# Chapitre 6 : Diviser pour régner

La méthode algorithmique *« diviser pour régner »* (*« divide and conquer »* en anglais) consiste à ramener la résolution d'un problème dépendant d'un entier $n$ en un ou plusieurs problèmes identiques portant sur des entiers de l'ordre de $n/k$ avec $k > 1$. La plupart du temps, $k = 2$.

Le fonctionnement d'un tel algorithme est le suivant : - on divise le problème en un ou plusieurs sous-problèmes ; - on résout récursivement les sous-problèmes ; - on utilise les résultats obtenus pour construire la solution du problème initial.

Les algorithmes utilisant la dichotomies sont des exemples de tels algorithmes.

Nous allons dans ce chapitre étudier plusieurs autres algorithmes « diviser pour régner ».

## 1 Exponentiation rapide

La version récursive de l'exponentiation rapide est un exemple d'algorithme «diviser pour régner».

[1]:
```
let rec puissance x n =
  match n with
  | 0 -> 1
  | 1 -> x
  | _ -> if n mod 2 = 0
  then puissance (x*x) (n/2)
  else x * puissance (x*x) (n/2)
;;
```

[1]:
```
val puissance : int -> int -> int = <fun>
```

### 1.1 Terminaison

La fonction termine pour $n \in \{0, 1\}$. Pour $n \geqslant 2$, $\lfloor \frac{n}{2} \rfloor < n$, donc la fonction termine pour tout $n \in \mathbb{N}$.

### 1.2 Correction

La correction de l'algorithme est assurée par les égalités $x^0 = 1$, $x^1 = x$, et pour tout $k \in \mathbb{N}$, $x^{2k} = (x^2)^k$ et $x^{2k+1} = x.(x^2)^k$.

### 1.3 Complexité

Notons $c_n$ le nombre de multiplications effectuées lors de l'appel `puissance x n`. Alors $c_0 = 0$, $c_1 = 0$ et pour tout $n \geqslant 2$, $c_n = c_{\lfloor \frac{n}{2} \rfloor} + f(n)$ où $f(n) = 1$ si n est pair, 2 sinon.

Considérons la suite $u$ telle que $u_1 = 0$ et pour tout $n \geqslant 2$, $u_n = u_{\lfloor \frac{n}{2} \rfloor} + 2$. Alors pour tout $n \in \mathbb{N}^*$, $c_n \leqslant u_n$.

De plus, on peut montrer par récurrence forte que la suite $u$ est croissante.

Pour tout $p \in \mathbb{N}$, posons $v_p = u_{2^p}$ , alors pour tout $p \in \mathbb{N}$, $v_{p+1} = v_p + 2$, donc la suite $v$ est arithmétique de raison 2 : pour tout $p \in \mathbb{N}$, $u_{2^p} = v_p = 2p$.

Soit maintenant $n \in \mathbb{N}^*$ quelconque et $p = \lfloor \log_2 n \rfloor$. Alors $2^p \leqslant n < 2^{p+1}$.

Comme $u$ est croissante, $v_p \leqslant u_n \leqslant v_{p+1}$, donc $2p \leqslant u_n \leqslant 2p + 2$. Par encadrement, $u_n \sim 2 \log_2 n$.

Finalement, $c_n = O(\log n)$.

**Remarque :**

Dans ce cas particulier, on peut être plus précis en considérant l'écriture binaire de $n$ : si $n = \overline{b_p ... b_1 b_0}$, alors $c_n = c_{\lfloor \frac{n}{2} \rfloor} + 1 + b_0$ avec $\lfloor \frac{n}{2} \rfloor = \overline{b_p ... b_1}$

On montre donc aisément que $c_n = p + \sum_{k=0}^{p-1} b_k$, donc $p \leqslant c_n \leqslant 2p$, ce qui permet d'affirmer que $c_n = \Theta(\log n)$ (i.e que $c_n = O(\log n)$ et $\log n = O(c_n)$).

