Algorithmes de tri 1

Tri insertion, tri rapide

18 octobre 2017

Table des matières

1 Nécessité du tri

Lorsqu'une base de données est très volumineuse et non ordonnée, trouver une information dans cette base est très long et très ingrat. En effet la seule solution est alors d'éplucher une par une toutes les données jusqu'à trouver celle qu'on cherche, si elle existe. En revanche si la base est triée avec un ordre facilement compréhensible, on peut trouver rapidement l'information souhaitée.

Un exemple
Pour chercher un mot dans un dictionnaire qui en contient 200 000, et qui ne serait pas classé par ordre lexicographique, il faudrait en moyenne 30 heures ; et pour conclure que ce mot n'est pas dans le dictionnaire, il faudra y passer 60 heures. Dans un vrai dictionnaire, c'est à dire trié par ordre lexicographique,

même très volumineux, on met rarement plus d'une minute à trouver ce qu'on cherche.

2 Position du problème

2.1 Options

On suppose que la base de donnée est une liste de longueur n, contenant des éléments d'un type ordonnable : nombres, chaines de caractères, tuples de nombres ou chaines. On veut obtenir une autre liste contenant exactement les mêmes éléments, mais triée selon l'ordre choisi. Il y a deux options possibles :

- on peut vouloir conserver l'ancienne liste : on lui appliquera alors une *fonction pure* ;
- si ce n'est pas nécessaire, on choisira alors une *procédure pure* qui remplacera simplement la liste initiale par sa version triée.

La deuxième option est généralement préférable car mobilisera beaucoup moins de mémoire, surtout si on parvient à trier en place.

2.2 Tri en place

Un algorithme de tri est dit **en place** lorsqu'il n'utilise pas de liste autre que la liste donnée en argument (et donc la modifie directement). Ceci minimise la complexité spatiale.

Exemple : tri à bulles de [4,1,3,5,2]

- Étape 1 : [*1*,*4*,3,5,2]
- Étape 2 : [1,*3*,*4*,5,2]
- Étape 3 : [1,3,4,5,2]
- Étape 4 : [1,3,4,*2*,*5*]
- Étapes 5 et 6 : [1,3,4,2,**5**]
- Étape 7 : [1,3,*2*,*4*,**5**]
- Étape 8 : [1,3,2,**4**,**5**]
- Étape 9 : [1,*2*,*3*,**4**,**5**]
- Étapes suivantes : rien à faire.

2.3 Qualité d'un algorithme de tri

Les algorithmes de tri se distinguent par :

- leur complexité temporelle , qui déterminera la durée d'exécution ;
- leur complexité spatiale, importante quand la liste à trier est très longue, ce qui sera toujours le cas pour les applications professionnelles ;
- leur robustesse, c'est à dire leur capacité à bien se comporter quelle que soit la liste argument.

Il n'existe aucun algorithme de tri performant dans ces 3 domaines simultanément. On va donc en étudier 3, ayant des qualités différentes.

3 Tri par insertion

3.1 Principe

C'est le tri du joueur de cartes, qui insère chaque nouvelle carte à sa bonne place au fur et à mesure qu'il les reçoit.

Exemple
Liste à trier : L=[2,5,3,1,4] ; en main : T=[ ].

1. Insertion de L[0] : T=[*2*]
2. Insertion de L[1] : T=[2,*5*]
3. Insertion de L[2] : T=[2,*3*,5]
4. Insertion de L[3] : T=[*1*,2,3,5]
5. Insertion de L[4] : T=[1,2,3,*4*,5]

Intérêt principal : il est possible de trier en place.

Exemple : tri en place de L=[2,5,3,1,4]

1. Placement du 2 : L=[**2**,5,3,1,4] rien à faire.
2. Placement du 5 : L=[**2**,**5**,3,1,4] rien à faire.
3. Placement du 3 : 1 décalage nécessaire ;
   — *sauvegarde du 3* et décalage : m=3 , L=[**2**,5,**5**,1,4] ;
   — insertion du 3 : L=[**2**,**3**,**5**,1,4] .
4. Placement du 1 : 3 décalages nécessaires ;
   — *sauvegarde du 1* et 1^{er} décalage : m=1 , L=[**2**,**3**,5,**5**,4] ;
   — $2^{è}$ décalage : L=[**2**,3,**3**,**5**,4] ;
   — $3^{è}$ décalage : L=[2,**2**,**3**,**5**,4] ;
   — insertion du 1 : L=[**1**,**2**,**3**,**5**,4] .
5. Placement du 4 : 1 décalage nécessaire ;
   — *sauvegarde du 4* et décalage : m=4 , L=[**1**,**2**,**3**,5,**5**] ;
   — insertion du 4 : L=[**1**,**2**,**3**,**4**,**5**] .

