NSI 1ère - Algorithmique - Trier

QK

January 19, 2019

Trier

Trier : définition.

Algorithme de tri Algorithme qui, partant d'une liste, renvoie une version **ordonnée** de la liste.

$$[5, 1, 4, 3, 2] \rightarrow [1, 2, 3, 4, 5]$$

Variété

Il existe de nombreux algorithmes de tri. On peut citer, parmi les plus courants :

- Tri par insertion
- Tri par selection
- Tri à bulle
- Tri rapide
- Tri fusion
- Tri par tas
- Tri par comparaison
- Smoothsort
- Timsort

Pourquoi en faire autant ?

Deux raisons essentielles :

1. Les applications sont fréquentes
2. Les applications sont variées

Pourquoi trier ?

Et bien, remarquons qu'on peut trier à peu près n'importe quoi.
Le tout est de définir une *comparaison*

- Fichiers par taille,
- Livres par ordre alphabétique
- Dossier par profondeur dans une arborescence,
- Joueurs par score,
- Fichiers par date d'édition etc.

Mais pourquoi aurait-on besoin de PLUSIEURS algorithmes ?

À nouveau car les applications varient.

- Si les données sont très nombreuses on va limiter la **complexité calculatoire**
- Si les données occupent beaucoup d'espace, on est tenté de limiter la **complexité spatiale**

Doit-on programmer nous même pour trier ?

Tout dépend des langages ! Python (mais beaucoup d'autres aussi) dispose de son propre tri qui est le plus efficace à ma connaissance.

Python emploie Timsort (*Tim Peters*, 2002) qui résout à la fois presque tous les problèmes :

1. Meilleure complexité calculatoire dans la majorité des cas
2. Complexité spatiale parmi les meilleures
3. Très rapide en pratique ($\neq$ 1.)

Python

2 méthodes différentes :

1. `liste.sort()` : **seulement les listes**

```
>>> liste = [4,3,2]
>>> liste.sort() # change la liste !
>>> liste
[2, 3, 4]
```

2. `sorted(objet)` : tous les objets **itérables**

```
>>> strs = ['ccc', 'aaaa', 'd', 'bb']
>>> sorted(strs, key=len) # ne change pas l'objet
['d', 'bb', 'ccc', 'aaaa']
```

Algorithmes de tri

On ne fera pas mieux que Python !

Ah non, c'est sûr. Si vous inventez un algorithme aussi efficace cette année, mon devoir est de vous signaler auprès des gens compétants pour qu'ils vous proposent l'avenir que vous méritez.

En clair, c'est très compliqué. On va se contenter d'efleurer le problème.

Qu'allons nous étudier alors ?

Le programme demande d'étudier en détail le **tri par insertion** et le **tri par sélection**.

J'ajouterai le **tri à bulle**, le **tri fusion** et le **tri rapide**

Comparons ces exemples :

Nom	Cas moyen	Pire des cas	Espace
Fusion	$n \log n$	$n \log n$	n
Rapide	$n \log n$	n^2	n
Insertion	n^2	n^2	1
Selection	n^2	n^2	1
Bulle	n^2	n^2	1

$\log n$ croît moins vite que n vers l'∞

$n \log n$: mieux que n^2

Limites, contraintes (en théorie)

- En terme de complexite calculatoire, $n \log n$ est la limite Un tri *par comparaison* ne peut pas faire mieux que ça.
- En terme de complexité spatiale, 1 c'est le mieux. L'algorithme n'utilise que l'espace de départ.
- Les deux en même temps ? C'est possible... mais pas efficace en pratique.

Les usages sont nombreux et différents

- Peu de données ? un algorithme simple est parfois le meilleur (insertion / selection)
- Beaucoup de données ? Timsort (Fusion + insertion), introsort (rapide + par tas)
- Enormément de données ? Il faut sortir l'artillerie lourde. . .

Tri par insertion

Présentation

C'est le tri du joueur de carte.

- On itère sur la liste.
  - Chaque nouvel élément est inséré à sa place parmi les éléments déjà triés .
  - On recommence jusqu'à épuisement.

