Chap. 3 Diviser pour régner

Certains algorithmes sont récursifs (on peut démontrer en logique que toute fonction est récursive et que tout algorithme l'est aussi).

1. Ils divisent le problème du tri en plusieurs sous-problèmes similaires mais de petites tailles.
2. Les sous problèmes sont traités de façon récursive (régner sur les sous-problèmes).
3. Ils combinent les solutions des sous-problèmes en une seule solution au problème posé.

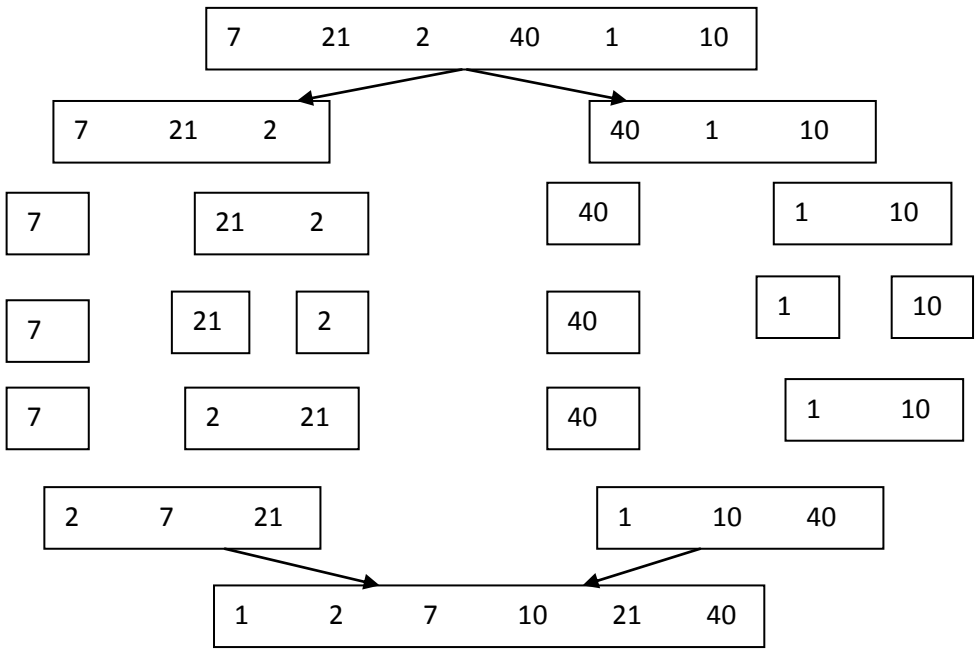
Rappel de la récursivité avec exemple :

```
3   fact <- function(n) {
4
5     if (n==0) return(1)
6     return(fact(n-1)*n)
7
8   }
9
10 # Appel de fact() et resultat
11
12  n=5
13  x=fact(n)
14  cat(n,"!  :", x)
15
16  # résultat :
17 # 5 !  : 120
```

Exemple du Tri par Fusion

1. Diviser le tableau à trier en 2 sous-tableau à peu près égaux
2. Régner : trier chacun des sous-tableaux
3. Combiner les deux sous-tableaux triés.

Lorsqu'un tableau a une taille = 1, il n'y a plus rien à faire (arrêt).



Pseudo-code :

```
1.  fonction fusion(gauche,droite){
2.      resultat = tableau
3.      index_gauche = index_droite = 0
4.      tant que index_gauche < longueur(gauche) and index_droite < longueur(droite):
5.          if gauche[index_gauche] <= droite[index_droite]
6.              ajouter a resultat (gauche[index_gauche])
7.              index_gauche = index_gauche + 1
8.          sinon
9.              ajouter a resultat  (droite[index_droite])
10.             index_droite = index_droite + 1
11.
12.     si longueur(gauche) > 0
13.         inserer dans resultat(gauche[index_gauche,..,fin])
14.     si longueur(droite) > 0
15.        inserer dans resultat(droite[index_droite,..,fin])
16.     retourner(resultat)
17.     }
18.
19. fonction tri_fusion(m){
20.     si longueur(m) <= 1:
21.         return(m)
22.    sinon {
23.       milieu = longueur(m)/2
24.       gauche = m[1,..,milieu]
25.       droite   = m[milieu+1,...,longueur(m)]
26.       gauche = tri_fusion(gauche)
27.       droite   = tri_fusion(droite)
28.       return list(fusion(gauche, droite))
29.     }
30. }
```