3.2 Algorithme

À la $i^{è}$ étape on est dans la situation suivante :

$$\text{L=[}\mathbf{\text{L[0],...,L[i-1]}}\text{,}\textit{\text{L[i]}}\text{,L[i+1],...,L[n-1]]}$$

En gras : déjà placés. En italique : à sauvegarder puis insérer au bon endroit, ce qui va nécessiter des décalages ; on introduit donc une variable k qui va repérer la *position* qu'on libère.

— Initialisation : m reçoit L[i] , k reçoit i.
— Boucle : remplacer L[k] par L[k-1] *puis* k par k-1 tant que $k \geq 1$ et $L[k-1] > m$.
— Remplacer alors L[k] par m.
— On fait ceci pour chaque i variant de 1 à n-1.

3.3 Implémentation en Python

```
def inser_sort(L):
    n=len(L)
    for i in range(1,n):
        m,k = L[i],i
        while k>0 and L[k-1]>m:
            L[k],k = L[k-1],k-1
        L[k]=m
```

C'est une *procédure pure.*

3.4 Terminaison et correction

— Il n'y a pas d'appel récursif mais des boucles while, il s'agit donc de vérifier qu'elles se termineront. Pour chacune, k démarre de i et diminue d'une unité à chaque tour ; on tombe donc en i+1 tours sur 0, qui permet (si ce n'est pas encore fait) de sortir du while.

— Un **invariant de boucle** pour la boucle for est le booléen :

P(i) : « [L(0),. . .,L(i-1)] est trié »

On peut montrer par récurrence que P(i) est vraie pour tout i de 1 à n. P(n) signifie que la liste finale est triée : l'algorithme est donc correct.

3.5 Complexité

Nombre de mémoires :

$$M(n) = n + 4$$

car on n'a pas besoin de mémoires pour la sortie : il n'y en a pas.

La complexité spatiale est $O(n)$ ce qui est optimal.

Nombre d'opérations :

$$C(n) = 2 + \sum_{i=1}^{n-1} \Big(3 + w(i)\Big) + 1$$

où $w(i)$ est le nombre d'opérations utilisées pour la $i^{\text{è}}$ boucle while : $0 \leq w(i) \leq 4i$. Au pire :

$$C(n) = 3 + \sum_{i=1}^{n-1}(3 + 4i) = 3n + 4\frac{n(n-1)}{2}$$

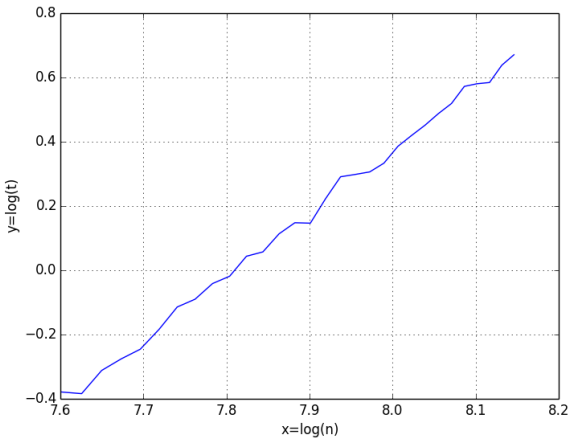
La complexité temporelle est $O(n^2)$ au pire et en moyenne.

Ceci signifie que trier une liste 2 fois plus longue prendra 4 fois plus de temps.

On peut vérifier expérimentalement que le temps t de calcul est effectivement proportionnel à n^2 en traçant un **graphique log-log**. En effet :

$$t = cn^2 \;\Leftrightarrow\; \ln(t) = \ln(c) + 2\ln(n)$$

— On écrit une fonction f qui calcule le temps mis pour trier une liste de longueur n.



