9 – Exemples d'algorithmes de tri. Comparaison

Introduction

Un algorithme de tri est, en informatique ou en mathématiques, un algorithme qui permet d'**organiser** une collection d'objets selon une **relation d'ordre déterminée.** Les objets à trier sont des éléments d'un ensemble muni d'un **ordre total**.

- En mathématiques, on appelle relation **d'ordre total** sur un ensemble E toute relation d'ordre ≤ pour laquelle deux éléments de E sont toujours comparables, c'est-à-dire : $\forall\, x, y \ \in E \ \ x \leq y \ ou \ x \geq y$

Pourquoi trier les données ?

Pour appliquer des **algorithmes de recherche**, ceux-ci sont plus efficaces sur une liste triée (exemple : recherche par dichotomie)

Dans le domaine des **statistiques**, ils est nécessaire de trier les données afin d'en extraire la médiane, l'étendue ou les quartiles par exemple.

Pour classer des éléments selon une variable (ex : classement des équipes NBA en fonction de leur % de victoires, de leur *defensive rating, offensive rating,…*)

I– Algorithmes de tri élémentaire

Les algorithmes élémentaires suivant sont des algorithmes assez faciles à comprendre, à mettre en œuvre sous python mais leur complexité est quadratique.

a- Tri par sélection

i- Principe et pseudo-code

Rechercher le **plus petit élément** du tableau, et **l'échanger** avec l'élément d'indice **1** ;
Rechercher le second plus petit élément du tableau, **l'échanger** avec l'élément d'indice **2** ;
Continuer de cette façon jusqu'à ce que le tableau soit entièrement trié.

Ce qui fait qu'à l'itération i, la partie du tableau de 0 à i-1 est triée.

```
pour i de 1 à fin-1
    min ← i
    pour j de i + 1 à fin
        si liste[j] < liste[min], alors min ← j
    fin pour
    si min ≠ i, alors échanger t[i] et t[min]
fin pour
```

ii- Terminaison

i et j s'incrémente à chaque tour de boucle, étant situés dans une boucle for ils ont forcément un indice max, respectivement fin-1 et fin de la liste.

iii- Correction

L'invariant de boucle suivant permet de prouver la correction de l'algorithme : à la fin de l'étape i, la liste est une permutation de la liste initiale et les i premiers éléments de la liste coïncident avec les i premiers éléments de la liste triée.
Quand i attendra l'avant dernier élément et le compara avec le dernier, la liste sera entièrement triée.

iv- Complexité

À la première itération, on effectue n−1 comparaisons. À la i-ème itération, on effectue donc n−i comparaisons (puisque à chaque itération on décrémente la taille du tableau). Le nombre total de comparaisons pour trier un tableau de taille n est donc la somme des n−i pour i allant de 1 à n−1 :

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

La complexité (en comparaisons) de notre algorithme est quadratique : $O(n^2)$

Le tri par sélection effectuera toujours $\frac{n(n-1)}{2}$ comparaisons, même sur une liste déjà triée.
La complexité (en nombre d'échange) est de n-1 échanges dans le pire cas, qui est atteint par exemple lorsqu'on trie la séquence 2,3,…,n,1.

Ce tri peut-être intéressant, sur des petites listes (<20) dans le cas où le coût d'une comparaison est bien inférieure au coût d'un échange.

Le tri par sélection est un tri sur place (les éléments sont triés directement dans la structure). Ce n'est pas un tri stable (l'ordre d'apparition des éléments égaux n'est pas préservé).

tri_selection.py

b- Tri par insertion

i- Principe et pseudo-code

En informatique, le tri par insertion est un algorithme de tri classique. La plupart des personnes l'utilisent naturellement pour trier des cartes à jouer.

Dans l'algorithme, on parcourt le tableau à trier du début à la fin. Au moment où on considère le i-ème élément, les éléments qui le précèdent sont déjà triés.

