Algorithmes de tri 2

Tri fusion, tri casier

15 novembre 2017

Table des matières

1 Tri fusion

1.1 Principe

Comme pour le tri rapide, on utilise le principe « diviser pour régner » : on découpe un gros problème en 2 petits problèmes. On débouche alors sur un algorithme *récursif*. Dans le tri rapide, ce découpage se fait par partition autour d'un pivot. Si ce pivot n'est pas médian, les deux parties peuvent être plus ou moins disproportionnées, ce qui augmente le nombre de calculs et l'occupation mémoire. Le principe du tri fusion est de découper en 2 parties égales pour éviter ce problème. Mais il en apparait alors un autre : comment, à partir de 2 tableaux triés de taille p, faire un tableau trié de taille $2p$? C'est le problème de la **fusion**.

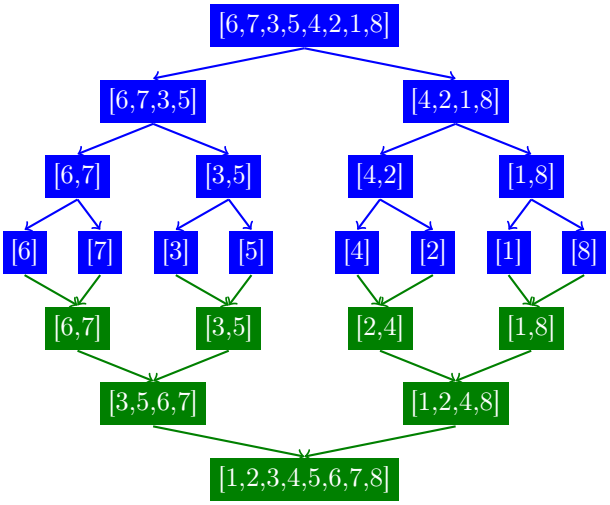
Dans le tri rapide, après les 2 tris partiels, tous les éléments de la partie gauche sont inférieurs au pivot, alors que tous les éléments de la partie droite sont supérieurs. Il est donc très facile d'en faire un tableau global trié : c'est une simple concaténation. Dans le tri fusion, le découpage en 2 est fait de manière

naturelle, et la difficulté est reportée à la fusion des 2 demi-listes récursivement triées. La solution s'appelle **interclassement**.

Problème	Solution Rapide	Solution Fusion
découpage	*partition*	naturel
fusion	concaténation	*interclassement*
gestion globale	récursivité	récursivité

En *rouge* : le problème principal.

1.2 Un exemple



Interclassement de Lg=[3,5,6,7] et Ld=[1,2,4,8] :

1. Lt=[ ] , Lg=[*3*,5,6,7] et Ld=[*1*,2,4,8]
2. Lt=[1] , Lg=[*3*,5,6,7] et Ld=[*2*,4,8]
3. Lt=[1,2] , Lg=[*3*,5,6,7] et Ld=[*4*,8]
4. Lt=[1,2,3] , Lg=[*5*,6,7] et Ld=[*4*,8]
5. Lt=[1,2,3,4] , Lg=[*5*,6,7] et Ld=[*8*]
6. Lt=[1,2,3,4,5] , Lg=[*6*,7] et Ld=[*8*]
7. Lt=[1,2,3,4,5,6] , Lg=[*7*] et Ld=[*8*]
8. Lt=[1,2,3,4,5,6,7] , Lg=[ ] et Ld=[*8*]
9. Lt=[1,2,3,4,5,6,7,8] , Lg=[ ] et Ld=[ ]

9 étapes pour interclasser 8 éléments.

1.3 Algorithme

Il est préférable d'utiliser 2 fonctions : une qui se charge de l'interclassement, l'autre qui se charge du découpage et de la récursivité.

Fonction principale :

- Entrée : une liste L. Sortie : une liste Lt triée contenant les mêmes éléments que L.
- On calcule la longueur de L ; si elle vaut 1, on retourne L ; sinon :
- On découpe L en 2 segments de longueurs égales, à 1 près.
- On trie récursivement ces segments.
- On les interclasse et on retourne le résultat.