— On prend N=[50,100,150,...], T=f(N) puis X=ln(N), Y=ln(T) et on trace par `plot(X,Y)`.
— On regarde si la courbe obtenue ressemble à une droite de pente 2.

C'est le cas. Les irrégularités proviennent de la fourchette $0 \leq w(i) \leq 4i$, qui génère une variabilité de $C(n)$.

4 Tri rapide

4.1 Principe

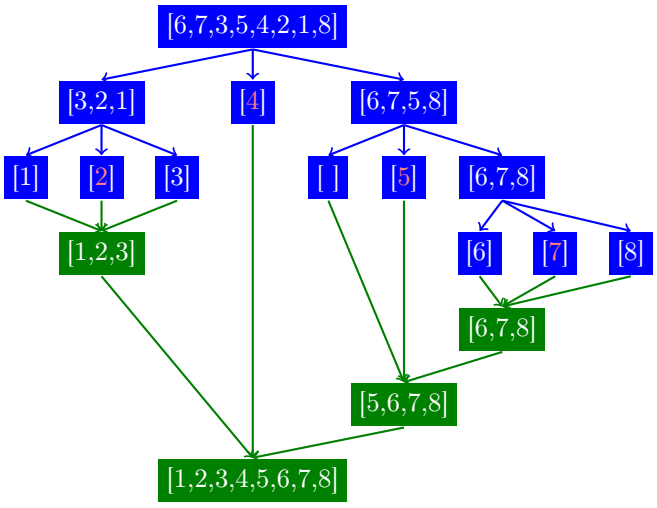
— On choisit un élément p dans la liste à trier, dit **pivot**. L'idéal serait qu'il soit médian, mais c'est coûteux de le choisir ainsi ; on se contentera en pratique de le choisir au milieu de la liste.
— Ce pivot sert à partitionner la liste L en 3 listes :

$$\begin{aligned} \text{Li} &= [\text{éléments} < \text{p}] \\ \text{Le} &= [\text{éléments} = \text{p}] \\ \text{Ls} &= [\text{éléments} > \text{p}] \end{aligned}$$

— On trie *récursivement* Li et Ls (inutile de trier Le), de façon à obtenir :

$$\begin{aligned} \text{Lit} &= [\text{éléments triés} < \text{p}] \\ \text{Le} &= [\text{éléments} = \text{p}] \\ \text{Lst} &= [\text{éléments triés} > \text{p}] \end{aligned}$$

— Il n'y a plus qu'à concaténer Lit, Le et Lst pour obtenir la version triée de L.



4.2 Algorithme

- On calcule la longueur n de la liste L à trier.Si $n \leq 1$ on retourne L, sinon :
- On calcule le pivot p et on crée 3 listes vides Li, Le, Ls.
- On teste successivement chaque élément de L, et suivant sa taille relativement à p on le place dans la liste idoine.
- On appelle récursivement la fonction de tri pour Li et Ls, ce qui donne des listes triées Lit et Lst.
- On retourne la concaténation de Lit, Le, Lst.

4.3 Terminaison et correction

La récursivité est valide car :

- les cas de base sont les listes de longueur ≤ 1 : déjà triées ;
- la variable de contrôle est la longueur du mot, qui diminue strictement à chaque appel : en effet Le est de longueur au moins 1, donc Li et Ls sont de longueur au plus n-1.

La preuve de la correction est assez difficile, alors qu'il est à peu près évident que l'algorithme fonctionne. Elle n'est donc pas abordée.

4.4 Implémentation en Python

```
def quick_sort(L):
    n=len(L)
    if n<=1: return L
    else:
        p=L[n//2] # pivot choisi au milieu de la liste
        Li,Le,Ls = [],[],[]
        for x in L:
            if x<p: Li.append(x)
```

```
        elif x==p: Le.append(x)
        else: Ls.append(x)
    return quick_sort(Li)+Le+quick_sort(Ls)
```

C'est une *fonction pure*.

4.5 Complexité

Nombre de mémoires :

— On crée n mémoires pour stocker L (mémoires « x »), n pour Li, Le, Ls et le résultat (mémoires « y »), 3 pour n, p, x.

— Il y a ensuite les mémoires créées par les appels récursifs, qui dépendent des tailles des listes Li et Ls.

