

XVIII – Algorithmes de tri

1/ Introduction

1.1) Le problème du tri

On sait que la recherche dichotomique dans une liste triée est beaucoup plus rapide que la recherche brutale dans une liste quelconque (complexité logarithmique contre complexité linéaire). Il est donc naturel de chercher à mettre au point des procédures de tri efficaces.

Dans ce chapitre, on suppose que l'on trie une liste de nombres dans l'ordre croissant. En pratique, on aura à trier des listes d'objets plus lourds à manipuler (comme des matrices, dont on compare les déterminants) ce qui explique que l'on ne prenne en compte que les opérations sur ces éléments dans les calculs de complexité.

On notera n le nombre d'éléments à trier et l'on étudiera les complexités des différents algorithmes de tri proposés, afin de montrer qu'ils ne sont pas équivalents.

1.2) Complexité des tris

Pour comparer deux algorithmes de tri, on comptera deux types d'opérations distinctes :

- les comparaisons de deux éléments de la liste
- les affectations d'éléments de cette liste

en supposant que ces deux opérations s'effectuent en temps constant. On s'intéressera alors à la complexité dans le meilleur et dans le pire des cas, ainsi qu'à la complexité en moyenne.

On peut déjà noter que :

❶ La complexité dans le meilleur des cas d'un tel algorithme est en $\mathcal{O}(n)$.

Même si la liste est déjà triée, il faut en effet $n - 1$ comparaisons pour s'en assurer.

❷ On montre que la complexité dans le pire des cas est d'au moins $\mathcal{O}(n \ln n)$.

1.3) Propriétés intéressantes des tris

Outre leur complexité temporelle, deux caractéristiques des tris vont nous intéresser.

- On dit qu'un tri **s'effectue en place** si l'algorithme ne requiert qu'un espace en mémoire constant, en plus de la liste d'entrée. On ne risque pas de saturer la mémoire.
- Un tri est dit **stable** s'il ne modifie pas les positions relatives de deux éléments égaux pour notre critère. Cela s'avère intéressant lorsque l'on désire trier successivement les éléments selon plusieurs critères, comme les hauteurs et couleurs des cartes.

2/ Le tri par sélection

2.1) Principe

C'est peut-être celui auquel on pense en premier. Son principe est extrêmement simple.

- ❶ On place le plus petit élément de la liste en première position, en l'échangeant avec l'élément qui l'occupait s'ils sont différents.
- ❷ On place de même le plus petit des $n - 1$ éléments restants en deuxième position.
- ❸ Plus généralement, si les k premiers éléments de la liste sont triés et bien à leur place, on place le plus petit des $n - k$ éléments restants en $(k + 1)^{\text{e}}$ position.
- ❹ La dernière étape correspond à $k = n - 2$: à l'issue de celle-ci, les $n - 1$ premiers éléments de la liste sont triés et à leur place, donc le dernier élément est le maximum.

En pratique, cela revient à appliquer la première étape aux listes $L[k:]$ pour $k \in \llbracket 0 ; n - 2 \rrbracket$. Ainsi, la liste est entièrement triée à l'issue de $n - 1$ recherches de minimum.

2.2) Réalisation

Voici la procédure de tri en Python. Pas de grosse difficulté ici, les recherches d'extrema et de leurs indices ont déjà été croisées.

```
def tri_selection(L):
    n=len(L)
    for k in range(n-1):
        # Recherche du minimum de L[k:]
        mini = k
        for l in range(k+1,n):
            if L[l]<L[mini]:
                mini = l
        if mini != k:
            L[k],L[mini] = L[mini],L[k]
```

Cette procédure est constituée de deux boucles `for` imbriquées, donc elle se termine à coup sûr. Pour montrer la validité de l'algorithme, il suffit de vérifier que la proposition

$\mathcal{P}(k)$: « les éléments de $L[0:k]$ sont triés dans l'ordre croissant et sont plus petits que ceux de $L[k:]$ »

est un **invariant** de la boucle principale. Ceci ne pose pas de difficulté non plus.

2.3) Complexité

Chaque itération de la boucle interne contient une comparaison, ce qui en fait

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

d'où une complexité en $\mathcal{O}(n^2)$.

Examinons maintenant les affectations :

- dans le meilleur des cas (la liste est déjà triée), on n'en effectue aucune ;
- dans le pire des cas, on en effectue deux par passage dans la boucle externe, soit $2(n-2)$ en tout, comme par exemple avec la liste $L=[2, 3, \dots, n-1, n, 1]$.