L'objectif d'une étape est d'insérer le i-ème élément à sa place parmi ceux qui précèdent. Il faut pour cela trouver où l'élément doit être inséré en le comparant aux autres, puis décaler les éléments afin de pouvoir effectuer l'insertion. En pratique, ces deux actions sont fréquemment effectuées en une passe, qui consiste à faire « remonter » l'élément au fur et à mesure jusqu'à rencontrer un élément plus petit.

```
pour i de 1 à n - 1
    tmp ← T[i]
    j ← i
    tant que j > 0 et tmp<T[j - 1]
      T[j] ← T[j - 1]
      j ← j - 1
    fin tant que
    T[j] ← tmp
  fin pour
```

ii- Terminaison

On parcours tous les éléments de la liste, la boucle « for » se termine quoi qu'il arrive, il en est de même avec la boucle « while » car j est décrémenté à chaque tour et on vérifie qu'il est strictement supérieur à 0,donc il arrive forcement un moment où j<=0.

iii- Correction

A la i-ème itération, les i-1-ème éléments de la liste sont trié, l'insertion consiste donc à placer le i-ème élément à la bonne place et ainsi, les i-ème premiers éléments sont triées.

iv- Complexité

Dans le meilleur des cas, avec des données déjà triées, l'algorithme effectuera seulement *n* comparaisons. Sa complexité dans le meilleur des cas est donc en $O(n)$.

Dans le pire des cas, avec des données triées à l'envers, les parcours successifs du tableau imposent d'effectuer (n-1)+(n-2)+(n-3)..+1 comparaisons et échanges, soit n(n-1)/2 . On a donc une complexité dans le pire des cas du tri par insertion en $O(n^2)$.

Si tous les éléments de la série à trier sont distincts et que toutes leurs permutations sont équiprobables, la complexité **en moyenne** de l'algorithme est de l'ordre de (n2-n)/4 comparaisons et échanges. La complexité en moyenne du tri par insertion est donc également en $O(n^2)$

tri_insertion.py

c- Tri à bulles

i- Principe et pseudo-code

L'algorithme parcourt le tableau et compare les éléments consécutifs. Lorsque deux éléments consécutifs ne sont pas dans l'ordre, ils sont échangés. Ainsi, au premier tour on fait remonter le maximum de la liste de longueur n. Ensuite au deuxième tour on faire remonter le maximum de la liste de longueur n-1 (le max ayant été placé à la fin de la liste au tour précédent).

```
pour i allant de taille de T - 1 à 1
  pour j allant de 0 à i - 1
    si T[j+1] < T[j]
      échanger(T[j+1], T[j])
```

ii- Terminaison

Deux boucles for imbriqué qui parcours une liste de plus en plus petite, l'algorithme se termine quant il ne reste que le 1er et le 2ème élément à comparer.

iii- Correction

A la première itération, on place le plus grand à la fin, à la deuxième itération les 2 derniers éléments sont triés. A la i-ème itération, les éléments compris entre n-i et n sont triés.

iv- Complexité

Si tous les éléments de la série à trier sont distincts et que toutes leurs permutations sont équiprobables, la complexité en moyenne de l'algorithme est de l'ordre de n(n-1)/4 comparaisons et échanges. La complexité en moyenne du tri bulle est donc également en $O(n^2)$

tri_bulle.py

II- Algorithmes plus efficaces

Ces algorithmes sont plus rapides mais plus difficiles à comprendre et à implémenter.

a- Tri rapide (QuickSort)

i- Principe et Pseudo-code

La méthode consiste à placer un élément du tableau (appelé pivot) à sa place définitive, en permutant tous les éléments de telle sorte que tous ceux qui sont inférieurs au pivot soient à sa gauche et que tous ceux qui sont supérieurs au pivot soient à sa droite.