Tri *stable* : il ne change pas l'ordre de deux éléments “égaux”
Tri en *place* : il n'utilise pas plus de mémoire

Exemple

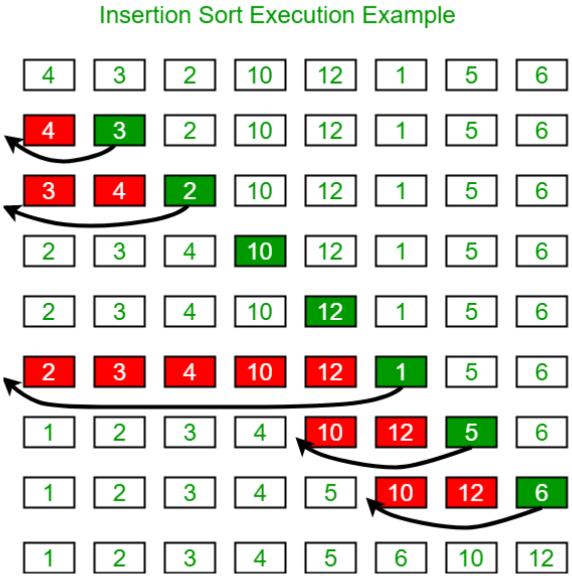

Figure 1: Exemple

Pseudo code

```
i = 1
Tant que i < longueur(A)
    j = i
    Tant que j > 0 et A[j-1] > A[j]
        echanger A[j] et A[j-1]
        j = j - 1
    fin tant que
    i = i + 1
fin tant que
```

Python : tri par insertion

```
def tri_insertion(A):
    i = 1
    while i < len(A):
        j = i
        while j > 0 and A[j-1] > A[j]:
            A[j-1], A[j] = A[j], A[j-1]
            j = j - 1
        i = i + 1
```

InteractivePython.org

Tri par selection

Présentation

Amélioration du tri à bulle.

On sépare la liste en 2 parties :

- déjà triés (construite de gauche à droite)
- pas encore triés.

On commence avec une liste déjà triée vide. On itère sur la liste et, à chaque tour on range le plus petit élément de la liste non triée à droite de la liste triée.

Tri *stable* : il ne change pas l'ordre de deux éléments “égaux”
Tri en *place* : il n'utilise pas plus de mémoire

Exemple

Triés	Non Triés	Plus petit restant
()	(1, 3, 4, 2)	(1)
(1)	(3, 4, 2)	(2)
(1, 2)	(3, 4)	(3)
(1, 2, 3)	(4)	(4)
(1, 2, 3, 4)		

Pseudo code

```
tri_selection(tableau t, entier n)
    pour i de 1 à n - 1
        min = i
        pour j de i + 1 à n
            si t[j] < t[min], alors min = j
        fin pour
        si min = i, alors échanger t[i] et t[min]
    fin pour
```

Python

```
def selection(a):
    n = len(a)
    for i in range(n):
        m = i  # on cherche l'indice du min
        for j in range(i+1, n):
            if a[m] > a[j]:
        m = j
    a[m],a[i] = a[i],a[m] # on echange
```

InteractivePython

Invariants de boucle

Comment sait-on que ça fonctionne ?

On s'en assure :

1. en cherchant une propriété vérifiée à chaque itération de la boucle : **un invariant de boucle**,
2. en prouvant que l'itération **se termine bien**.

Invariants de boucle

C'est une propriété qui est **vraie à chaque étape d'une boucle**

Tri par insertion : invariant et terminaison

Tri par insertion

```
i = 1
Tant que i < longueur(A)
    j = i
    Tant que j > 0 et A[j-1] > A[j]
        echanger A[j] et A[j-1]
        j = j - 1
    fin tant que
    i = i + 1
fin tant que
```

Tri par insertion : invariant et terminaison

Invariant

Les `i` premiers éléments sont triés.

1. C'est vrai dès le départ car on commence à `i = 1`
2. Cela reste vrai car on ajoute le nouvel élément à sa place.

Terminaison

1. À chaque tour de la boucle extérieure, la liste restante diminue.
2. À chaque tour de la boucle intérieure, `j` diminue. Elle s'arrête bien.

Tri par selection : invariant et terminaison

Tri par selection

```
tri_selection(tableau t, entier n)
    pour i de 1 à n - 1
        min = i
        pour j de i + 1 à n
            si t[j] < t[min], alors min = j
        fin pour
        si min = i, alors échanger t[i] et t[min]
    fin pour
```

Tri par sélection : invariant et terminaison

Invariant

Les `i` premiers éléments sont triés.