Cela nous donne une complexité moyenne en $\mathcal{O}(n)$.

2.4) Propriétés

Ce tri s'effectue en place, mais il n'est pas stable à cause des échanges. En effet, si l'on trie la liste $L=[2, 2, 1]$, on échange le premier 2 et le 1, ce qui inverse les positions des deux 2.

3/ Le tri à bulles

3.1) Principe

C'est un des algorithmes de tris les plus populaires, mais aussi l'un des moins efficaces.

- ❶ On parcourt la liste $L=L[0:n]$ en comparant les couples d'éléments successifs. Lorsqu'ils ne sont pas rangés dans l'ordre croissant, on les échange. Après ce premier passage, le plus grand élément se trouve ainsi en fin de liste.
- ❷ On réitère alors le procédé avec la liste $L[0:n-1]$, puis avec $L[0:n-2]$, etc ... d'où le nom de l'algorithme : il imite la remontée de bulles dans un verre.
- ❸ Après $n-1$ passages, on est assurés d'avoir une liste triée. Cela peut arriver plus tôt : si l'on effectue un parcours sans aucun échange, la liste est déjà triée donc on s'arrête.

3.2) Réalisation

On réitère les parcours de listes à l'aide d'une boucle `while`, qui se poursuit tant qu'il y a des échanges. Pour les détecter, on utilise une variable booléenne nommée `Encore` :

```
def tri_bulles(L):
    p=len(L)
    Encore=True
    while Encore:
        Encore = False
        for i in range(p-1):
            if L[i]>L[i+1]:
                L[i],L[i+1] = L[i+1],L[i]
                Encore = True
        p=p-1
```

La boucle `while` admet la variable `p` pour **variant**, ce qui assure sa terminaison. De plus, elle admet pour **invariant** la proposition

$\mathcal{P}(p)$: « les éléments de la liste $L[p:n]$ sont à leur place »

ce qui prouve la validité de l'algorithme.

3.3) Complexité

Dénombrons les comparaisons et les affectations.

- Dans le meilleur des cas, si la liste est déjà triée, on effectue seulement un parcours, soit $n - 1$ comparaisons et aucune affectation. On a ainsi une complexité en $\mathcal{O}(n)$.
- Dans le pire des cas, si la liste est triée dans l'ordre décroissant, on effectue n parcours avec à chaque fois $p - 1$ comparaisons et échanges. Il y en a $\sum_{p=1}^n (p - 1) = \frac{n(n-1)}{2}$ en tout, d'où une complexité en $\mathcal{O}(n^2)$.

Comme on ne compare chaque élément qu'à son voisin immédiat, on se retrouve souvent dans le pire des cas ou presque. Ainsi, dès que certains des plus petits éléments se trouvent en fin de liste, on s'approche de la complexité maximale, comme avec $L=[2, 3, \dots, n, 1]$. On peut donc atteindre une complexité quadratique avec des listes presque triées, ce qui n'est vraiment pas terrible !

3.4) Propriétés

Ce tri s'effectue en place et il est stable, car il échange des éléments successifs et distincts.

4/ Le tri par insertion

4.1) Principe

C'est l'algorithme que l'on utilise lorsque l'on trie des cartes ou des copies. Il consiste à insérer successivement chaque élément dans l'ensemble des éléments déjà triés. Pour cela :

- ❶ On parcourt la liste à trier L du début à la fin à l'aide d'une boucle `for`.
- ❷ Lorsque l'on arrive à l'élément $L[i]$, la liste $L[0:i]$ est déjà triée. Il suffit alors d'insérer $L[i]$ à la bonne place pour que la liste $L[0:i+1]$ soit triée. Voici comment faire :
 - on utilise une boucle `while` portant sur un indice j initialement égal à i et qui diminue de 1 à chaque étape, si bien que les éléments de $L[0:i]$ sont les $L[j-1]$
 - tant que $L[j-1]$ est strictement supérieur à $L[i]$, on le décale d'un cran vers la droite en le stockant dans $L[j]$ (ce qui « libère » l'emplacement de $L[j-1]$)
 - dès que l'on trouve j tel que $L[j-1] \leq L[i]$, on place l'élément $L[i]$ à l'indice j

4.2) Réalisation

Voici ce que cela donne en employant la fonction `enumerate` pour varier les plaisirs :

```
def tri_insertion(L):
    for i,v in enumerate(L):
        j=i
        while j>0 and L[j-1]>v:
            L[j]=L[j-1]
            j=j-1
        L[j]=v
```