Fonction-procédure d'interclassement :

- Entrée : 2 listes *triées* Lg et Ld. Sortie : une liste Lt triée contenant les mêmes éléments que Lg+Ld.
- On crée une liste vide Lt.
- Tant que Lg et Ld sont non vides, on compare le premier élément de Lg et Ld :
- Le plus petit des deux est retiré de la liste correspondante et mis à la fin de Lt.
- En sortie de boucle, une au moins des listes Lg, Ld est vide ; pour ne pas faire de test supplémentaire, on les concatène toutes les deux à Lt, et on retourne le résultat.

La récursivité est correcte car :

- les cas de base sont les listes de longueur 1 ;
- la variable de contrôle est la longueur n de la liste d'entrée, qui est divisée par 2 à chaque appel et donc décroît strictement, pour finir à 1.

1.4 Implémentation en Python

```
def interc(Lg, Ld):
    Lt=[]
    while Lg!=[] and Ld!=[]:
        if Lg[0]<Ld[0]: Lt.append(Lg.pop(0))
        else : Lt.append(Ld.pop(0))
    return Lt+Lg+Ld

def mergesort(L):
    n=len(L)
    if n==1: return L
    else:
        p=n//2
        Lg=mergesort(L[:p])
        Ld=mergesort(L[p:])
        return interc(Lg,Ld)
```

1.5 Complexité

Complexité spatiale de `interc` : À chaque étape la taille totale des 3 listes est inchangée et vaut la longueur n de la liste d'entrée ; donc `interc` coûte n mémoires. Complexité temporelle de `interc` :

- à chaque passage dans la boucle `while`, on fait 4 opérations (2 tests, 1 `append`, 1 `pop`) ;
- il y a au pire n passages dans le `while`, soit $4n$ opérations ;
- hors du `while`, il y a la concaténation finale, qui revient à faire au pire $n/2$ `append`.

Conclusion : `interc` coûte $O(n)$ opérations.

Complexité spatiale de `mergesort` :

- c'est une fonction pure qui retourne une liste de même longueur que l'entrée, on crée donc d'emblée n mémoires « x » et n mémoires « y » ;
- on crée 2 mémoires pour n et p ;
- le premier appel récursif utilise $M\left(\frac{n}{2}\right)$ mémoires, dont les mémoires « y » sont utilisées pour stocker Lg ;
- ces mémoires seront recyclées lors du deuxième appel récursif ;
- `interc` coûte n mémoires.

Total : $M(n) = 3n + 2 + M\left(\frac{n}{2}\right)$. C'est une récurrence du même type que pour la complexité spatiale de `quicksort` dans le meilleur des cas, elle conduit donc au même résultat : $M(n) = O(n)$.

Complexité temporelle de `mergesort` :

- le nombre$C(n)$ d'opérations vérifie la récurrence :

$$C(n) = a + 2C\left(\frac{n}{2}\right) + bn$$

- c'est une récurrence du même type que pour `quicksort` dans le meilleur des cas, elle conduit donc au même résultat :

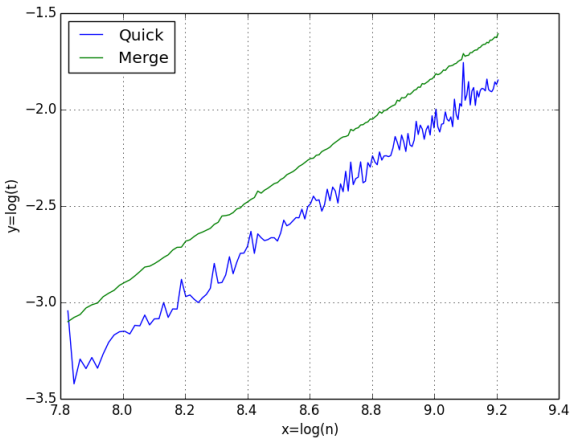
$$C(n) = O(n \ln(n))$$

Contrairement au tri rapide, cette complexité est *garantie dans tous les cas.*

Diagramme log-log de comparaison de `mergesort` et `quicksort` :

Comportement du tri fusion

- Robuste : toujours efficace, quelle que soit la liste argument.
- Rapide, mais un peu moins que le tri rapide.
- Il demande assez peu de mémoire, et peut même être externé, c'est à dire trier séparément des portions de la liste argument, quand celle-ci est trop grande.



