Programmation fonctionnelle

Cours n°4 : Récursivité en profondeur, algorithmes de tri.

Licence 1 Informatique, Université Paris 8

10 Octobre 2022

I. Récursivité en profondeur

Commençons par un exercice

- On veut écrire une fonction **llistes** prenant en paramètres une liste L, et renvoyant la liste de toutes les listes contenues dans L.
- Par exemple,

```
(llistes '(1 2 '(3 4) #t '(7 #f 0)))    '('(3 4) '(7 #f 0))
```

Commençons par un exercice

- On veut écrire une fonction **llistes** prenant en paramètres une liste L, et renvoyant la liste de toutes les listes contenues dans L.
- Par exemple,

```
(llistes '(1 2 '(3 4) #t '(7 #f 0)))    '('(3 4) '(7 #f 0))
```

- Si la liste est vide, elle ne contient pas de sous-liste à l'intérieur, et on peut renvoyer une liste vide.
- On teste si le premier élément est une liste. Si oui, on va l'ajouter au résultat de l'appel récursif de **lliste** sur le reste.
- Sinon, on oublie le premier élément (on ne le veut pas dans le résultat final) et on renvoie l'appel récursif de **lliste** sur le reste.

Commençons par un exercice

- On veut écrire une fonction **llistes** prenant en paramètres une liste L, et renvoyant la liste de toutes les listes contenues dans L.
- Par exemple,

```
(llistes '(1 2 '(3 4) #t '(7 #f 0)))    '('(3 4) '(7 #f 0))
```

- Si la liste est vide, elle ne contient pas de sous-liste à l'intérieur, et on peut renvoyer une liste vide.
- On teste si le premier élément est une liste. Si oui, on va l'ajouter au résultat de l'appel récursif de **lliste** sur le reste.
- Sinon, on oublie le premier élément (on ne le veut pas dans le résultat final) et on renvoie l'appel récursif de **lliste** sur le reste.

```
1 #lang racket
2 (define llistes
3   (lambda (L)
4     (if (null? L)
5         L
6         (if (list? (car L))
7             (cons (car L) (llistes (cdr L)))
8             (llistes (cdr L))))))
```

Récursivité en profondeur

- On a vu précédemment qu'une liste non plate est une liste qui contient des éléments qui sont eux-mêmes des listes.
- Une fonction parcourt **récursivement en profondeur** une liste L si elle s'applique pour **chaque sous-liste** l de L de la même manière qu'elle s'applique sur L, et ceci de manière récursive.
- On a ainsi deux niveaux de récursivité :
  - Le premier, usuel, sur la structure de L;
  - Le second sur tous les éléments de L qui sont des listes.
- **Exemple 1 : Somme des éléments d'une liste.**

```
1  #lang racket
2  (define (sommeListe L)
3    (cond ((null? L) 0)
4          ((number? (car L))
5             (+ (car L) (sommeListe (cdr L))))
6          (else (sommeListe (cdr L)))))
```

et si l'on teste :

```
9  (sommeListe '(1 2 5 #t 8 'a 'c))              16
10 (sommeListe '(1 2 5 #t '(2 4 6) 'a #f))       8
```

Récursivité en profondeur

- Ainsi, la fonction n'ajoute pas les valeurs qui sont contenues dans la sous-liste '(2 4 6) de la deuxième liste.

- Pour résoudre ce problème, il faut se demander (hors de l'appel récursif) que faire si **(car L)** est lui-même une liste ? Il faut ajouter la somme des valeurs contenues dans **(car L)** à la somme des valeurs de l'appel récursif !

```
(+ (somme (car L)) (somme (cdr L))
```

Récursivité en profondeur

- Ainsi, la fonction n'ajoute pas les valeurs qui sont contenues dans la sous-liste '(2 4 6) de la deuxième liste.

