac{1}{4} n$$

Le terme dominant de C quand n est grand est en n^2 : $C(n) = O(n^2)$.
La complexité *quadratique*.

Tri par insertion « en place » : questions

1. Quelle est la complexité dans le pire des cas ? A quelle configuration de tableau correspond-elle ?
2. Quelle est la complexité dans le meilleur des cas ?
3. Ce tri fonctionne-t-il avec autre chose que des listes d'entiers ?

Tri par insertion « en place » : réponses

1. $n^2/2$ comparaisons et affectations. tableau trié à l'envers.
2. n comparaisons et affectations seulement - tableau trié
3. Oui, avec toute liste d'éléments qu'on peut affecter et comparer.

Remarque : il existe un algorithme bien plus efficace pour le tri de grands tableaux, le tri fusion, qui est en $O(n \log_2(n))$.

Sommaire

Tri par sélection « en place »

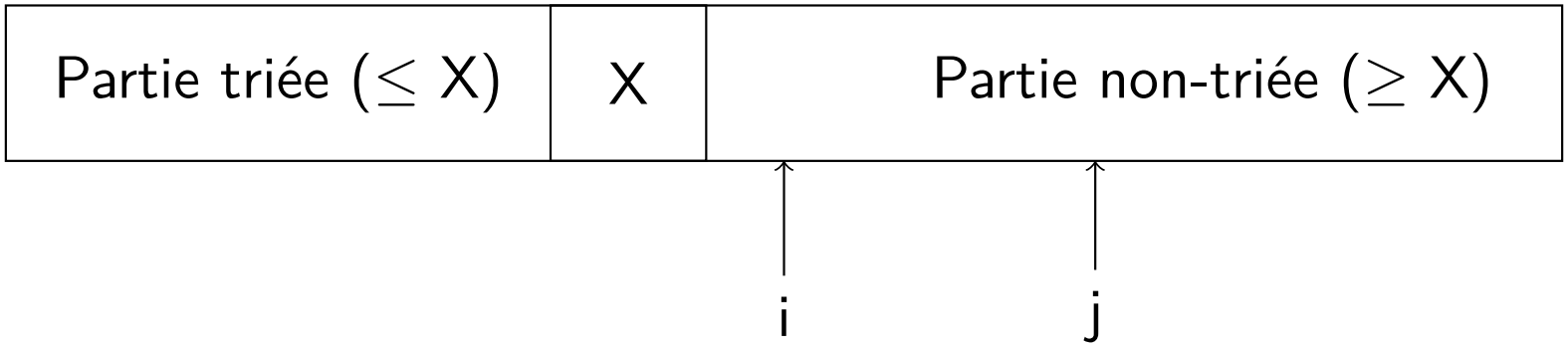
- On met la liste non triée et la liste triée dans une unique liste, dont la taille ne change pas
- À l'itération principale i, le tableau a est constitué de deux parties :
  - les éléments non triés à la fin du tableau (de l'index i à l'index $n-1$)
  - les éléments triés à leur place définitive au début (de l'index 0 à $i-1$)
- Itération principale : selectionner le plus petit élément $v = a[j]$ de la partie non triée, décaler vers la droite les éléments de $j-1$ à i (si $j-1 \geq i$), mettre l'élément v à la position i

Par exemple, en partant d'un tableau non trié $a = [5, 2, 3, 1, 4]$:

- On sélectionne $a[3] = 1$, et on le place à l'index 0. $a = [\underline{1}, 5, 2, 3, 4]$
- On sélectionne $a[2] = 2$, et on le place à l'index 1. $a = [\underline{1, 2}, 5, 3, 4]$
- On sélectionne $a[3] = 3$, et on le place à l'index 2. $a = [\underline{1, 2, 3}, 5, 4]$
- On sélectionne $a[4] = 4$, et on le place à l'index 3. $a = [\underline{1, 2, 3, 4}, 5]$
- Le dernier élément est forcément à sa place. $a = [\underline{1, 2, 3, 4, 5}]$

Tri par sélection « en place »

Au début de chaque itération, on a (invariant de boucle) :



La fonction ressemblera donc à (dernière itération, `i=len[a]-2`) :

```
def tri_selection(a):
    for i in range(0, len(a) - 1):
        # Invariant: les éléments 0 à i-1 du tableau
        # sont triés et à leur place.
        # Itération:
        # - chercher l'index j du plus petit élément de
        #   la partie non triée et sa valeur v
        # - décaler vers la droite les éléments j-1 à i
        #   (parcours décroissant)
        # - mettre v à la position i
```

Tri par sélection « en place » : itération

On a donc deux boucles, une pour trouver la position du min, l'autre pour décaler.

```
# - chercher l'index j du plus petit élément de
#   la partie non triée et sa valeur v
v = a[i]
j = i
for k in range(i + 1, len(a)):
    if a[k] < v:
        v = a[k]
        j = k
# - décaler vers la droite les éléments j-1 à i
#   (parcours décroissant)
for k in range(j - 1, i - 1, -1):
    a[k + 1] = a[k]
# - mettre v à la position i
a[i] = v
```

démo : `tri_selecion.py`

Tri par selection « en place » : coût

les opérations "coûteuses" sont :

- la recherche du minimum ($n - i$ comparaisons),
- le décalage des éléments (en moyenne $\frac{n-i}{2}$ affectations), donc au total et en moyenne :

$$C(n) = \sum_{i=0}^{n-1}(n-i) = \sum_{k=1}^{n} k = \frac{(n-1)n}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

Le terme dominant de C quand n est grand est en n^2 : $C(n) = O(n^2)$.
La complexité du tri par sélection est donc *quadratique*.

Tri par sélection « en place » : questions

1. Quelle est la complexité dans le pire des cas ? A quelle configuration de tableau correspond-elle ?
2. Quelle est la complexité dans le meilleur des cas ?
3. Ce tri fonctionne-t-il avec autre chose que des listes d'entiers ?

Tri par sélection « en place » : réponses

1. La recherche du minimum oblige à toujours parcourir le tableau non trié, donc la complexité est en $O(n^2)$ dans tous les cas.
2. Il y a moins d'affectations si le tableau est déjà trié (zéro itérations dans la deuxième boucle), mais le terme dominant reste la première boucle.
3. Oui, avec toute liste de types qu'on peut affecter et comparer.

Sommaire

Pour la prochaine fois ...

N'oubliez pas le TP en temps libre.

- Lire la section 8.1 (recherche dichotomique du zéro d'une fonction) du livre « informatique pour tous en classes prépa ».
  Attention, il y a des compléments non-vus en cours [1].
- Lire la section 13.1 du livre « informatique pour tous en classes prépa ».
- Faire le QCM 6 sur Chamilo.

1. Quelques subtilités sur la recherche dichotomique (différence entre `return a` et `return (a + b)/2`, explications détaillées sur la complexité du if/for/while/...)