Cours

Chapitre 3
Tri d'une liste de valeurs numériques

Savoirs et compétences :

- *Alg – C17 : tris d'un tableau à une dimension de valeurs numériques (tri par insertion, tri rapide, tri fusion).*

1 Présentation

Un algorithme de tri est un algorithme qui permet d'organiser une collection d'objets selon un ordre déterminé. Le tri permet notamment de faciliter les recherches ultérieures d'un élément dans une liste (recherche dichotomique). On s'intéresse ici à des méthodes de tri d'une liste de valeurs numériques. Celle-ci est implémentée sous la forme d'un tableau à une dimension.

R Pour trier des chaînes de caractères (mots), il suffit d'associer une valeur numérique à chaque caractère (code ASCII par exemple). On se limite dans le cadre du programme aux cas détaillés du tri par insertion, du tri rapide et du tri fusion.

2 Tri par insertion

2.1 Exemple du tri de carte

Soit un paquet de « n » cartes. On prend la première dans une main. On saisie la seconde carte et on l'insère avant ou après la première selon le cas. A l'étape « i », la ième carte est insérée à sa place dans le paquet déjà trié. Pour cela, on peut :

- soit partir du début du tas déjà trié et s'arrêter lorsqu'on rencontre une carte plus grande que la i^e (méthode 1) ;
- soit partir de la fin du tas déjà trié, et s'arrêter si on rencontre une carte plus petite que la i^e (méthode 2).

Le paquet contient alors « i » cartes triées. On procède ainsi de suite jusqu'à la dernière carte.

2.2 Application à une liste de nombres

5	8	3	2	9

												Méthode 1	Méthode 2
Étape 1	5	**8**	3	2	9	$\Longrightarrow$	5	**8**	3	2	9	1 comparaison 0 affectation	1 comparaison 0 affectation
Étape 2	5	8	**3**	2	9	$\Longrightarrow$	**3**	5	8	2	9	1 comparaison 3 affectations	2 comparaisons 3 affectations
Étape 3	3	5	8	**2**	9	$\Longrightarrow$	**2**	3	5	8	9	1 comparaison 4 affectations	3 comparaisons 4 affectations
Étape 4	2	3	5	8	**9**	$\Longrightarrow$	2	3	5	8	**9**	4 comparaisons 0 affectation	1 comparaison 0 affectation

L'analyse de la complexité de l'algorithme peut se faire par l'étude du nombre de comparaisons à effectuer.

2.3 Méthode 1

Pseudo Code

Algorithme : Tri par insertion – Méthode 1

Données :
- `tab`, liste : une liste de nombres

Résultat :
- `tab`, liste : la liste de nombres triés

tri_insertion(`tab`) :
  `n` ← **longueur**(`tab`)
  Pour `i` **de** 2 **à** `n` :
    `x` ← `tab[i]`
    `j` ← 1
    Tant que `j` $\leq$ `i-1` **et** `tab[j]<x` :
      `j` ← `j+1`
    Fin Tant que
    Pour `k` **de** `i-1` **à** `j-1` **par pas de** -1 **faire** :
      `tab[k+1]` ← `tab[k]`
    Fin Pour
    `tab[j]` ← `x`
  Fin Pour

Python

```
def tri_insertion_01(tab):
    """
    Trie une liste de nombre en utilisant la
    méthode du tri par insertion.
    En Python, le passage se faisant par référence,
    il n'est pas indispensable de retourner le tableau.
    Keyword arguments:
    tab -- liste de nombres
    """
    for i in range (1,len(tab)):
        x=tab[i]
        j=0
        while j<=i-1 and tab[j]<x:
            j = j+1
        for k in range(i-1,j-1,-1):
            tab[k+1]=tab[k]
        tab[j]=x
```

Dans le cas de la méthode 1, la complexité algorithmique peut se déterminer ainsi :

- meilleur des cas : le tableau est trié à l'envers. Il y a donc $n-1$ comparaisons à effectuer. La complexité est donc de classe linéaire : $C(n)=\mathcal{O}(n)$;
- pire des cas, le tableau est déjà trié. Il y a alors une comparaison à effectuer à la première étape, puis deux, ... puis $n-1$. On en déduit donc un nombre total de $\frac{n(n-1)}{2}$ comparaisons. La complexité est donc de classe quadratique : $C(n)=\mathcal{O}\left(n^2\right)$.