- Pour résoudre ce problème, il faut se demander (hors de l'appel récursif) que faire si **(car L)** est lui-même une liste ? Il faut ajouter la somme des valeurs contenues dans **(car L)** à la somme des valeurs de l'appel récursif !

$$(+ \text{ (somme (car L)) (somme (cdr L))}$$

- **Exemple 2 :** Essayons donc:

```
12 (define (sommeListeProf L)
13   (cond ((null? L) 0)
14         ((number? (car L))
15             (+ (car L) (sommeListeProf (cdr L))))
16         ((list? (car L))
17             (+ (sommeListeProf (car L)) (sommeListeProf (cdr L))))
18         (else (sommeListeProf (cdr L)))))
```

et le test:

```
20 (sommeListeProf '(1 2 5 #t 8 'a 'c))              16
21 (sommeListeProf '(1 2 5 #t '(2 4 6) 'a #f))       20
```

- C'est mieux ! Attention toutefois à ne pas supprimer certains cas de la fonction précédente (notamment lorsque **(car L)** est un nombre); il faut bien toujours faire quelque chose dans ce cas !

Un exemple : reprogrammons *flatten*

- On veut écrire une fonction **aplatit** qui aplatit une liste, c'est à dire elle supprime toutes les parenthèses internes d'une liste L.

- Voici la démarche:
  - Si la liste est vide, on renvoie la liste vide;
  - Si **(car L)** n'est pas une liste, on ne s'en préoccupe pas et on le rajoute au début de l'appel récursif de **aplatit** sur le reste avec un **cons**.
  - Si **(car L)** est une liste, il faut déjà aplatir cette liste, puis tout ajouter au début de l'appel récursif de **aplatit** sur le reste. Le mieux est d'utiliser un **append**.

```
1 #lang racket
2 (define (aplatit L)
3   (cond ((null? L) '())
4         ((list? (car L)) (append (aplatit (car L))
5                                  (aplatit (cdr L))))
6         (else (cons (car L)) (aplatit (cdr L)))))
```

II. Algorithmes de tri

Algorithmes de tri

- Le principe est le suivant: on dispose d'une liste de nombres (supposés tous différents), par exemple

$$'(5\ 2\ 14\ 1\ 6)$$

que l'on veut trier, c'est-à-dire que l'on veut remettre les éléments dans l'ordre croissant.

- Sur l'exemple, nous voudrions renvoyer

$$'(1\ 2\ 5\ 6\ 14)$$

Algorithmes de tri

- Le principe est le suivant: on dispose d'une liste de nombres (supposés tous différents), par exemple

$$'(5\ 2\ 14\ 1\ 6)$$

que l'on veut trier, c'est-à-dire que l'on veut remettre les éléments dans l'ordre croissant.

- Sur l'exemple, nous voudrions renvoyer

$$'(1\ 2\ 5\ 6\ 14)$$

- Il existe de nombreux algorithmes permettant d'obtenir ce résultat, ici nous parlerons de:
  - **Tri par sélection** (du minimum ou du maximum);
  - **Tri à bulles**;
  - **Tri par insertion**;
  - **Tri fusion**;
  - **Tri rapide**.

Tri par sélection (du minimum)

- **Principe :** On va rechercher le minimum de la liste et le supprimer, puis recommencer avec le reste de la liste.