# 2 Tri fusion

Le tri fusion consiste à partager le tableau ou la liste à trier en deux parties de tailles respectives $\lceil \frac{n}{2} \rceil$ et $\lfloor \frac{n}{2} \rfloor$.

qu'on trie par un appel récursif, puis à fusionner les deux parties triées.

Comme il n'est pas aisé d'implémenter correctement la fusion en place dans le cas d'un tableau, nous allons étudier cet algorithme de tri sur les listes.

## 2.1 Partage de la liste

### 2.1.1 Implémentation

[2]:
```
let rec decoupe l =
  match l with
  | [] -> [], []
  | [x] -> [x], []
  | t1::t2::q -> let q1, q2 = decoupe q in
                 t1::q1, t2::q2
;;
```

[2]:
```
val decoupe : 'a list -> 'a list * 'a list = <fun>
```

### 2.1.2 Terminaison et correction

On peut montrer par récurrence double sur $n$ que pour toute liste `l` de longueur $n$, l'appel `decoupe l` termine en effectuant $1 + \lfloor \frac{n}{2} \rfloor$ appels à `decoupe`, et que cet appel renvoie deux listes $l_1$ et $l_2$ de longueurs respectives $\lceil \frac{n}{2} \rceil$ et $\lfloor \frac{n}{2} \rfloor$.

### 2.1.3 Complexité

Chaque appel de la fonction decoupe effectue uniquement un nombre borné d'opérations, toutes en temps constant, sauf l'éventuel appel récursif.

Par conséquent, l'exécution de `decoupe l` prend un temps en $O(n)$.

## 2.2 Fusion de listes triées

### 2.2.1 Implémentation

[3]:
```
let rec fusion l1 l2 =
  match (l1, l2) with
  | _, [] -> l1
  | [], _ -> l2
  | t1::q1, t2::q2 -> if t1<t2
                      then t1::(fusion q1 l2)
                      else t2::(fusion l1 q2)
;;
```

[3]:
```
val fusion : 'a list -> 'a list -> 'a list = <fun>
```

### 2.2.2 Correction

Montrons que pour toutes listes $l_1$ et $l_2$ triées par ordre croissant, l'appel `fusion` $l_1$ $l_2$ termine et retourne une liste $l$ triée par ordre croissant dont les éléments sont les mêmes que ceux de $l_1$ `@` $l_2$ (où `@` est l'opérateur de concaténation).

Pour cela, notons pour $n \in \mathbb{N}$, $\mathcal{P}(n)$ l'assertion « pour toutes listes $l_1$ et $l_2$ triées par ordre croissant dont la somme des longueurs vaut $n$, la fonction `fusion` $l_1$ $l_2$ termine et retourne une liste $l$ triée par ordre croissant dont les éléments sont les mêmes que ceux de $l_1$ `@` $l_2$ ».

- Montrons $\mathcal{P}(0)$. Soit $l_1$ et $l_2$ deux listes dont la somme des longueurs vaut 0, alors $l_1$ et $l_2$ sont vides. Or `fusion [] []` renvoie `[]` qui possède les mêmes éléments que `[]@[]` et qui est triée. $\mathcal{P}(0)$ est donc vérifiée.
- Soit $n \in \mathbb{N}$ quelconque. Supposons que $\mathcal{P}(n)$ est vérifiée et montrons $\mathcal{P}(n+1)$ . Soient $l_1$ et $l_2$ deux listes triées par ordre croissant dont la somme des longueurs vaut $n+1$.
  - Si $l_1$ est vide, `fusion` $l_1$ $l_2$ retourne $l_2$, qui est triée par ordre croissant et possède les mêmes éléments que $l_1$ `@` $l_2$.
  - Si $l_2$ est vide, on a le même résultat.
  - Sinon, $l_1$ et $l_2$ sont toutes deux non vides.
    * Si $t_1 < t_2$, alors `fusion` $l_1$ $l_2$ retourne le résultat de l'évaluation de $t_1$ :: `fusion` $q_1$ $l_2$. Or $q_1$ et $l_2$ sont triées et la somme des longueurs de $q_1$ et $l_2$ vaut n. Par hypothèse de récurrence, `fusion` $q_1$ $l_2$ renvoie une liste l triée ayant les mêmes éléments que $q_1$ `@` $l_2$. Par conséquent, $t_1$ :: `fusion` $q_1$ $l_2$ retourne une liste ayant les mêmes éléments que $t_1$::$q_1$ `@` $l_2$, donc que $l_1$ `@` $l_2$. De plus, $l_1$ et $l_2$ sont triées donc $t_1$ minore tous les éléments de $q_1$ et $t_2$ tous ceux de $l_2$. Comme de plus $t_1 < t_2$, $t_1$ minore tous les éléments de $q_1$ `@` $l_2$, donc de l, qui est triée. Donc $t_1$::$l$ est triée. L'appel `fusion` $l_1$ $l_2$ termine et retourne une liste triée contenant les mêmes éléments que $l_1$ `@` $l_2$.