Cette opération s'appelle le partitionnement. Pour chacun des sous-tableaux, on définit un nouveau pivot et on répète l'opération de partitionnement. Ce processus est répété récursivement, jusqu'à ce que l'ensemble des éléments soit trié.

```
Fonction tri_rapide (liste) :
  Si liste vide, alors
        Return liste vide
  Sinon
        Pivot ← liste[-1]
        Plus_petit ← [éléments < pivot]

        Plus_grand ← [éléments > pivot]
        Return tri_rapide (plus_petit)+pivot+tri_rapide(plus_grand)
```

ii- Terminaison

Notons n la longueur de la liste à trier. **Soit (u_p) la suite des longueurs maximales successives des listes passées en argument de la fonction de tri rapide.**

Prenons l'exemple de T = [6, 5, 2, 1, 3, 4]. On a donc $u_0 = 6$ puis, avec l'algorithme :

[2, 1, 3] [4] [6, 5] donc $u_1 = 3$

A l'étape suivante :

[2, 1] [3] [] [4] [] [5] [6] donc $u_2 = 2$

En continuant ce processus, on obtient :

[] [1] [2] [3] [] [4] [] [5] [6] donc $u_3 = 1$

Pour tout p, on a $u_{p+1} < u_p$ car même si tous les éléments restants sont du même côté du pivot, on a retiré un élément : le pivot.
La suite (u_p) est une suite strictement décroissante d'entiers naturels. Elle ne prend qu'un nombre fini de valeurs, ce qui signifie que la fonction récursive tri_rapide n'est appelée qu'un nombre fini de fois. L'algorithme se termine.

iii- Correction

Par récurrence forte sur la longueur n de la liste passée en argument.

• **Initialisation**.
Si n = 0 ou n = 1, la liste est déjà triée et la fonction tri_rapide ne la modifie pas.

• **Hérédité**.

Soit $n \in \mathbb{N}^*$, on suppose que, pour toute liste L de longueur inférieure ou égale à n, l'appel tri_rapide(L) renvoie une copie triée de L.

Soit L une liste de longueur n + 1. Comme n + 1 > 1, on pose p = L[−1] et on définit deux listes L1 et L2 contenant les éléments de L respectivement strictement inférieur à p et supérieur à p.
Ainsi L1 et L2 de longueurs inférieures où égale à n car L privée de p ne contient que n éléments.
Par conséquent, l'hypothèse de récurrence assure que tri_rapide(L1) et tri_rapide(L2) renvoient deux listes triées, l'une contenant les éléments strictement plus petit que p et l'autre les éléments plus grand que p.

Il s'ensuit que la liste concaténée tri_rapide(L1) + [p] + tri_rapide(L2) est bien une copie triée de la liste L de longueur n + 1.

iv- Complexité

La complexité de tri rapide dépend de la distribution autour du pivot.

On peut raisonnablement penser que :

- **le pire des cas** correspond à un pivot qui est inférieur à tous les autres éléments du tableau (k = 0) ou supérieur à tous les autres éléments du tableau (k = n − 1). La complexité est en $O(n^2)$
- **le meilleur des cas** est celui où les deux sous-tableaux ont à peu près le même nombre d'éléments. La complexité est alors en $O(nlog_2(n))$.

Tri_rapide.py

b- Tri fusion

i- Principe et pseudo-code

L'opération principale de l'algorithme est la fusion, qui consiste à réunir deux listes triées en une seule. L'efficacité de l'algorithme vient du fait que deux listes triées peuvent être fusionnées en temps linéaire.

L'algorithme est naturellement décrit de façon récursive.

1. Si le tableau n'a qu'un élément, il est déjà trié.
2. Sinon, on sépare le tableau en deux parties à peu près égales.
3. On trie récursivement les deux parties avec l'algorithme du tri fusion.
4. On fusionne les deux tableaux triés en un tableau trié.