La boucle `while` se termine car l'entier j est un **variant** de boucle. Pour montrer la validité de l'algorithme, il suffit de noter que la boucle `while` admet pour **invariant** la proposition

$$\mathcal{P}(j) : \text{« pour tout entier } k \text{ tel que } j < k \leq i, \text{ on a } L[k] > v \text{ »}$$

tandis que la boucle `for` possède l'invariant

$$\mathcal{P}(i) : \text{« la liste } L[0 : i] \text{ est triée »}$$

4.3) Propriétés

Ce tri s'effectue en place (pas de création de liste) et il est stable puisque l'on ne décale que les éléments strictement supérieurs à $L[i]$: aucun risque d'inverser deux éléments égaux.

4.4) Complexité

Comptons les comparaisons et les affectations.

- Dans le meilleur des cas, la liste est déjà triée et la condition de la boucle `while` n'est jamais vérifiée. Cela fait alors une comparaison et une affectation à chaque tour de la boucle `for`, soit $n - 1$ opérations en tout. On a donc une complexité en $\mathcal{O}(n)$.
- Dans le pire des cas, la liste est triée en sens inverse et la boucle `while` est effectuée jusqu'à ce que $j=0$, ce qui donne i comparaisons et $i + 1$ affectations. On a ainsi en tout $\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$ comparaisons et $\sum_{i=0}^{n-1} (i+1) = \frac{n(n+1)}{2}$ affectations, c'est-à-dire une complexité en $\mathcal{O}(n^2)$.

Cependant, ce tri a plusieurs avantages par rapport au tri par sélection et au tri à bulles :

- ❶ Il effectue des affectations simples et non des échanges, ce qui minimise les opérations
- ❷ Si la liste est presque triée (les éléments sont proches de leur position définitive, ou le nombre d'éléments mal placés est borné), on montre que la complexité reste en $\mathcal{O}(n)$

Le tri par insertion est considéré comme le plus efficace sur des entrées de petite taille, et très rapide sur des listes presque triées, ce qui pousse en pratique à le combiner avec d'autres méthodes comme le tri rapide, qui gère bien les listes de grande taille.

5/ Le tri par comptage

5.1) Principe

Cet algorithme s'applique uniquement à une liste L de nombre entiers.

- ❶ On relève d'abord les occurrences de chacun des entiers de L dans un dictionnaire. On en profite pour déterminer le maximum M et le minimum m de L .
- ❷ On crée la liste triée en lui ajoutant successivement les éléments de $\llbracket m ; M \rrbracket$ présents dans le dictionnaire, en tenant compte de leur nombre d'occurrences bien entendu.

5.2) Réalisation

On écrit deux fonctions correspondant aux deux étapes de l'algorithme :

<pre>def comptage(L): D,m,M = {},L[0],L[0] for e in L: if e in D: D[e]=D[e]+1 else: D[e]=1 if e>M: M=e if e<m: m=e return D,m,M</pre>	<pre>def tri_comptage(L): """ Utilise la fonction comptage pour créer la liste triée """ D,m,M = comptage(L) T=[] for e in range(m,M+1): if e in D: for n in range(D[e]): T.append(e) return T</pre>
---	--

Les fonctions ne contiennent que des boucles `for`, qui se terminent forcément. Pas de souci non plus du côté de la validité : on a simplement traduit l'algorithme décrit précédemment.

5.3) Propriétés

Ce tri ne s'effectue pas en place puisque l'on doit créer un dictionnaire et une liste pour l'effectuer. Il n'est pas non plus stable, car on ne distingue pas les éléments de même valeur.

5.4) Complexité

La fonction `comptage` est de complexité linéaire car elle ne contient qu'une boucle `for` de n étapes. Pour la seconde fonction, c'est un peu plus subtil :

- les boucles internes génèrent exactement n additions, d'où une complexité en $\mathcal{O}(n)$.
- la boucle externe provoque $M + 1 - m$ tests, d'où une complexité en $\mathcal{O}(M - m)$.

Ainsi, la complexité est en $\mathcal{O}(n + d)$ en notant $d = M - m$ l'étendue de la liste L . Celle-ci dépassant rarement l'ordre de grandeur de n en pratique, on aura une complexité linéaire. Par contre, on doit créer un dictionnaire et une nouvelle liste de tailles sensiblement égales à celle de L , ce qui occupe de la mémoire. De plus, ce tri ne s'applique qu'à des listes de nombres entiers, ce qui est assez restrictif.