2 Tableau synoptique

	Insertion	Rapide	Fusion
Compl. S	$O(n)$	$O(n)$ au mieux $O(n^2)$ au pire	$O(n)$
Compl. T	$O(n^2)$	$O(n\ln(n))$ en moyenne $O(n^2)$ au pire	$O(n\ln(n))$
Avantages	En place. Robuste.	Généralement rapide. Facile à programmer.	Robuste, rapide. Externable.
Problèmes	Lent sur les grandes listes.	Parfois lent. Parfois encombrant.	Un peu moins rapide que QS.

3 Tri casier

Ce tri est hors programme mais est plus rapide que le tri rapide, dans certains cas. En contrepartie, il est gourmand en mémoire, mais au vu de la capacité des ordinateurs actuels, c'est devenu un problème secondaire. Son principal défaut est de ne s'appliquer et n'être performant que dans certains cas.

3.1 Principe

En pratique, ce tri n'est applicable qu'aux listes d'entiers ou de mots. En effet, on a besoin de lister tous les éléments possibles entre le min et le max de la liste argument. On construit une nouvelle liste, dite casier, qui contient le *nombre d'occurrences* de chaque entier (ou mot) de la liste. On parcourt ensuite cette liste d'occurrences, et on remplit au fur et à mesure la version triée de la liste initiale

3.2 Un exemple

Liste à trier : L=[6,2,3,2,4,6,8,2,8,2]

— On calcule le min et le max : $m = 2$, $M = 8$.
— On crée un casier de longueur $M - m + 1$: C=[0,0,0,0,0,0,0]
— On parcourt L et incrémente l'élément correspondant de C, en position décalée de m :

1. C=[0,0,0,0,1,0,0] car 6 est en position 4
2. C=[1,0,0,0,1,0,0] car 2 est en position 0
3. C=[1,1,0,0,1,0,0]
4. C=[2,1,0,0,1,0,0]
5. C=[2,1,1,0,1,0,0]
6. C=[2,1,1,0,2,0,0]
7. C=[2,1,1,0,2,0,1]
8. C=[3,1,1,0,2,0,1]
9. C=[3,1,1,0,2,0,2]
10. C=[4,1,1,0,2,0,2]

Casier final : C=[4,1,1,0,2,0,2]

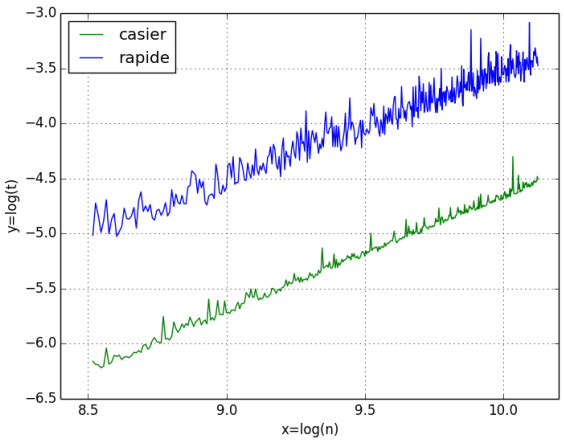
— On crée une liste vide qui contiendra au final le résultat : T=[ ]
— On parcourt C et remplit T :

1. T=[2,2,2,2] car 0 correspond à 2
2. T=[2,2,2,2,3] car 1 correspond à 3
3. T=[2,2,2,2,3,4]
4. T=[2,2,2,2,3,4]
5. T=[2,2,2,2,3,4,6,6]
6. T=[2,2,2,2,3,4,6,6]
7. T=[2,2,2,2,3,4,6,6,8,8]

La liste triée est [2,2,2,2,3,4,6,6,8,8] .