L'algorithme de tri fusion se divise donc en deux fonctions :
- une fonction « fusion » qui s'occupe de fusionner deux listes triées
- une fonction « tri_fusion » qui appelle « fusion » sur deux listes de taille n/2 (à un élément près si taille de la liste impair).

ii- Terminaison

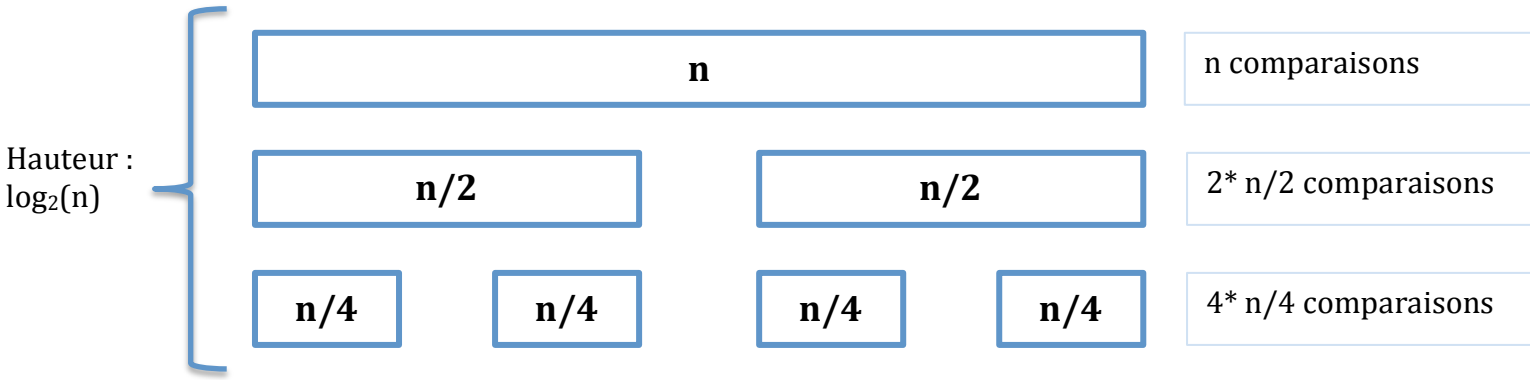
« tri_fusion » s'appelle elle-même avec des listes de taille strictement inférieure.
Cette fonction est donc appelée un nombre fini de fois.

Quant à la fonction « fusion » c'est une boucle « while » qui parcours les deux listes en les fusionnant.

iii- Correction

Fusion récursive de deux sous listes triées.

iv- Complexité



La complexité du tri fusion est donc en $O(nlog(n))$.

Tri_fusion.py

c- Tri par tas

i- Principe et pseudo-code

L'idée qui sous-tend cet algorithme consiste à voir le tableau comme un arbre binaire. Le premier élément est la racine, le deuxième et le troisième sont les deux descendants du premier élément, etc.

On peut définir deux sortes de tas binaires : les tas min et les tas max.

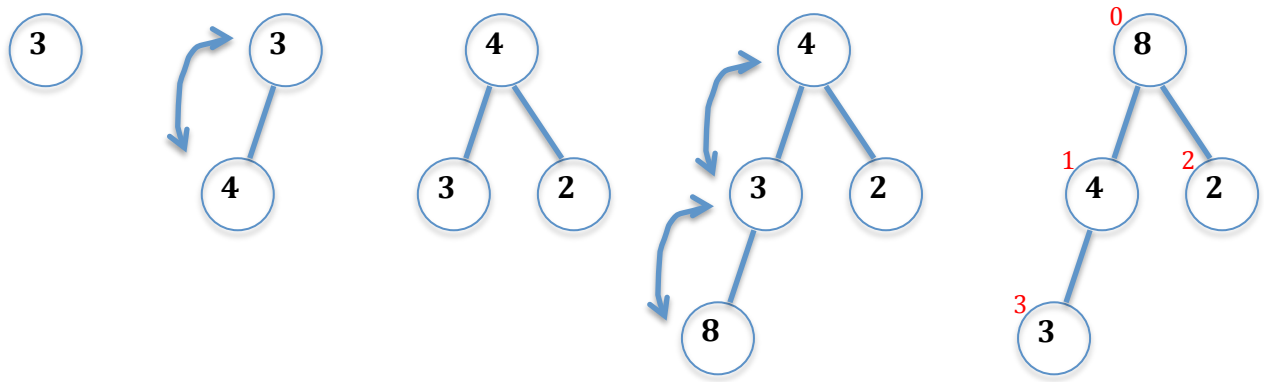
- Tas-min : chaque élément est supérieur à son parent.
- Tas-max : chaque élément est inférieur à son parent.