* Si $t_1 \geqslant t_2$, on montre de même que l'appel `fusion` $l_1$ $l_2$ termine et renvoie une liste triée contenant les mêmes éléments que $l_1$ `@` $l_2$.

$\mathcal{P}$ est donc héréditaire.

On en déduit que $P(n)$ est vrai pour tout entier $n$.

#### 2.2.3 Complexité

À chaque appel de fusion, on effectue uniquement un nombre borné d'opérations, qui sont toutes de temps constant, sauf l'appel récursif. On en déduit que la complexité temporelle de l'exécution de `fusion` $l_1$ $l_2$ est un $O(n)$ où $n$ est la somme des longueurs de $l_1$ et $l_2$.

### 2.3 Tri fusion

#### 2.3.1 Implémentation

[4]:
```
let rec tri_fusion l =
  match l with
  | [] -> []
  | [x] -> [x]
  | _ -> let l1, l2 = decoupe l in
  fusion (tri_fusion l1) (tri_fusion l2)
;;
```

[4]:
```
val tri_fusion : 'a list -> 'a list = <fun>
```

#### 2.3.2 Correction

Pour $n \in \mathbb{N}$, notons $\mathcal{P}(n)$ l'assertion « Pour une liste $l$ de $n$ éléments, `tri_fusion` $l$ renvoie une liste triée par ordre croissant ayant les mêmes éléments que $l$ », et raisonnons par récurrence forte sur $n$.

$\mathcal{P}(0)$ et $\mathcal{P}(1)$ sont vraies.

Soit $n \geqslant 2$. Supposons $\mathcal{P}(k)$ vraie pour tout $k < n$. Soit l une liste de longueur n. Notons $l_1$ et $l_2$ les deux listes obtenues par l'appel `decoupe` l. Alors $l_1$ et $l_2$ sont respectivement de longueur $\lceil \frac{n}{2} \rceil$ et $\lfloor \frac{n}{2} \rfloor$. Comme $n \geqslant 2$, $n > \lceil \frac{n}{2} \rceil \geqslant \lfloor \frac{n}{2} \rfloor$, donc pour $i \in \{1, 2\}$, `tri_fusion` $l_i$ renvoie une liste $l'_i$ triée ayant les mêmes éléments que $l_i$.

Par conséquent, la liste renvoyée par `tri_fusion` $l$ est une liste triée (car `fusion` est correcte) qui possède exactement les mêmes éléments que $l'_1$ `@` $l'_2$

#### 2.3.3 Complexité

Notons, pour $n \in \mathbb{N}$, $C(n)$ le temps de calcul mis dans le pire des cas par `tri_fusion` $l$ pour une liste $l$ de longueur $n$.

Les opérations effectuées par `tri_fusion` sur une liste $l$ de longueur $n \geqslant 2$ sont un filtrage (en temps constant), un appel à `decoupe` $l$ en temps $O(n)$, un appel à `fusion` sur deux listes $l'_1$ et $l'_2$ de longueurs respectives $\lceil n/2 \rceil$ et $\lfloor n/2 \rfloor$, ce qui demande un temps $O(\lceil n/2 \rceil + \lfloor n/2 \rfloor) = O(n)$, et deux appels récursifs à `tri_fusion` sur des listes de longueurs $\lceil n/2 \rceil$ et $\lfloor n/2 \rfloor$, dont les temps de calcul sont donc au plus respectivement $C(\lceil n/2 \rceil)$ et $C(\lfloor n/2 \rfloor)$.