3.3 Implémentation en Python

```
def Casier(L):
    m,M = L[0],L[0]  # min et max provisoires
    for x in L:
        if x<m: m=x
        elif x>M: M=x
    p=M-m+1  # longueur du casier
    C=[0]*p  # liste de 0 de longueur p
    for x in L: C[x-m]+=1  # x-m : position de x dans C
    return m,C,p  # on a besoin des 3 pour la suite.
def countsort(L):
    m,C,p = Casier(L)
    T=[]
    for i in range(p):
        for j in range(C[i]): T.append(m+i)
    return T
```



3.4 Complexité

Nombre d'opérations pour `casier` :

— il y a 2 boucles `for x in L` de longueur n ; dans chacune on fait un nombre fixe d'opérations, donc $O(n)$;

— hors boucles, il y a un nombre fixe et donc négligeable d'opérations ; mais `[0]*p` coûte p opérations, qui est le nombre d'éléments *distincts* de L : dépend a priori de n ;

— au total $p + O(n)$.

Nombre d'opérations pour `countsort` :

— à chaque tour de la boucle `for i`, il y a $2C[i]$ opérations ;

— le nombre total d'opérations pour cette boucle est donc :

$$N = \sum_{i=0}^{p-1} 2C[i] = 2n$$

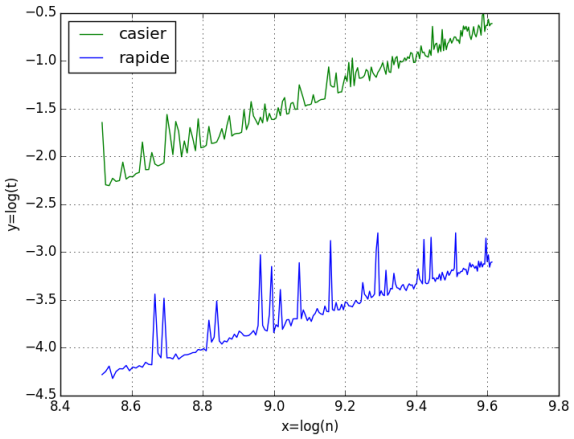
— au total on fait donc $p + O(n)$ opérations.

Si p est *borné*, c'est à dire quand les éléments de la liste ne peuvent prendre qu'un nombre borné de valeurs, alors la complexité temporelle est $O(n)$. C'est typiquement le cas quand on trie des entiers de moins de 6 chiffres par exemple, ou des mots de moins de 15 lettres. La complexité reste $O(n)$ quand p est lui-même un $O(n)$, ce qui est difficile à contrôler en pratique. Si ce n'est pas le cas, p peut être largement supérieur à n, auquel cas la complexité temporelle dépasse $O(n)$. *En pratique, la complexité n'est excellente que si les éléments de la liste argument sont bornés.*

Diagramme log-log de comparaison de `countsort` et `quicksort` dans le cas p borné :

Diagramme log-log de comparaison de `countsort` et `quicksort` dans le cas p non borné :

Nombre de mémoires utilisées par `casier` : au plus $n + p + c^{te}$. Nombre de mémoires utilisées par `countsort` : on crée encore une nouvelle liste de longueur



n, et un nombre fixe de mémoires accessoires, donc la complexité spatiale totale est $p + O(n)$. La complexité spatiale est donc bonne si p est borné, mauvaise sinon.

3.5 Utilisation d'un dictionnaire

En Python, au lieu de créer un casier sous forme de liste, il est préférable d'utiliser un **dictionnaire**, c'est à dire un ensemble du type :

$$C = \{..., x : y, ...\}$$

Les éléments proprement dits de cet ensemble sont les x et sont dits **clés** ; le y situé après x : est le **contenu** de la clé x. Pour créer une nouvelle clé x, ou modifier son contenu, on écrit simplement `C[x]=...` *Les clés sont triées au fur et à mesure de leur création par insertion naturelle.* Ici chaque clé est un élément de L, et son contenu est son nombre d'occurrences. Ainsi on n'utilise que les éléments qui apparaissent réellement dans L, et non tous les éléments entre m et M. Ceci évite aussi de calculer m et M, et permet d'utiliser des chaines de caractères.

```
def casier2(L):
    C={}
    for x in L:
        if x not in C: C[x]=1 # nouvelle clé
        else : C[x]+=1 # incrémentation du contenu
    return R

def countsort2(L):
    C=casier2(L)
    T=[]
    for x in C: # les clés sont triées par insertion
        y=C[x] # nombre d'occurrences de x dans L
        for i in range(y): T.append(x)
```

```
return T
```