Les fils d'un nœud i sont d'indice 2*i+1 et 2*i+2.
Le père d'un nœud i est d'indice [(i-1)/2].

1ère étape :

Pour la Tas-max, c'est-à-dire que la racine (L[0]) est le plus grand éléments, en partant d'une liste non-triée quelconque on va « entasser » les éléments . C'est-à-dire que pour chaque élément de la liste, si l'élément d'indice i est plus grend que sont père, on les échange.

Ex : L=[3,4,2,8]



Liste après l'entassement : L = [8,4,2,3]

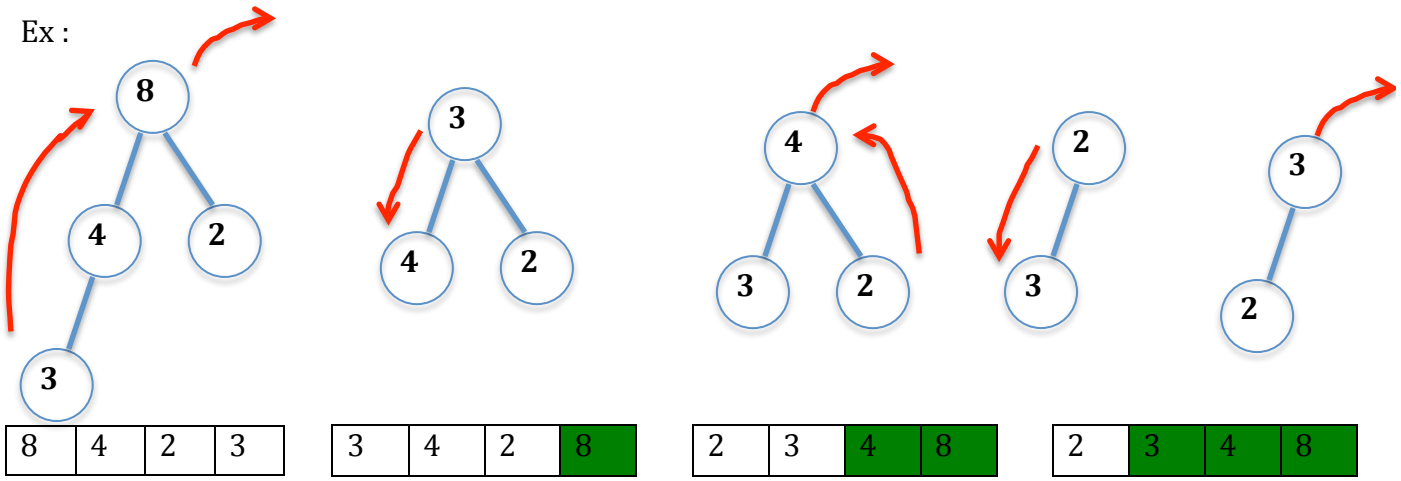
A chaque fois qu'on ajoute un élément a l'arbre on vérifie que :

L[i]<L((i-1)//2), sinon on inverse les 2 éléments dans la liste.

2ème étape :

Ensuite on procède au tamisage, c'est-a dire qu'on va enlever la racine de l'arbre (L[0] → le max de la liste) et le remplacer par la dernière feuille de l'arbre. Sur la liste, cette opération consiste à échanger le premier élément et le dernier afin d'avoir le max en fin de liste.
On compare ensuite successivement la nouvelle racine avec ses fils afin de faire remonter le nouveau max. Et on recommence jusqu'à ce que l'arbre soir vide.