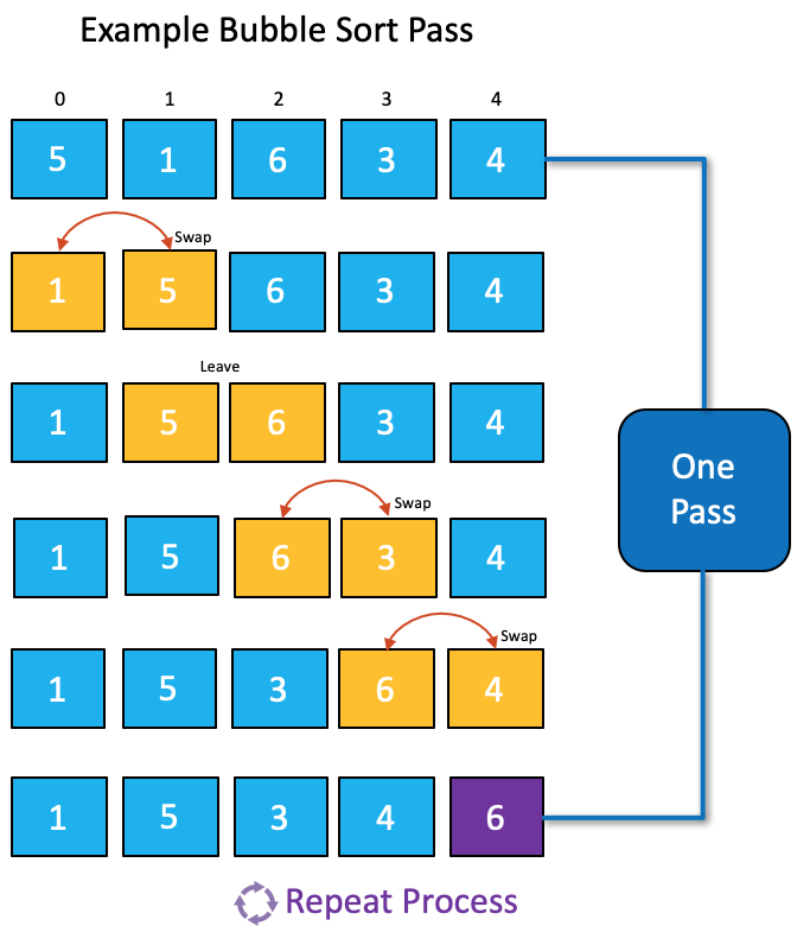
- Pour ce faire, on a besoin de deux fonctions auxiliaires: une fonction **minimum**, et une fonction **enleve**.

```
1 | #lang racket
2 | (define (minimum L)
3 |   (if (null? (cdr L))
4 |       (car L)
5 |       (if (< (car L) (minimum (cdr L)))
6 |           (car L)
7 |           (minimum (cdr L)))))
```

```
1 | #lang racket
2 | (define (enleve x L)
3 |     (cond ((null? L) L)
4 |           ((= x (car L)) (cdr L))
5 |           (else (cons (car L) (enleve x (cdr L))))))
```

Tri à bulles

- **Principe :** On va parcourir la liste de droite à gauche en comparant un élément avec son successeur, et en les échangeant si le précédent est plus grand que le successeur.



Tri à bulles II

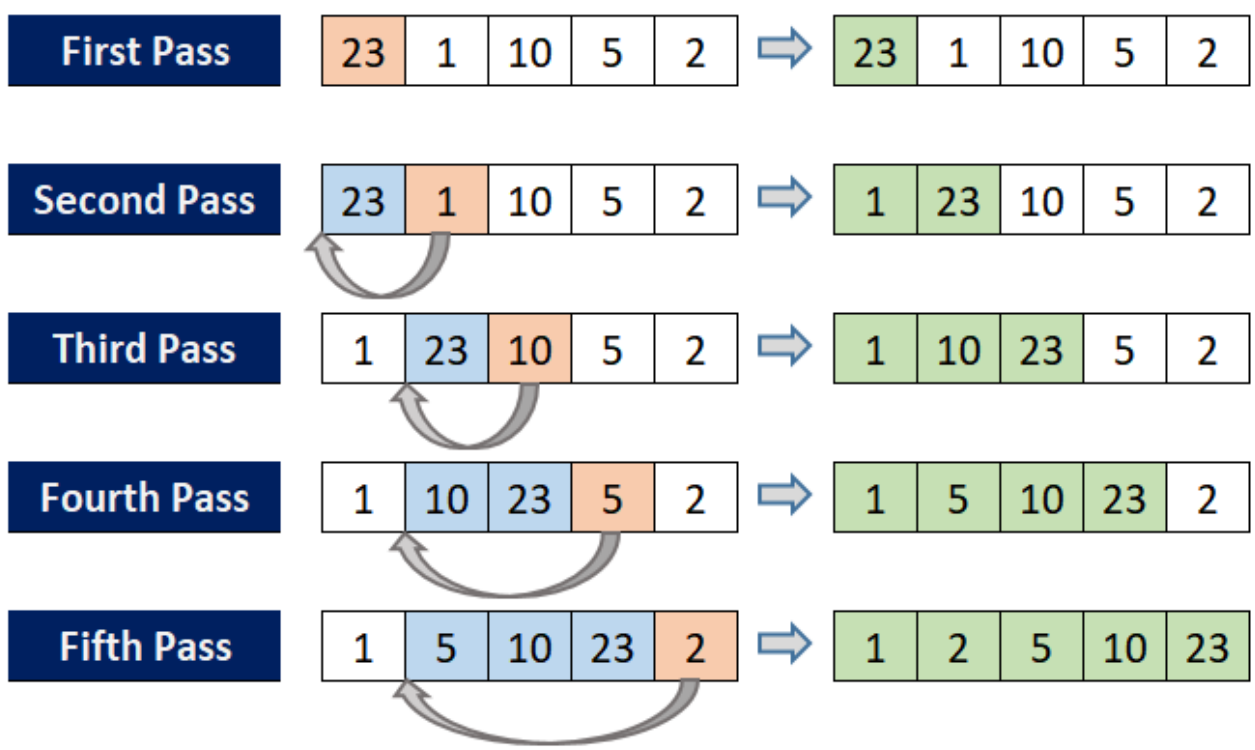
- De manière récursive, on va utiliser une fonction **bulle**, qui cherche le minimum et le fait remonter en début de liste.