2.4 Méthode 2

▪ **Pseudo Code**

Algorithme : Tri par insertion – Méthode 2

Données :
- `tab`, liste : une liste de nombres

Résultat :
- `tab`, liste : la liste de nombres triés

tri_insertion(`tab`) :
  `n` ← **longueur**(`tab`)
  Pour `i` **de** 2 **à** `n` :
    `x` ← `tab[i]`
    `j` ← `i`
    Tant que `j >1` **et** `tab[j-1]>x` :
      `tab[j]` ← `tab[j-1]`
      `j` ← `j-1`
    Fin Tant que
    `tab[j]` ← `x`
  Fin Pour

```
def tri_insertion_02(tab):
    """
    Trie une liste de nombre en utilisant la méthode
    du tri par insertion.
    En Python, le passage se faisant par référence,
    il n'est pas indispensable de retourner le tableau.
    Keyword arguments:
    tab -- liste de nombres
    """
    for i in range (1,len(tab)):
        x=tab[i]
        j=i
        while j>0 and tab[j-1]>x:
            tab[j]=tab[j-1]
            j = j-1
        tab[j]=x
```

Dans le cas de la méthode 2 la complexité algorithmique peut se déterminer ainisi :

- meilleur des cas : le tableau est déjà trié. Il y a donc $n-1$ comparaisons à effectuer. La complexité est donc de classe linéaire : $C(n)=\mathcal{O}(n)$;
- pire des cas, le tableau est trié à l'envers. Il y a alors une comparaison à effectuer à la première étape, puis deux, ... puis $n-1$. On en déduit donc un nombre total de $\frac{n(n-1)}{2}$ comparaisons. La complexité est donc de classe quadratique : $C(n)=\mathcal{O}\left(n^2\right)$.

R On peut aussi montrer que la complexité en moyenne est de classe quadratique lorsque les permutations sont équiprobables. L'efficacité du tri par insertion est excellente lorsque le tableau est déjà trié ou « presque trié » ($C(n)=\mathcal{O}(n)$). Il surpasse alors toutes les autres méthodes de tri qui sont au mieux en $\mathcal{O}(n\ln(n))$.

3 Le tri rapide (*Quicksort*)

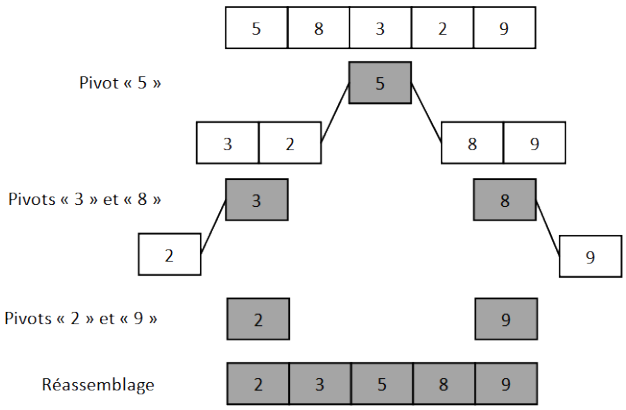
3.1 Méthode

▪ **Exemple** Tri de valeurs numériques Problème : Comment trier une liste de nombres par la méthode de « tri rapide » ? ▪

L'algorithme fait parti de la catégorie des algorithmes « diviser pour régner ». À chaque appel de la fonction de tri, le nombre de données à traiter est diminué de un. C'est-à-dire que l'on ne traite plus l'élément appelé « pivot » dans les appels de fonction ultérieurs, il est placé à sa place définitive dans le tableau. Le tableau de valeurs est ensuite segmenté en deux parties :

- dans un premier tableau, toutes les valeurs numériques sont inférieures au « pivot » ;
- dans un second tableau, toutes valeurs numériques sont supérieures au « pivot ».

L'appel de la fonction de tri est récursif sur les tableaux segmentés. On peut par exemple choisir le premier élément du tableau comme « pivot » :



Le « coût » temporel de l'algorithme de tri est principalement donné par des opérations de comparaison sur les éléments à trier. On raisonne donc sur le nombre de données à traiter pour l'analyse de la complexité de l'algorithme.