Le temps de calcul de `tri_fusion` $l$ est donc au plus $O(n) + C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor)$.

Cette inégalité étant vérifiée pour toute liste de longueur $n$, le temps de calcul de `tri_fusion` sur une liste de longueur $n$ dans le cas le pire vérifie:

$$C(n) \leqslant C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n)$$

Ou bien on suppose que $C$ est croissante, ou bien on introduit plutôt $C'(n)$, le temps de calcul de `tri_fusion` dans le cas le pire pour une liste de longueur *au plus* $n$.

Pour tout $n \in \mathbb{N}$, $C(n) \leqslant C'(n) \leqslant C'(n+1)$, car pour toute liste de longueur $n$, `tri_fusion` met un temps au plus égal à $C'(n)$ et pour toute liste de longueur au plus $n$, `tri_fusion` met un temps au plus $C'(n+1)$.

Donc $C'$ majore $C$ et est croissante.

De plus, on a:

$$C'(n) \leqslant C'\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C'\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n)$$

Pour $k \in \mathbb{N}$, posons $u_k = C'(2^k)$.

Alors, à partir d'un certain rang, $u_k \leqslant 2u_{k-1} + \alpha 2^k$ où $\alpha$ est une constante.

On pose alors pour tout $k \in \mathbb{N}$, $v_k = \frac{u_k}{2^k}$, de sorte qu'à partir d'un certain rang $k_0$, $v_k \leqslant v_{k-1} + \alpha$, d'où $v_k - v_{k-1} \leqslant \alpha$.

Alors pour tout $k \geqslant k_0$, $v_k - v_{k_0} \leqslant (k - k_0)\alpha$, donc $v_k \leqslant (k-k_0)\alpha + v_{k_0}$. Par conséquent, $v_k = O(k)$, donc $u_k = O(k2^k)$.

Soit $n \in \mathbb{N}^*$ et $p = \lceil \log_2 n \rceil$, alors $C(n) \leqslant C'(n) \leqslant C'(2^p)$ avec $C'(2^p) = u_p = O(p2^p)$.

Or $O(\lceil \log_2 n \rceil 2^{\lceil \log_2 n \rceil}) = O(n \log n)$, donc

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

## 3 Tri par pivot

Ce tri est aussi appelé *tri rapide* (ou *quicksort*), mais ce nom pourrait vous induire en erreur : le tri par pivot, s'il est rapide dans les meilleurs cas, se comporte mal dans le pire des cas.

Ce tri consiste, lorsque la liste $l$ à trier est assez grande à effectuer les étapes suivantes :

- On choisit dans $l$ un élément quelconque $p$, appelé *pivot*, et on construit deux listes $l_1$ et $l_2$ des autres éléments de $l$ contenant respectivement les éléments inférieurs ou égaux à $p$ et ceux strictement supérieurs à $p$.
- On trie récursivement les deux listes $l_1$ et $l_2$, ce qui donne deux listes triées $l_1'$ et $l_2'$.
- On renvoie la liste $l_1'$ `@` $(p{::}l_2')$.

### 3.1 Partition

La fonction `filter : ('a -> bool) -> 'a list -> 'a list` du module `List` prend en argument une fonction `f : 'a -> bool` et une liste `lst` et renvoie la liste des éléments de `lst` pour lesquelles la fonction `f` renvoie `true`. Sa complexité (temporelle et spatiale) est un $O(n)$.