Ex :



Liste finale : L =[2,3,4,8]

ii- Terminaison

L'opération « extraire la racine » consiste en faite à réduire le nombre d'éléments à trier dans la liste. A chaque itération on réduit donc de 1 la taille de la liste car les i derniers éléments sont déjà triés.

iii- Correction

iv- Complexité

Quand on extrait la racine de l'arbre en la remplaçant par la dernière feuille, on effectue ensuite l'opération pour faire descendre l'élément de la racine si ce n'est pas le max. Comme cette arbre est un arbre binaire presque complet (c'est-à-dire que tout les étages sont complet sauf le dernier), sa hauteur est donc $\log_2(n)$.L'opération « faire descendre la racine dans l'arbre » ne peut être exécuter que $\log_2(n)$ fois. Il y a n feuille dans l'arbre, la complexité du tri par tas est donc $O(n\log(n))$.

Comparaison (en temps) des algorithmes :

Comparaison_tri.py

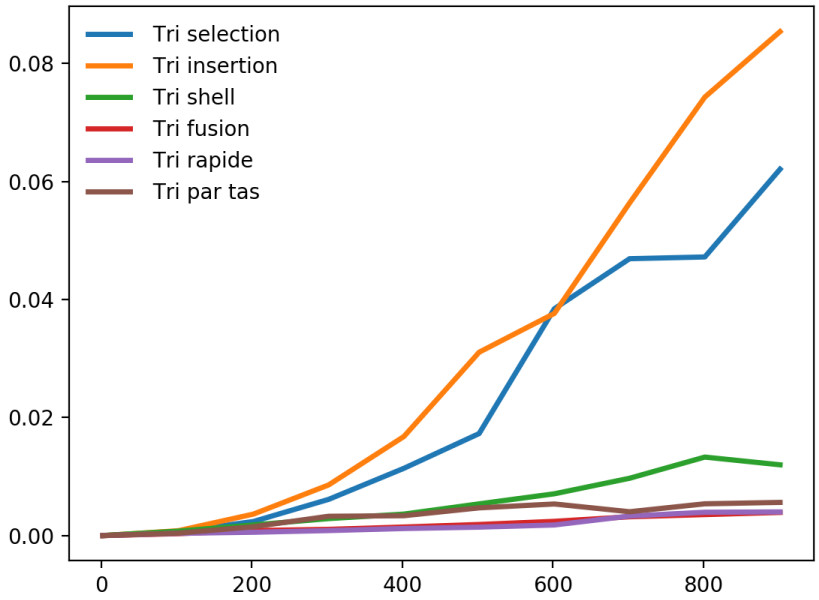


Figure 1 : Comparaison pour des listes de 2 à 1000 éléments par pas de 100

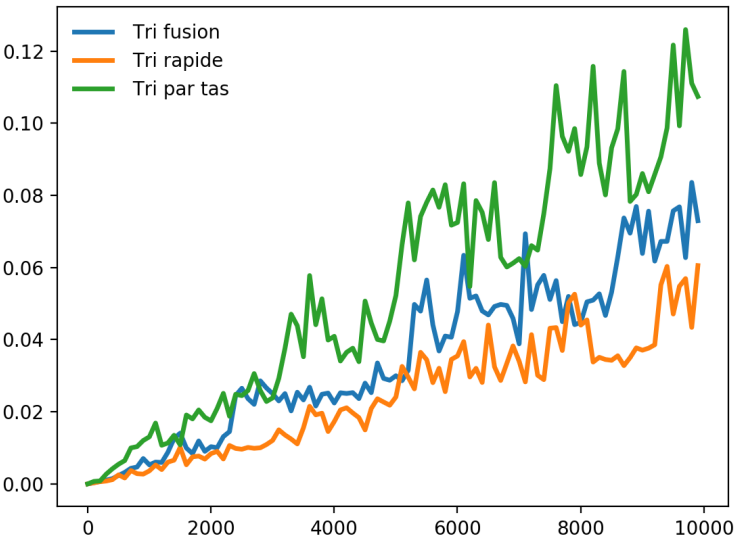


Figure 2 : Comparaison pour des listes allant de 2 à 10000 éléments par pas de 100

http://mathieu.gourcy.free.fr/cariboost_files/coursTrisIVP.pdf