Dans le pire des cas, un des deux segments est vide à chaque appel de la fonction de tri. Cela arrive lorsque le tableau est déjà trié. Le nombre de données à traiter pour le i^e appel, est $n-i+1$. Le nombre total pour n appels de fonction est donc $\frac{n(n+1)}{2}$. On peut aussi écrire une relation de récurrence du type $C(n)=C(n-1)+n-1$ La complexité est donc de classe quadratique $C(n)=\mathcal{O}\left(n^2\right)$.

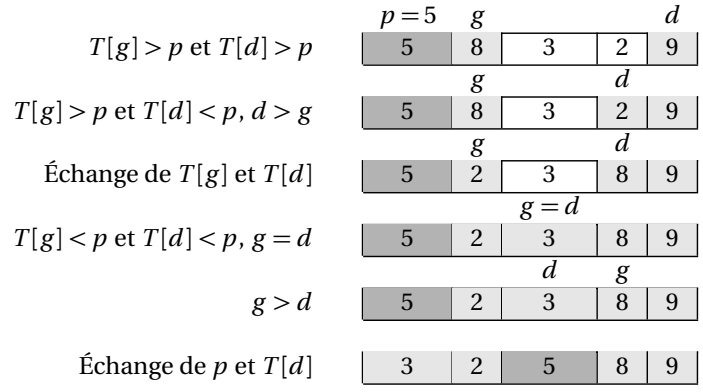
Dans le meilleur des cas, les deux segments sont de taille égale. Pour un nombre de données à traiter n, chacun des segments suivant a donc au plus $\frac{n-1}{2}$ éléments (on retire le pivot). On répète ainsi la segmentation des tableaux jusqu'à arriver au plus à un seul élément. On peut écrire une relation de récurrence du type $C(n)=2C\left(\frac{n-1}{2}\right)+n-1$.

La complexité est donc de classe quasi linéaire $C(n)=\mathcal{O}(n\ln(n))$.

R Certains algorithmes de tri rapide prennent pour « pivot » le dernier élément, la valeur moyenne du premier et du dernier, ou un positionnement aléatoire dans le tableau. Pour se placer dans le meilleur des cas pour chaque segment de tableau, il faut prendre pour pivot la valeur médiane du tableau de valeurs. Le problème est que cette recherche de pivot idéal a aussi un « coût ».

On écrit tout d'abord l'algorithme effectuant la segmentation du tableau :

- le pivot (1^{er} élément du tableau) est mis à sa place définitive ;
- pour des indices inférieurs, toutes les valeurs sont plus petites ou égales ;
- pour des indices supérieurs, toutes les valeurs sont plus grandes.



Le pivot a sa place définitive. Les éléments à sa gauche sont plus petits ou égaux. Les éléments à sa droite sont plus grands

3.2 Algorithmes du tri rapide

Pseudo Code

Algorithme : Tri Quicksort – Segmentation

Données :
- `tab`, liste : une liste de nombres
- `i`, `j`, entiers : indices de début et de fin de la segmentation à effectuer

Résultats :
- `tab`, liste : la liste de nombre segmenté avec le pivot à sa place définitive
- `k` entier : l'indice de la place du pivot

segmente(`tab,i,j`) :
```
g ← i+1
d ← j
p ← tab[i]
Tant que g ≤ d Faire
    Tant que d≥ 0 et tab[d]>p Faire
        d ← d-1
    Fin Tant que
    Tant que g≤ j et tab[g]≤p Faire
        g ← g+1
    Fin Tant que
    Si g<d alors
        Échange( tab,g,d )
        d ← d-1
        g ← g+1
    Fin Si
Fin Tant que
k← d
Échange( tab,i,d )
Retourner k
```

Python

```
def segmente(tab,i,j):
    """
    Segmentation d'un tableau par rapport à
    un pivot.
    Keyword arguments:
    tab (list) -- liste de nombres
    i,j (int) -- indices de fin et de début de la
    segmentation
    Retour :
    tab (list) -- liste de nombres avec le pivot
    à sa place définitive
    k (int) -- indice de la place du pivot
    """
    g =i+1
    d=j
    p=tab[i]
    while g<=d :
        while d>=0 and tab[d]>p:
            d=d-1
        while g<=j and tab[g]<=p:
            g=g+1
        if g<d :
            tab[g],tab[d]=tab[d],tab[g]
            d=d-1
            g=g+1
    k=d
    tab[i],tab[d]=tab[d],tab[i]
    return k
```

Résultat Le nombre de comparaisons du type $T[d] > p$ et $T[g] \leq p$ est égal à $n-1$. La complexité de cet algorithme est donc de classe linéaire : $C(n) = \mathcal{O}(n)$.