À vous de jouer !

- Une fois que la bulle est remontée en première place, il reste à conserver cet élément en première place, et à le rajouter dans l'appel récursif de *tri-bulle* sur le reste de la liste.

Tri par insertion

- **Principe :** On va insérer les valeurs successivement dans l'ordre à leur « bonne » place dans la sous-liste qui les précèdent.
  - On suppose que le premier élément est à sa place, et on va placer le deuxième élément : avant le premier si il est plus petit, après sinon.
  - Ensuite, on va placer le troisième élément où il faut dans la sous-liste précédente contenant les deux premiers éléments dans l'ordre, etc.

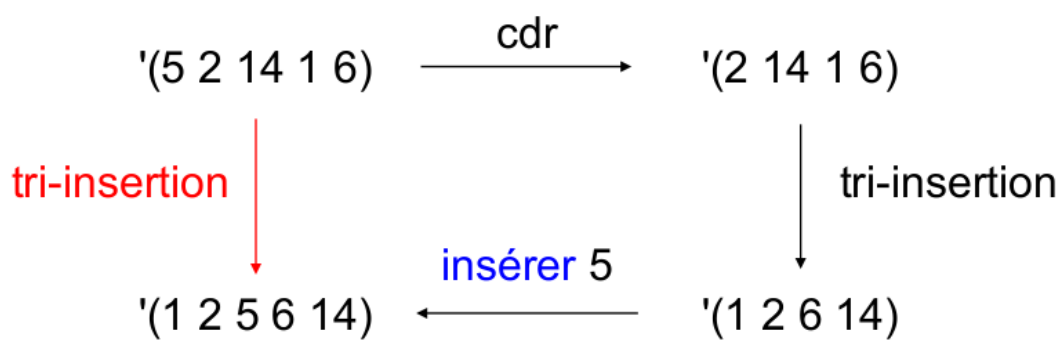


Tri par insertion II

- On a besoin d'une fonction **insere**, qui insère un élément au bon endroit dans une liste déjà triée.

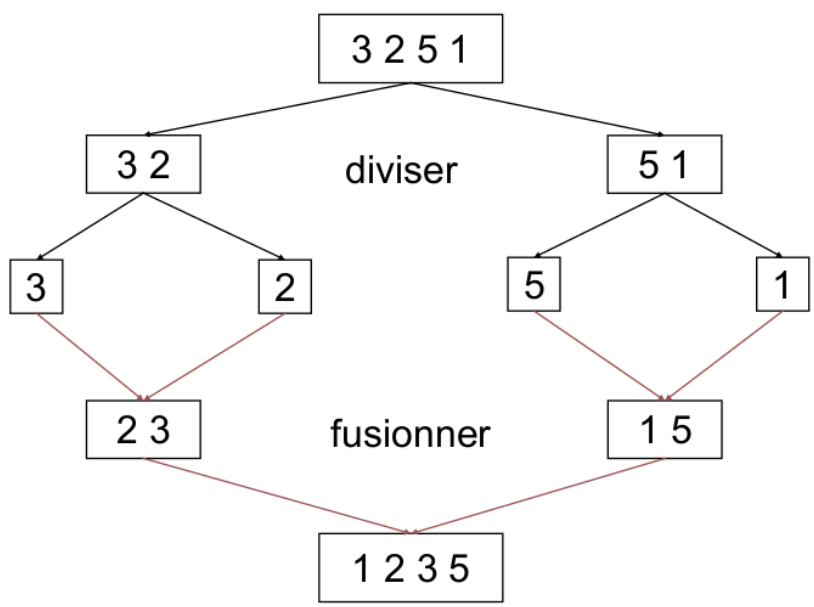
À vous de jouer !