1. C'est vrai dès le départ car on commence à `i = 1`
2. Cela reste vrai car on ajoute à droite un élément plus grand que les autres.

Terminaison

1. À chaque tour de la boucle extérieure, la liste restante diminue.
2. À chaque tour de la boucle intérieure, `j` augmente. Elle s'arrête bien.

Complexité du tri par insertion

Complexité calculatoire, complexité en espace

Complexité calculatoire : **Majoration** du **nombre de calculs** réalisés par un algorithme

Complexité en espace : **Majoration** de **l'espace en mémoire** nécessaire à un algorithme

Étude expérimentaile

- On trie des tableaux de taille croissante de 1000 à 10000 par pas de 100
  - Déjà triés
  - Aléatoires
  - Triés par ordre inverse

Déjà triés

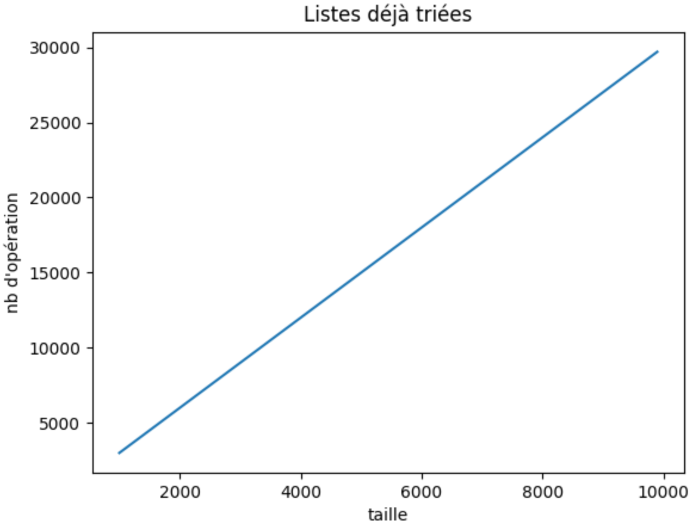

Figure 2: Déjà triés

Aléatoires

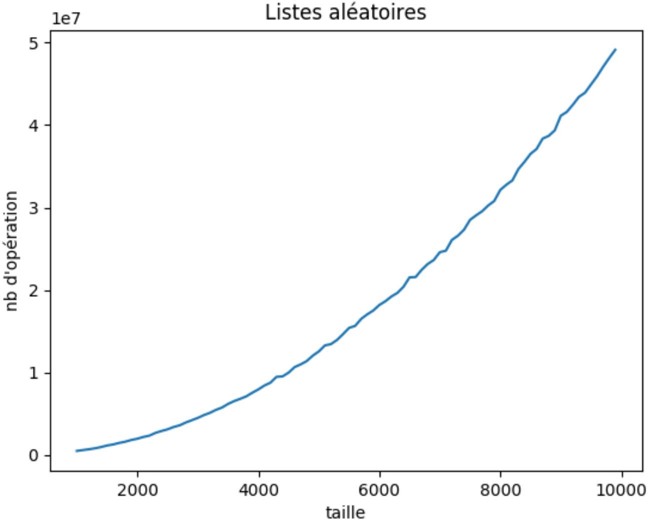

Figure 3: Aléatoire

Triés par ordre inverse

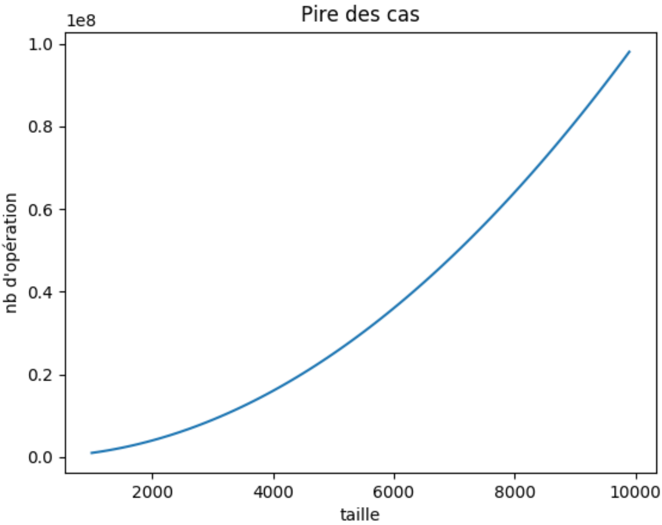

Figure 4: Triés par ordre inverse

Complexité : calcul à la main

- *pire des cas* : liste triée à l'envers.
- Pour simpliflier, on ne compte que les échanges.
- Dans ce cas, à chaque itération de la boucle intérieure, on parcourt toute la liste triée et on échange tous les couples d'éléments avant d'insérer.
- Celle-ci contenant `i` éléments, le nombre d'échanges est :