Pseudo Code

Algorithme : Tri Quicksort – Tri rapide

Données :
- `tab`, liste : une liste de nombres
- `i`, `j`, entiers : indices de début et de fin de la portion à trier

Résultats :
- `tab`, liste : liste triée entre les indices `i` et `j`

tri_quicksort(`tab,i,j`) :
```
Si g<d alors
    k← segmente(tab,i,j )
    tri_quicksort(tab,i,k-1 )
    tri_quicksort(tab,k+1,j )
Fin Si
```

Python

```
def tri_quicksort(tab,i,j):
    """
    Tri d'une liste par l'utilisation du
    tri rapide (Quick sort).
    Keyword arguments:
    tab (list) -- liste de nombres
    i,j (int) -- indices de fin et de
    début de la zone de tri
    Retour :
    tab (list) -- liste de nombres avec
    le pivot àsa place définitive
    """
    if i<j :
        k = segmente(tab,i,j)
        tri_quicksort(tab,i,k-1)
        tri_quicksort(tab,k+1,j)
```

R Cette méthode de tri est très efficace lorsque les données sont distinctes et non ordonnées. La complexité est alors globalement en $\mathcal{O}(n\ln(n))$. Lorsque le nombre de données devient petit (< 15) lors des appels récursifs de la fonction de tri, on peut avantageusement le remplacer par un tri par insertion dont la complexité est linéaire lorsque les données sont triées ou presque.

3.3 Tri rapide optimisé

Pseudo Code

Algorithme : Tri Quicksort – Tri rapide optimisé

Données :
- `tab`, liste : une liste de nombres
- `i`, `j`, entiers : indices de début et de fin de la portion de liste à trier

Résultats :
- `tab`, liste : liste triée entre les indices `i` et `j`

```
tri_quicksort_optimized(tab,i,j):
    Si i<j alors
        k← segmente(tab,i,j)
        Si k-i>15 alors
            tri_quicksort(tab,i,k-1)
        Sinon
            tri_insertion(tab,i,k-1)
        Fin Si
        Si j-k>15 alors
            tri_quicksort(tab,k+1,j)
        Sinon
            tri_insertion(tab,k+1,j)
        Fin Si
    Fin Si
```

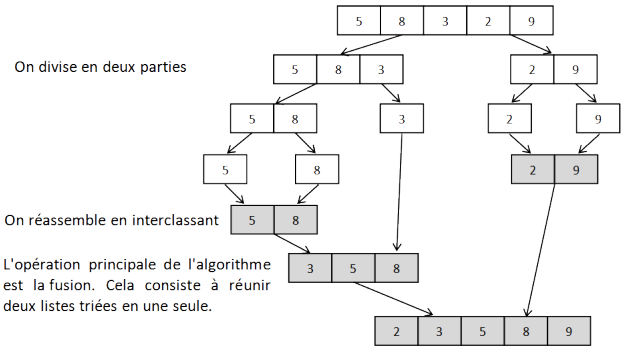
4 Le tri fusion

4.1 Méthode

La méthode de tri fusion pour un tableau de données est la suivante :

1. on coupe en deux parties à peu près égales les données à trier ;
2. on trie les données de chaque partie par la méthode de tri fusion ;
3. on fusionne les deux parties en interclassant les données.

L'algorithme est donc récursif. Il fait partie des algorithmes « diviser pour régner ». La récursivité s'arrête car on finit par arriver à des listes composées d'un seul élément et le tri est alors trivial.



4.2 Algorithme de tri fusion

Pseudo Code

Algorithme : Tri Fusion – Fusion de deux listes

Données :

- `tab`, liste : une liste de nombres `tab[g:d]` avec `g` indice de la valeur de gauche, `d` indice de la valeur de droite
- `m`, entier : indice tel que $g \leq$`m`$<$`d` et tel que les sous-tableaux `tab[g:m]` et `tab[m+1:d]` soient ordonnés

Résultats :