6/ Le tri rapide

6.1) Le principe

Cette méthode de tri récursive utilise la méthode « diviser pour régner » : on choisit un élément arbitraire de la liste, le **pivot**, puis on le met à sa place définitive en disposant les éléments plus petits à sa gauche et les autres à sa droite : c'est l'opération de **partitionnement**.

Il suffit ensuite de répéter récursivement le processus avec les deux sous-listes formées des éléments situés de part et d'autre du pivot pour obtenir une liste triée.

6.2) Réalisation

Pour réaliser ce tri, on écrit trois fonctions.

- La fonction `partition(L,g,d)` organise les éléments de la sous-liste `L[g:d]` autour du pivot et renvoie sa position finale (elle est indispensable lors du tri récursif). On prendra ici pour pivot le dernier élément de la liste, à savoir `L[d-1]`. Ensuite :
 - ❶ On parcourt la liste `L` des indices `g` inclus à `d-1` exclu.
 - ❷ Lorsque l'on trouve un élément strictement inférieur à `L[d-1]`, on l'échange avec `L[g]`, puis avec `L[g+1]`, etc ... On utilise pour cela une variable `p` initialement égale à `g` et incrémentée de 1 à chaque échange. De la sorte, `p` donne la position du successeur du dernier élément déplacé.
 - ❸ À la fin, il n'y a plus qu'à échanger `L[d-1]` et `L[p]` pour obtenir la partition voulue. Dans tous les cas, la position finale du pivot est `p`.
- La fonction auxiliaire `tri_rec(L,g,d)` appelle la précédente puis trie récursivement les sous-listes de part et d'autre du pivot. Cela permet de trier la sous-liste `L[g:d]`. Dans le cas où `g ≥ d-1`, la liste contient au plus un élément donc elle est déjà triée. C'est le **cas terminal** de notre fonction récursive.
- Enfin, la fonction `tri_rapide(L)` utilise la précédente la précédente pour trier `L`.

Voici ce que cela donne en Python :

<pre>def partition(L,g,d): v = L[d-1] p = g for i in range(g,d-1): if L[i]<v: if i!=p: L[i],L[p] = L[p],L[i] p=p+1 if p!=d-1: L[d-1],L[p] = L[p],L[d-1] return p</pre>	<pre>def tri_rapide(L): tri_rec(L,0,len(L)) def tri_rec(L,g,d): """ Tri récursif de L[g:d] """ if g<d-1: p=partition(L,g,d) tri_rec(L,g,p) tri_rec(L,p+1,d)</pre>
---	---

Notons que la boucle `for` de la fonction `partition` admet pour **invariant** la proposition

$\mathcal{P}(i)$: « les éléments de `L[g:p]` sont strictement inférieurs à `L[d-1]`,
ceux de `L[p:i]` sont tous supérieurs ou égaux à `L[d-1]`, et $i \geq p$ »

ce qui prouve la validité de l'algorithme de partitionnement.

6.3) Propriétés

Ce tri s'effectue en place (pas de création de liste supplémentaire), mais il n'est pas stable à cause des échanges effectués pour déplacer les éléments.

6.4) Complexité

On va s'intéresser au nombre de comparaisons effectuées, qui majore le nombre d'échanges.

- On peut déjà noter que la fonction `partition(L, g, d)` effectue $d - g - 1$ comparaisons car elle compare le pivot à tous les éléments de la liste `L[g:d-1]`.
- Soit maintenant $C(n)$ le nombre de comparaisons effectuées lors du tri récursif d'une liste de longueur n . Si le pivot est en k^{e} position, on se retrouve après partition avec deux sous-listes de $k - 1$ éléments à gauche et $n - k$ à droite, si bien que

$$C(n) = C(k - 1) + C(n - k) + n - 1$$

- Dans le pires des cas (quand $k = 1$ ou $k = n$), une des deux sous-listes est vide et l'on a $C(n) = C(n - 1) + n - 1$. Cette relation de récurrence conduit à $C(n) = \mathcal{O}(n^2)$.
- Dans le meilleur des cas, le pivot est en milieu de liste et $C(n) = 2C(\lfloor n/2 \rfloor) + \mathcal{O}(n)$, ce qui nous conduit à $C(n) = \mathcal{O}(n \ln n)$.
- On peut montrer que la complexité moyenne est également en $\mathcal{O}(n \ln n)$.

6.5) Problèmes et remèdes

Trois problèmes vont se poser et nous inciter à améliorer l'algorithme.