$$C = 1 + 2 + 3 + \cdots + n$$

Insertion : $O(n^2)$

- Remarquons qu'on peut écrire C à l'envers :

$$\begin{array}{ccccccccc} C & = & 1 & + & 2 & + \cdots + (n-1) + & n \\ C & = & n & + (n-1) + \cdots + & 2 & + & 1 \end{array}$$

- On ajoute en colonne :

$$2C = (n+1) + (n+1) + \cdots + (n+1) + (n+1)$$

$$2C = n(n+1)$$

$$C = n(n+1)/2$$

C est toujours plus petit qu'un polynôme en n^2.

On note : $C = O(n^2)$

Le tri par insertion est de complexité **quadratique**

Complexité du tri par selection

1. Déterminer le pire des cas
2. Construire les figures similaires à l'étude expérimentale de l'insertion
   - On trie des tableaux de taille croissante de 1000 à 10000 par pas de 100
     - Déjà triés
     - Aléatoires
     - Triés par ordre inverse
3. Reprendre l'algorithme, compter les échanges (et seulement les échanges) dans le pire des cas.
4. Conclure

Compléments

Tri à bulles

Le plus facile à comprendre et implémenter

```
tri_à_bulles(Tableau T)
    pour i allant de taille de T - 1 à 1
        tableau_trié = vrai
        pour j allant de 0 à i - 1
            si T[j+1] < T[j]
                échanger(T[j+1], T[j])
                tableau_trié = faux
        si tableau_trié
            fin tri_à_bulles
```

Tri à bulles

- Complexite : $O(n^2)$
- Complexité en espace : constante (en place)
- Stable
- Meilleur cas : liste triée
- Pire des cas : liste triée à l'envers
- Utilisation pratique : aucune, juste pédagogique.
- Des améliorations sont possibles : combsort (tri à peignes)

Tri à bulles en Python

```
def bubble_sort(l):
    while True:
        changement = False
        for i in range(len(l) - 1):
            if l[i] > l[i+1]:
                l[i], l[i+1] = l[i+1], l[i]
                changement = True
        if not changement:
            return l
```

Tri Fusion

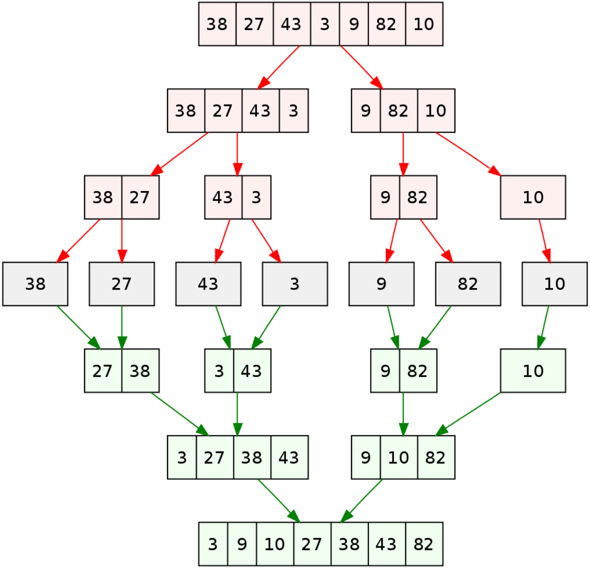

Figure 5: Tri Fusion

Tri rapide

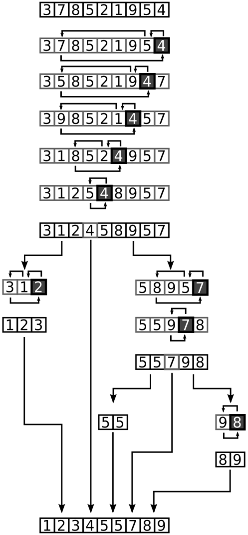

Figure 6: Tri rapide