En Python:

```
1.  def fusion(gauche,droite):
2.      resultat = []
3.      index_gauche, index_droite = 0, 0
4.      while index_gauche < len(gauche) and index_droite < len(droite):
5.          if gauche[index_gauche] <= droite[index_droite]:
6.              resultat.append(gauche[index_gauche])
7.              index_gauche += 1
8.          else:
9.              resultat.append(droite[index_droite])
10.             index_droite += 1
11.     if gauche:
12.         resultat.extend(gauche[index_gauche:])
13.     if droite:
14.         resultat.extend(droite[index_droite:])
15.     return resultat
16.
17. def tri_fusion(m):
18.     if len(m) <= 1:
19.         return m
20.     milieu = len(m) // 2
21.     gauche = m[:milieu]
22.     droite = m[milieu:]
23.     gauche = tri_fusion(gauche)
24.     droite = tri_fusion(droite)
25.     return list(fusion(gauche, droite))
```

N'est pas implémenté en R (mais peut l'être facilement)

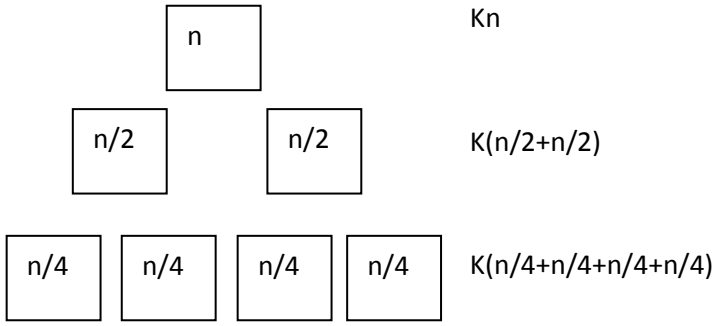
Complexité.

$T(n) = C + T(n/2) + T(n/2) + \theta(n) \leq Kn + 2T(n/2)$

avec C = temps du aux comparaisons et affectations

K = temps compté englobant C et $\theta(n)$. Comment résoudre cette équation (exprimer le temps en fonction de n seulement) ?

1. En examinant l'arbre de récursivité.



La hauteur de l'arbre est $\log_2(n)$ (ici on a ln(4)/ln(2) = 2 niveaux). A chaque niveau la somme est Kn, le coût total est donc de Knlog(n) => l'algorithme est $O(n\log(n))$

2. On peut réduire l'éq. à une éq. simple où l'éq. de la complexité pour n dépend seulement de la complexité de cette équation pour (n-1). L'idée est la suivante:

G(n) = G(n-1) + hn et G(1) = h

G(n-1) = G(n-2) + h(n-1)

G(n-2) = G(n-3) + h(n-2)

....

G(2) = G(1) + h2

G(1) = h

Donc, G(n) = hn + h(n-1) +...+ h2 + h et $G(n) = h\sum_{i=1}^{n} i$

Dans notre cas on a :

$$\begin{cases} T(n) = Kn + 2T(n/2) \\ T(1) = 1 \end{cases}$$

On élimine le 2 et on pose $n = 2^h$ => $h = \log_2(n)$. (car ln(n) = hln(2) et donc h = ln(n)/ln(2) = $\log_2(n)$)

$$\begin{cases} T(2^h) = K2^h + 2T(2^h/2) \\ T(2^0) = 1 \end{cases}$$

En divisant par 2^h :

$$\begin{cases} \frac{T(2^h)}{2^h} = \frac{K2^h}{2^h} + \frac{2T(2^{h-1})}{2^h} = K + \frac{T(2^{h-1})}{2^{h-1}} \\ T(2^0) = 1 \end{cases}$$

On pose $G(h) = \frac{T(2^h)}{2^h}$

$$\begin{cases} G(h) = G(h-1) + K = 1 + \sum_{i=1}^{h} K \cong Kh \\ G(0) = 1 \end{cases}$$

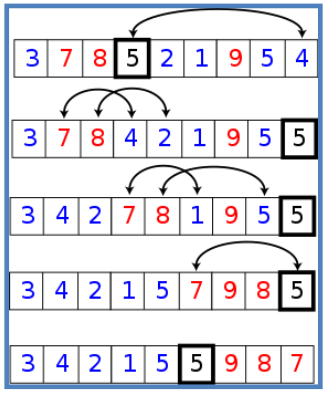
Donc : $\frac{T(2^h)}{2^h} = Kh$. Or $n = 2^h$ et donc : $\boxed{\frac{T(n)}{n} = Klog(n) \text{ et } T(n) = Knlog(n)}$

Ceci constitue la limite inférieure pour les algorithmes de tri ne faisant pas d'hypothèse sur l'entrée.

Algorithme quicksort

Inventé par C. Hoare en 1961(étudiant en stage à Moscou). Il est généralement utilisé sur des tableaux, mais peut aussi être adapté aux listes. Dans le cas des tableaux, c'est un tri en place mais non stable.

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. 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.



- Le pivot est placé à la fin (arbitrairement), en l'échangeant avec le dernier élément du sous-tableau ;
- Tous les éléments inférieurs au pivot sont placés en début du sous-tableau ;
- Le pivot est déplacé à la fin des éléments déplacés.