On en déduit une fonction `partition` de complexité linéaire :

[5]:
```
let partition p l =
  let l1 = List.filter (fun x -> x <= p) l in
  let l2 = List.filter (fun x -> x > p) l in
  l1, l2
;;
```

[5]:
```
val partition : 'a -> 'a list -> 'a list * 'a list = <fun>
```

### 3.2 Tri par pivot

#### 3.2.1 Implémentation

[6]:
```
let rec quicksort l =
  match l with
  | [] -> []
  | t::q -> let l1, l2 = partition t q in
            let l1t = quicksort l1 in
            let l2t = quicksort l2 in
            l1t@(t::l2t)
;;
```

[6]:
```
val quicksort : 'a list -> 'a list = <fun>
```

#### 3.2.2 Complexité

La fonction `partition` est de complexité linéaire, donc il existe deux constantes $\alpha$ et $\beta$ telles que pour tout $p$ et pour toute liste $l$, le temps de calcul de `partition` $p$ $l$ est majoré par $\alpha|l| + \beta$ où $|l|$ désigne la longueur de la liste $l$

Le temps de calcul de `quicksort` $l$ avec $l$ non vide de longueur $n$ est égal à la somme des coûts du partitionnement, des deux appels récursifs, puis de la concaténation, plus quelques coûts en temps constant.

Le coût en temps de la partition est majoré par $\alpha(n-1) + \beta$.

Le coût de la concaténation est linéaire par rapport à la taille de $v_1$, qui est majorée par $n-1$ donc il existe deux constantes $\lambda, \mu$ telles que le coût de la concaténation est majorée par $\lambda(n-1) + \mu$.

Finalement, il existe des constantes $a$ et $b$ telles que le temps de calcul de `quicksort` $l$ est majoré par $an + b$ plus le temps de calcul des appels récursifs.

Notons $N$ le nombre total d'appels à `quicksort` et $S$ la somme des longueurs des listes sur lesquelles s'effectuent ces appels. Alors le temps de calcul de `quicksort` $l$ est majoré par $aS + bN$, donc par $m(S + N)$ où $m = \max\{a, b\}$.

Pour tout $n \in \mathbb{N}$, notons $C(n)$ la valeur de $S+N$ dans le pire des cas pour une liste de longueur $n$.

- $C(0) = 1$, car si la liste est vide, on réalise un seul appel.
- Soit $n \in \mathbb{N}^*$. Alors il existe $k \in [\![0, n-1]\!]$ tel que $C(n) \leqslant C(k) + C(n-1-k) + n + 1$.

Montrons par récurrence forte que $C(n) \leqslant (n+1)^2$.

- $C(0) \leqslant (0+1)^2$.
- Soit $n \in \mathbb{N}^*$. On suppose que pour tout $k \in [\![0, n-1]\!]$, $C(k) \leqslant (k+1)^2$.
  Comme il existe $k \in [\![0, n-1]\!]$ tel que $C(n) \leqslant C(k) + C(n-1-k) + n + 1$, avec $k \leqslant n-1$ et $n-1-k \leqslant n-1$, on en déduit que $C(n) \leqslant (k+1)^2 + (n-k)^2 + n + 1$.

  $$\begin{aligned}\text{Or } (k+1)^2 + (n-k)^2 + n + 1 &= k^2 + 2k + 1 + n^2 - 2nk + k^2 + n + 1 \\ &= n^2 + 2n + 1 + 2k^2 + 2k - 2nk - n + 1 \\ &= (n+1)^2 - 2k(n-1-k) - (n-1)\end{aligned}$$

  donc $C(n) \leqslant (n+1)^2$

Par conséquent, la complexité temporelle de `quicksort` $l$ est, dans le pire des cas, en $O(n^2)$.

En considérant le cas où la liste est déjà triée, on peut montrer que cette borne est atteinte.

# 4 Exercices

## 4.1 Exercice 1

Écrire un tri fusion qui trie suivant l'ordre lexicographique un tableau dont les éléments sont des couples de flottants. On pourra utiliser la commande `Array.sub t i n` renvoyant le sous-tableau de `t` commençant à l'indice `i` et de longueur `n`. On rappelle que l'ordre lexicographique est utilisé par défaut par OCaml pour comparer des couples de flottants :

[7]:
```
(4,2)<(2,3);;
(2,2)<(2,3);;
```

[7]:
```
- : bool = false
- : bool = true
```

## 4.2 Exercice 2 : distance minimale entre les points d'un nuage de points

On se donne un tableau de taille $n$ contenant des couples de flottants représentant un nuage de points du plan, que l'on suppose deux à deux distincts. On souhaite déterminer les deux points les plus proches.