- Ensuite, on va insérer le premier élément de notre liste dans l'appel récursif de *tri-insertion* sur le reste:



Tri fusion

- **Principe :** Algorithme utilisant le paradigme **Diviser pour régner**, consistant à:
  - **Diviser** le problème en un certain nombre de sous-problèmes (en général pour des listes, diviser une liste en deux sous-listes de taille équivalente);
  - **Régner** sur les sous-problèmes en les résolvant récursivement;
  - **Combiner** les solutions des sous-problèmes en une solution pour le problème initial.
- Pour le tri fusion, on va diviser une liste de n éléments en deux sous-listes de $n/2$ éléments, trier ces deux sous-listes récusrivement par tri fusion, et combiner les deux sous-listes triées.



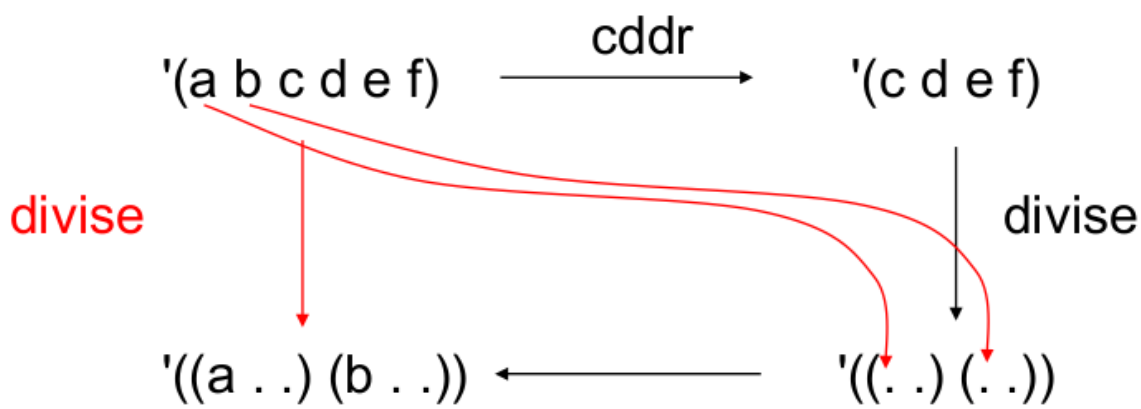
Tri fusion II

- Nous avons donc 3 fonctions à définir:
  - Une fonction **divise**, qui divise la liste en deux sous-listes;
  - Une fonction **fusion**, qui fusionne deux listes triées en une liste globale triée.
  - La fonction **tri-fusion**, qui réalise le tri global de la liste.

Tri fusion II

- Nous avons donc 3 fonctions à définir:
  - Une fonction **divise**, qui divise la liste en deux sous-listes;
  - Une fonction **fusion**, qui fusionne deux listes triées en une liste globale triée.
  - La fonction **tri-fusion**, qui réalise le tri global de la liste.

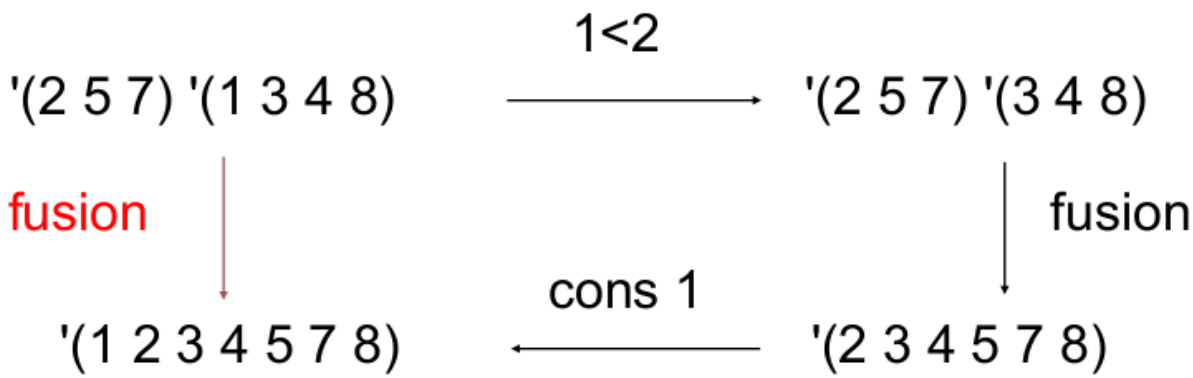
- Fonction **divise**:




```
1 #lang racket
2 (define (divise L)
3   (cond ((null? L) '(() ()))
4         ((null? (cdr L)) (list L '()))
5         (else (let ((r (divise (cddr L))))
6                 (list (cons (car L) (car r)) (cons (cadr L) (cadr r)))))))
```