Il y a plusieurs possibilités de choix du pivot:

- aléatoirement une valeur entre min(tab[]) et max(tab[]). T(n) = $O(n\log n)$, $O(n^2)$ dans le pire des cas, mais en général il s'écarte peu de nlogn.
- Toujours choisir le premier ou le dernier élément de tab[]. T(n) = $O(n\log n)$ en général, $O(n^2)$ dans le pire des cas, mais celle-ci est atteinte même lorsque le tableau est presque trié.

Pseudocode

```
fonction partition(tab, gauche, droite){
  pivot := tab[droite]
  i := gauche
  pour j = gauche à (droite - 1) faire
   {
    si tab[j] < pivot alors
      si i <> j alors{
          echanger tab[i] avec tab[j]
          i := i + 1
      }
   }
  echanger A[i] avec A[droite]
  retourner i
}

fonction quicksort(A, lo, hi){
si lo < hi alors {
  p := partition(A, lo, hi)
  quicksort(A, lo, p - 1)
  quicksort(A, p + 1, hi)
  }
}

# on appelle quicksort avec le tableau et
# les valeurs 1 et taille du tableau
N = 10
pour i = 1 à N faire
    A[i] = random(15)
quicksort (A, 1,  N)
```

Code R

```
quicksort <- function(tab) {

# imprime le tab d'entree
  print(tab); flush.console()

  if(length(tab) > 1) {

    pivot <- tab[1]

    low <- quicksort(tab[tab < pivot])
    mid <- tab[tab == pivot]
    high <- quicksort(tab[tab > pivot])
    c(low, mid, high)

  }
  else
    tab # renvoie l'unique valeur restante
}
#----------------
# on tire au hasard les 10 premiers nombres
z <-sample(10)
z
v <-qs(z)
v
#---------------
```

C'est un algorithme en place mais non stable.

Tri non basé sur la comparaison des éléments

On fait des hypothèses sur le type d'entrée. Il s'agit par exemple de trier des valeurs entières. Soit le tableau suivant de k = 5 éléments :

k	1	2	3	4	5
Tab(k)	1	2	6	1	2

On établit le tableau de comptage (TabCompte) suivant :

K	0	1	2	3	4	5	6
TabCompte(k)	0	2	2	0	0	0	1

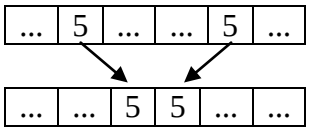
Pseudo-code. borneSup est la valeur entière maximale contenue dans tab

```
1  fonction TriComptage(tab, borneSup)
2    tabCompte[borneSup+1]  # a cause du 0
3    tailleTab = taille(tab -1)
4    x = 0
5  pour i = 0 à borneSup faire  #initialisation à 0
6    tabCompte[i=0]
7  finpour
8  pour i = 0 à tailleTab faire
9    tabCompte[tab[i]] = tabCompte[tab[i]] + 1
10 finpour
11 pour i = 0 à borneSup faire
12   pour j = 0 à tabCompte[i] faire
13     tab [x+1] = i
14   finpour
15 finpour
16 retourne (tab)
17
```

Implémentation en R

```
3  triCompte <- function(ttab, borne){
4
5    tailleTab <- length(ttab)
6    tabCompte <- vector(borne, mode= "numeric")
7    x <- 1
8
9  for (i in 1:borne)
10   tabCompte[i] <-0
11
12 for (i in 1:tailleTab)
13     tabCompte[ttab[i]] <- tabCompte[ttab[i]]+1
14
15 cat("tab compte: ",tabCompte, "\n")
16
17 for (i in 1:borne)
18   for (j in 1:tabCompte[i])
19     if (tabCompte[i]>0){
20     ttab[x] = i
21     x <- x+1
22    }
23 return(ttab)
24 }
25
26 tab = c(1, 2, 6, 1, 2, 8, 4)
27 cat("tab trie: ", triCompte(tab, max(tab)),"\n tab: ", tab)
28
```

C'est un algorithme stable. si tab[i]=tab[j] = p avec i<j alors tab[i] sera classé devant tab[j] en sortie:



Complexité de l'algorithme:
for i borne sup
for i tailleTab
for i borneSup
for j tabCompte

O(nbre élements, borneSup) = *O*(max,(n,borneSup)) . C'est donc un algorithme linéaire en temps. Mais ceci est dû à la restriction sur les entrées (valeurs entières) et à l'utilisation importante de la mémoire (2 tableaux dont 1 peut être très volumineux).

On a donc le classement suivant :

	optimal	moyen	pire	Stable
quicksort	nlogn	nlogn	n^2	Non
fusion	nlogn	nlogn	nlogn	Oui
insertion	n^2	n^2	n^2	Oui
Bulles	n	n^2	n^2	Oui