1) Quelle est la complexité de l'algorithme consistant à considérer tous les couples de points ?

Si le nuage comporte peu de points, on utilisera l'algorithme naïf. Dans le cas contraire, on va appliquer une stratégie *« diviser pour régner »*. On sépare le nuage de points $P$ en deux parties $P_G$ et $P_D$ approximativement de mêmes tailles autour d'un axe vertical d'équation $x = \ell$.

La distance minimale entre deux points de $P$ est donc atteinte :

- soit entre deux points de $P_G$ ;
- soit entre deux points de $P_D$ ;

- soit entre un point de $P_G$ et un point de $P_D$.

On calcule récursivement la distance minimale $\delta_G$ séparant les points du nuage $P_G$ et la distance minimale $\delta_D$ séparant les points du nuage $P_D$.

On pose ensuite $\delta = \min\{\delta_G, \delta_D\}$.

2) Quelle serait la complexité de l'algorithme si on calcule les distances entre tous les couples de points constitués d'un point de $P_G$ et d'un point de $P_D$ ?

3) Montrer que si la distance minimale est atteinte dans le troisième cas, alors elle l'est entre deux points dont les abscisses appartiennent à $[\ell - \delta, \ell + \delta]$.

Notons $B$ l'ensemble des points de $P$ dont les abscisses appartiennent à $[\ell - \delta, \ell + \delta]$.

4) Soit $M(x_M, y_M)$ un point de $B$. Montrer qu'il existe au plus sept autres points $N(x, y)$ de $B$ tels que $y_M \leqslant y \leqslant y_M + \delta$.

Pour déterminer si deux points de $B$ sont distants de moins de $\delta$, il suffit donc de calculer la distance entre chaque point de $B$ et les sept suivants par ordonnée croissante.

Pour séparer le nuage en deux, il est préférable que $P$ soit trié par abscisse croissante, mais pour la dernière étape, il faudrait disposer des points triés par ordonnée croissante. Pour éviter d'avoir à trier les sous-tableaux à chaque appel récursif, on introduit de la redondance : on prendra en entrée deux tableaux $x$ et $y$, contenant tous les deux les mêmes couples de points. Les éléments de $x$ seront triés selon l'ordre lexicographique en prenant d'abord en compte les abscisses, et ceux de $y$ seront triés selon l'ordre lexicographique en prenant d'abord en compte les ordonnées. Ainsi, dans une première étape, on coupera $x$ en deux : les deux sous-tableaux obtenus $x_1$ et $x_2$ seront encore triés lexicographiquement par abscisse. On coupera alors $y$ en deux sous-tableaux, en répartissant les éléments suivants qu'ils sont strictement inférieurs ou supérieurs ou égaux au 1er élément de $x_2$. On obtient deux sous-tableaux $y_1$ et $y_2$ qui sont encore trisé lexicographiquement par ordonnée. Les éléments du nuage étant deux à distincts, on peut montrer que les points de $x_1$ et $y_1$ sont les mêmes, ainsi que ceux de $y_1$ et $y_2$. Finalement, nous n'aurons jamais à effectuer de tri, ni au début car les tableaux donnés $x$ et $y$ sont supposés triés, ni lors des appels récursifs.

Enfin, les tableaux à 1, 2 ou 3 éléments sont pénibles à manipuler pour l'algorithme diviser pour régner : pour les trier on utilisera l'algorithme naïf.

5) Proposer un algorithme qui renvoie la plus petite distance ainsi qu'un couple de points la réalisant, et évaluer sa complexité. On pourra utiliser la commande `Array.of_list l` qui convertit une liste `l` en tableau.