- ❶ Une liste triée (dans un sens quelconque) correspond au pire des cas, puisque l'une des deux sous-listes est toujours vide lors du tri récursif. Afin d'y remédier, on va modifier la fonction de partitionnement : on choisit aléatoirement le pivot dans la liste, puis on l'échange avec le dernier élément pour se ramener à la situation précédente.
- ❷ L'empilement des appels récursifs peut faire saturer la machine lorsque les sous-listes sont de tailles inégales. Pour éviter cela, on traitera systématiquement la plus petite sous-liste en premier, ce qui limite l'engorgement de la pile d'appels : le nombre d'appels empilés est alors en $\mathcal{O}(\ln n)$ dans le pire des cas, contre $\mathcal{O}(n)$ sinon.
- ❸ Enfin, le second appel récursif étant la dernière instruction, on peut le supprimer au profit d'une boucle `while` (qui remplace aussi le test `g < d - 1`). Chaque itération contient un partitionnement et le tri récursif d'une sous-liste, et modifie les indices manipulés pour traiter la sous-liste restante lors de la prochaine itération.

Voici les modifications à effectuer :

<pre>def tri_rec(L, g, d): while g < d - 1: p = partition(L, g, d) if p < g < d - 1 - p: tri_rec(L, g, p) g = p + 1 else: tri_rec(L, p + 1, d) d = p</pre>	<pre># La fonction pour effectuer # le tirage aléatoire. from random import randint def partition(L, g, d): # Choix du pivot a = randint(g, d - 1) L[d - 1], L[a] = L[a], L[d - 1] # La suite est inchangée</pre>
---	--

Remarque : l'ultime raffinement consiste à arrêter la récursion lorsque la taille des listes est petite (en-dessous de 20) et à achever le travail avec un tri par insertion, qui s'avère le plus efficace sur les listes de petite taille.

7/ Le tri fusion

7.1) Le principe

C'est encore un tri récursif basé sur la méthode « diviser pour régner ». Voici l'idée :

- ❶ On partage la liste à trier en deux listes de même taille, en la coupant au milieu.
- ❷ Une fois ces listes triées récursivement, on les fusionne (d'où le nom) en une liste triée.

On évite ainsi le pire cas du tri rapide, où les deux parties sont de tailles disproportionnées.

7.2) Réalisation

Pour réaliser le tri fusion, on écrit deux fonctions.

- La fonction `fusion(L1,L2)` prend en argument deux listes triées `L1` et `L2` et renvoie la liste triée obtenue en les fusionnant. Pour cela :
 - On crée d'abord une liste vide `L`.
 - On parcourt ensuite les deux listes `L1` et `L2` à l'aide de deux indices `i1` et `i2`.
À chaque étape, on ajoute le minimum de `L1[i1]` et `L2[i2]` à la fin de `L`.
On incrémente alors de 1 l'indice correspondant.
 - Lorsque la lecture d'une des listes est terminée, l'indice associé vaut sa taille.
Il ne reste qu'à ajouter les éléments non lus de l'autre liste à la fin de `L`.
- La fonction `tri_fusion(L)` utilise la précédente pour créer récursivement une copie triée de la liste `L`. Elle est assez simple à écrire.

Voici les codes associés :

```
def tri_fusion(L):
    n=len(L)

    # On n'oublie surtout
    # pas le cas terminal

    if n<2:
        return L
    else:
        m=n//2
        L1=tri_fusion(L[:m])
        L2=tri_fusion(L[m:])
        return fusion(L1,L2)
```

```
def fusion(L1,L2):
    n1,n2 = len(L1),len(L2)
    i1,i2,L = 0,0,[]
    while i1<n1 and i2<n2:
        e1,e2 = L1[i1],L2[i2]
        if e1<=e2:
            L.append(e1)
            i1=i1+1
        else:
            L.append(e2)
            i2=i2+1
    if i2==n2:
        L[i1+i2:]=L1[i1:]
    else:
        L[i1+i2:]=L2[i2:]
    return L
```

La boucle `while` admet pour **variant** la quantité `n1+n2-i1-i2`, ce qui assure sa terminaison. De plus, elle admet pour **invariant** la proposition

$\mathcal{P}(i1,i2)$: « la liste `L` est triée, ses éléments sont ceux de `L1[0:i1]` et `L2[0:i2]`, et ils sont plus petits que ceux de `L1[i1:]` et `L2[i2:]` »

ce qui prouve la validité de l'algorithme de fusion.