- `tab`, liste : liste triée entre les indices `g` et `d`

fusion_listes(`tab,g,d,m`) :

```
n1← m-g+1
n2← d-m
Initialiser tableau G
Initialiser tableau D
Pour i allant de 1 à n1 faire
    G[i] ← tab[g+i-1]
Fin Pour
Pour j allant de 1 à n2 faire
    D[j] ← tab[m+j]
Fin Pour
i ← 1
j ← 1
G[n1+1] ← +∞
D[n2+1] ← +∞
Pour k allant de g à d faire
    Si G[i]≤D[j] alors
        tab[k]← G[i]
        i← i+1
    Sinon
        Si G[i]>D[j] alors
            tab[k]← D[j]
            j← j+1
        Fin Si
    Fin Si
Fin Pour
```

Python

```
def fusion_listes(tab,g,d,m):
    """
    Fusionne deux listes triées.
    Keyword arguments:
     * tab (list) -- liste : une liste de nombres
        tab[g:d] avec g indice de la valeur de
        gauche, d indice de la valeur de droite
     * g,d,m (int) -- entiers : indices tels que
        g<=m<d et tel que les sous-tableaux
        tab[g:m] et tab[m+1:d] soient ordonnés
    Résultat :
     * tab (list) : liste triée entre les indices
       g et d
    """
    n1 = m-g+1
    n2 = d-m
    G,D = [],[]
    for i in range (n1):
        G.append(tab[g+i])
    for j in range (n2):
        D.append(tab[m+j+1])
    i,j=0,0
    G.append(99999999999)
    D.append(99999999999)
    for k in range (g,d+1):
        if G[i]<=D[j]: # and i<=n1
            tab[k]=G[i]
            i=i+1
        elif G[i]>D[j]: # and j<=n2
            tab[k]=D[j]
            j=j+1
```

Résultat Cet algorithme a une complexité en temps de classe linéaire : $C(n) = \mathcal{O}(n)$. Par contre, il oblige à utiliser un espace supplémentaire égal à la taille du tableau original `tab`.

■ **Pseudo Code**

Algorithme : Tri Fusion

Algorithme récursif du table de tri.

Données :

- `tab`, liste : une liste de nombres non triés `tab[g:d]`
- `g,d`, entiers : indices de début et de fin de la liste

Résultats :

- `tab`, liste : liste triée entre les indices `g` et `d`

tri_fusion(`tab,g,d`) :
  Si `g<d` **alors**
    `m` ← `(g+d)` **div** 2
    tri_fusion(tab,g,m)
    tri_fusion(tab,m+1,d)
    fusion_listes(tab,g,d,m)
  Fin Si

■ **Python**

```
def tri_fusion(tab,g,d):
    """
    Tri d'une liste par la métode du tri fusion
    Keyword arguments:
    tab (list) -- liste : une liste de nombres non
      triés tab[g:d]
    g,d (int) -- entiers : indices de début et
      de fin de liste si on veut trier
      tout le tableau g=0, d=len(tab)-1
    Résultat :
    tab (list) : liste triée entre les indices
      g et d
    """
    if g<d:
        m=(g+d)//2
        tri_fusion(tab,g,m)
        tri_fusion(tab,m+1,d)
        fusion_listes(tab,g,d,m)
```

Si l'on s'intéresse au nombre de données à traiter à chaque appel de fonction, la relation de récurrence est du type : $C(n)=2C\left(\frac{n}{2}\right)+n$. La méthode de tri fusion a donc une efficacité temporelle comparable au tri rapide en $\mathcal{O}(n\ln(n))$. Par contre, elle n'opère pas en place : une zone temporaire de données supplémentaire de taille égale à celle de l'entrée est nécessaire. Des versions plus complexes peuvent être effectuées sur place mais sont moins rapides.

5 Synthèse

	Tri par insertion[1]	Tri rapide	Tri fusion
Pire des cas	Liste triée $C(n)=\mathcal{O}(n^2)$	Liste triée[2] $C(n)=\mathcal{O}(n^2)$	$C(n)=\mathcal{O}(n^2)$
Cas moyen	$C(n)=\mathcal{O}(n^2)$	$C(n)=\mathcal{O}(n\log n)$	$C(n)=\mathcal{O}(n\log n)$
Meilleur des cas	Liste triée $C(n)=\mathcal{O}(n)$	Liste triée[3] $C(n)=\mathcal{O}(n\log n)$	$C(n)=\mathcal{O}(n\log n)$

1 : dépend de la méthode de tri.
2 : lorsque le pivot est la première valeur des listes.
3 : lorsque le pivot est pris au milieu des listes.

Références

[1] Patrick Beynet, *Supports de cours de TSI 2*, Lycée Rouvière, Toulon.