— Dans le pire des cas, Li ou Ls est vide, l'autre est de taille n-1 :

$$M(n) = 3 + 2n + M(n-1)$$
$$M(n) = M(1) + \sum_{k=2}^{n}(3+2k) = c^{te} + 3n + n(n+1)$$

Dans le pire des cas (pivot min ou max), la complexité spatiale est $O(n^2)$, ce qui n'est pas très bon.

— Dans le meilleur des cas, Li et Ls ont même taille, qui est au plus $\frac{n-1}{2}$:

$$M(n) \leq 3 + 2n + M\left(\frac{n-1}{2}\right)$$

— Pour résoudre on introduit l'entier p tel que $2^p \leq n < 2^{p+1}$. Comme la complexité est une fonction croissante :

$$U(p) =_{def} M\left(2^p\right) \leq M(n) \leq M\left(2^{p+1}\right) = U(p+1)$$

— $U(p) = 3 + 2 \cdot 2^p + U(p-1) = c^{te} + \sum_{k=0}^{p}\left(3 + 2^{k+1}\right) \sim 2^{p+2}$

— $2^{p+2} \leq 4n$ donc $M(n) \leq 4n$.

Dans le meilleur des cas (pivot médian), la complexité spatiale est $O(n)$. En moyenne, elle sera intermédiaire entre $O(n)$ et $O(n^2)$.

Nombre de calculs :

— On fait un certain nombre de calculs ou test préliminaires, puis pour chaque x de la liste un certain nombre d'opérations ; au total $a + bn$ où a et b sont des constantes.

— Avec les appels récursifs, si Li et Ls sont de tailles r , s :

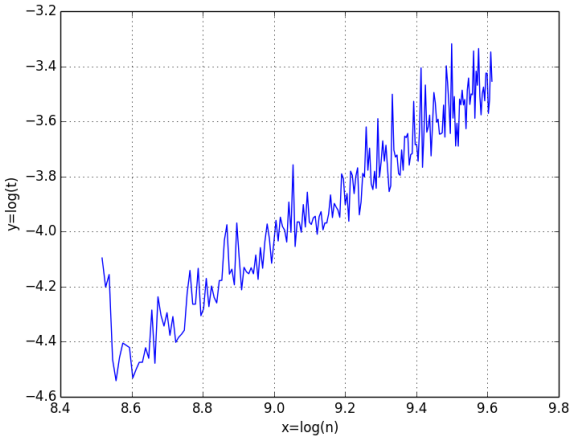
$$C(n) = a + bn + C(r) + C(s)$$

— Cas extrême : $(r, s) = (0, n-1)$ ou l'inverse ; alors $C(n) = a+bn+C(n-1)$. C'est la même récurrence que la complexité spatiale dans le pire des cas, elle conduit donc au même résultat.

Dans le pire des cas (liste déjà triée dans un sens ou l'autre), la complexité temporelle est $O(n^2)$.

— Cas où le pivot est médian : $r = s \leq \frac{n-1}{2}$; alors :

$$C(n) \leq a + bn + 2C\left(\frac{n-1}{2}\right)$$



— Soit p l'entier tel que $2^p \leq n < 2^{p+1}$ et $U(p) =_{def} C\,(2^p)$

$$U(p) \leq a + b2^p + 2U(p-1)$$

— Le facteur 2 est gênant, on introduit $V(p) = 2^{-p}U(p)$:

$$V(p) \leq a2^{-p} + b + V(p-1)$$

— $V(p) \leq c + \sum_{k=0}^{p} \left(a2^{-k} + b\right) = O(p)$ puis $U(p) = O(2^p p)$.

Dans le meilleur des cas (pivot systématiquement médian), la complexité temporelle est $O(n\ln(n))$.

Pour savoir ce qui se passe en moyenne, on crée un graphique log-log à partir de longues listes aléatoires d'entiers.

Commentaires :

— La courbe est très chaotique, car on est parfois dans un cas favorable, parfois non. *Ce tri n'est pas robuste.*

— Globalement la pente est plutôt 1, ce qui indiquerait une complexité $O(n)$.

— Ceci est dû au fait que pour n grand, $\ln(n)$ est quasi-constant ; il y a donc peu de différence entre $O(n\ln(n))$ et $O(n)$.

La complexité temporelle moyenne est $O(n\ln(n))$, ce qui est optimal pour un tri par comparaisons.