7.3) Propriétés

Ce tri ne s'effectue pas en place mais il est stable car, en prenant en priorité les éléments de la moitié de gauche de la liste (condition $e_1 \leq e_2$), on évite d'inverser deux éléments égaux.

7.4) Complexité

On va s'intéresser cette fois-ci au nombre $A(n)$ d'affectations effectuées lors du tri récursif d'une liste de longueur n , car il majore le nombre de comparaisons.

- Le découpage de cette liste en deux en provoque n car tous ses éléments sont copiés, de même que la fusion des deux sous-listes triées récursivement.
- De ce fait, $A(n) = 2A\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \mathcal{O}(n)$, ce qui conduit à $A(n) = \mathcal{O}(n \ln n)$.

Remarque : la complexité du tri fusion est plus faible que celle du tri rapide. Néanmoins, sur des listes aléatoires de taille moyenne, on observe qu'il est nettement moins efficace du fait qu'il ne s'effectue pas en place et occupe de la mémoire.

7.5) Variante de la fonction de fusion

On peut aussi écrire la fonction `fusion` avec une boucle `for` comportant n_1+n_2 étapes (son compteur k vaut i_1+i_2), en incluant les cas où l'une des deux listes est entièrement lue :

```
def fusion(L1,L2):
    n1,n2 = len(L1),len(L2)
    i1,i2 = 0,0
    L=[]
    for k in range(n1+n2):
        if i2==n2 or (i1<n1 and L1[i1]<=L2[i2]):
            L.append(L1[i1])
            i1=i1+1
        else:
            L.append(L2[i2])
            i2=i2+1
    return L
```

7.6) Variante du tri rapide

Voyons enfin une variante du tri rapide qui s'inspire de la fonction `tri_fusion`. Elle est simple à écrire mais n'est pas en place.

- ❶ On choisit un élément v au milieu de L , puis on le retire de la liste.
- ❷ On crée alors deux listes L_1 et L_2 constituées respectivement des éléments strictement inférieurs à v et des éléments supérieurs ou égaux à v .
- ❸ On renvoie la concaténation de L_1 triée récursivement, $[v]$ et L_2 triée récursivement.

On retrouve la même complexité et les mêmes problèmes que pour la dernière version du tri rapide, mais avec une efficacité moindre puisque le tri ne s'effectue pas en place.

Voici comment l'écrire avec une seule fonction récursive :

```
def tri_rapide(L):
    n=len(L)
    if n<2:
        return L
    else:
        v=L[n//2]
        L.remove(v)
        L1,L2=[],[]
        for e in L:
            if e<v:
                L1.append(e)
            else:
                L2.append(e)
        return tri_rapide(L1)+[v]+tri_rapide(L2)
```

Exercices

Exercice 1.

- 1) D rouler   la main l'algorithme de tri par s lection sur la liste $[15, 4, 2, 9, 55, 16, 0, 1]$.
- 2) Montrer la validit  de cet algorithme.

Exercice 2.

- 1) D rouler   la main l'algorithme de tri   bulles sur la liste $[15, 4, 2, 9, 55, 16, 0, 1]$.
- 2) Montrer la validit  de cet algorithme.

Exercice 3.

- 1) D rouler   la main l'algorithme de tri par insertion sur la liste $[15, 4, 2, 9, 55, 16, 0, 1]$.
- 2) Montrer la validit  de cet algorithme.

Exercice 4.

- 1) D rouler   la main l'algorithme de tri rapide sur la liste $[15, 4, 2, 9, 55, 16, 0, 1]$.
- 2) Montrer la validit  de cet algorithme.
- 3) Proposer un exemple de liste sur laquelle la complexit  est en $\mathcal{O}(n^2)$.
- 4) Proposer un exemple de liste sur laquelle la complexit  est en $\mathcal{O}(n \ln n)$.

Exercice 5.

- 1) D rouler   la main l'algorithme de tri fusion sur la liste $[15, 4, 2, 9, 55, 16, 0, 1]$.
- 2) Montrer la validit  de cet algorithme.

Exercice 6.

Modifier l'algorithme de tri rapide pour utiliser un tri par insertion d s que les tailles des listes   trier sont inf rieures   10.

Exercice 7.

On peut optimiser le tri fusion en  vitant la fusion lorsque,   l'issue des deux appels r cursifs, les  l ments de la moiti  gauche se trouvent  tre tous plus petits que ceux de la moiti  droite. Modifier l'algorithme en suivant cette id e.