Tri fusion III

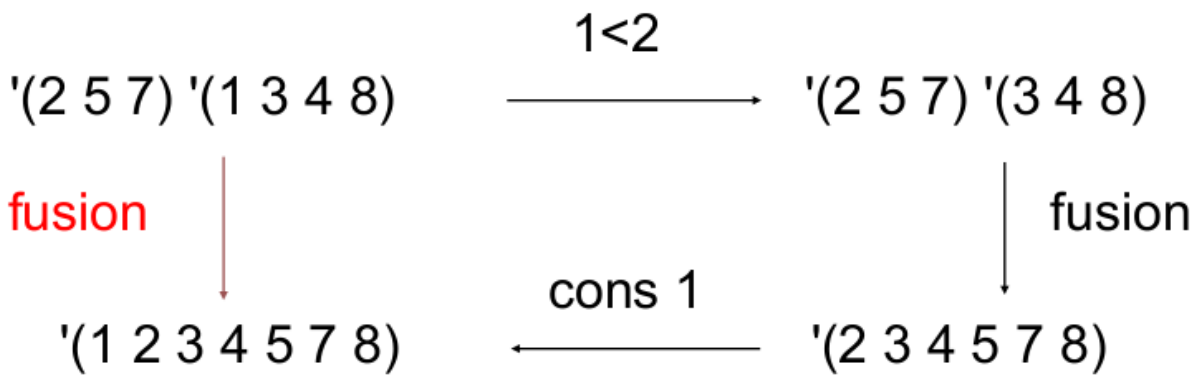
- Fonction **fusion**:




```
1 #lang racket
2 (define (fusion L1 L2)
3   (cond ((null? L1) L2)
4         ((null? L2) L1)
5         ((< (car L1) (car L2))
6             (cons (car L1) (fusion (cdr L1) (L2)))
7         (else
8             (cons (car L2) (fusion L1 (cdr L2))))))))
```

Tri fusion III

- Fonction **fusion**:



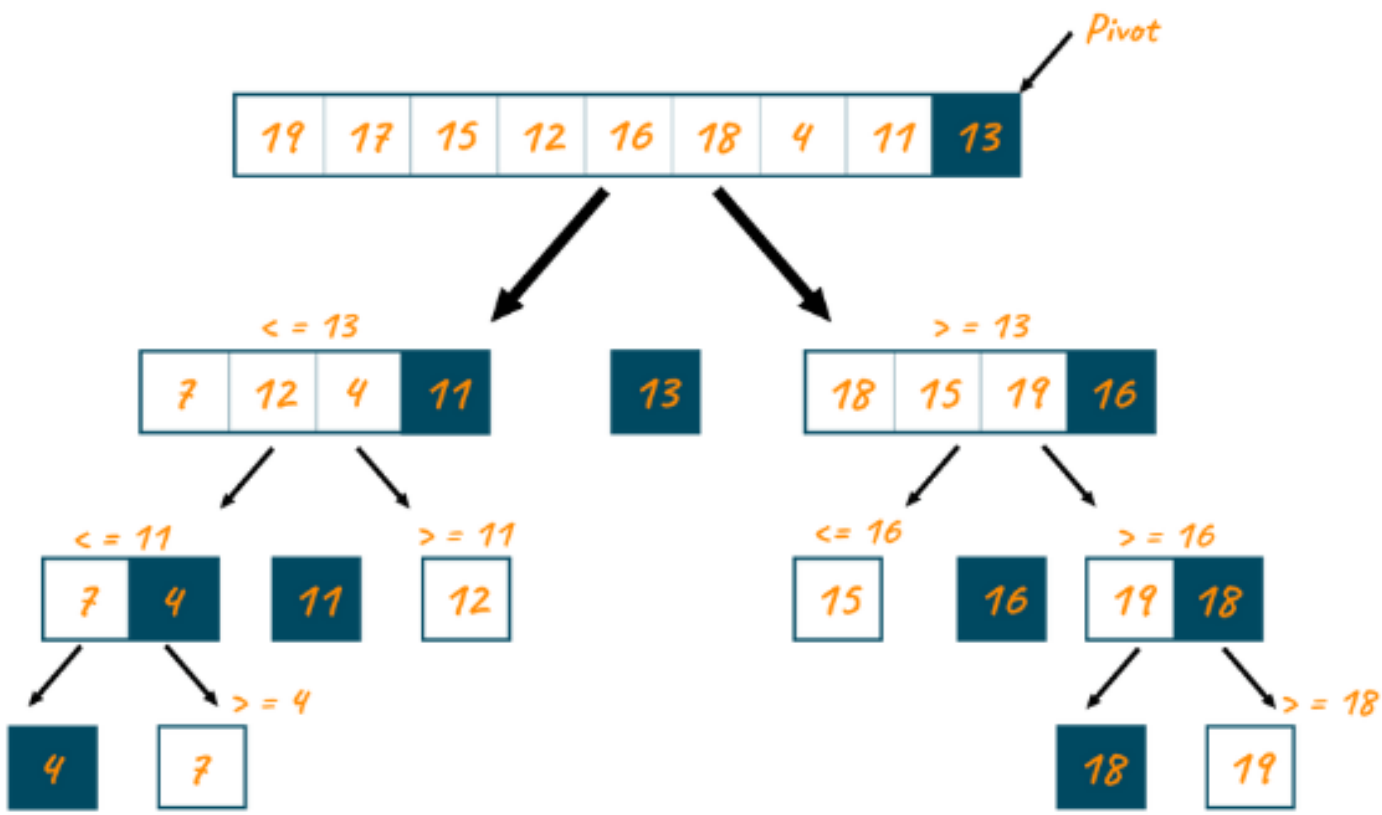

```
1 | #lang racket
2 | (define (fusion L1 L2)
3 |   (cond ((null? L1) L2)
4 |          ((null? L2) L1)
5 |          ((< (car L1) (car L2))
6 |              (cons (car L1) (fusion (cdr L1) (L2)))
7 |          (else
8 |              (cons (car L2) (fusion L1 (cdr L2)))))))
```

- Fonction finale **tri-fusion**:

```
1 | #lang racket
2 | (define (tri-fusion L)
3 |   (if (null? (cdr L))
4 |       L
5 |       (let ((r (divise L)))
6 |         (fusion (tri-fusion (car r)) (tri-fusion (cadr r))))))
```

Tri rapide (Quick sort)

- **Principe :** On va définir un pivot à partir duquel on place les valeurs. Pour trier dans un ordre croissant, on place à gauche du pivot les valeurs inférieures et à droite les valeurs supérieures. On va ensuite appeler récursivement la fonction sur les sous-listes à gauche et à droite.
- L'efficacité du tri rapide repose grandement sur le choix du pivot. Souvent, on prend le premier ou le dernier élément de la liste.



Tri rapide (Quick sort) II

- Pour une liste L, on obtient une valeur pivot p et deux listes $L1$,$L2$ (sous-listes de L) ; pour trier dans un ordre croissant, avec $L1$ la liste des éléments de L inférieurs à p, on retourne la concatenation de $L1$, $'(p)$ et $L2$.

- On va écrire une fonction **ope-list** qui permet d'obtenir la liste des éléments de L satisfaisant une certaine propriété; par exemple d'être inférieur ou supérieur au pivot.

```
1 #lang racket
2 (define (ope-list f L N p)
3   (if (null? L) N
4       (if (f (first L) p)
5           (ope-list f (cdr L) (cons (car L) N) p)
6           (ope-list f (cdr L) N p))))
```

  - Ainsi, pour un opérateur de comparaison f, une liste L et une valeur de pivot p, la fonction **ope-list** retourne la liste N des éléments x de L tels que $(f\ x\ p)$ est vrai.

Tri rapide (Quick sort) III

- Voici ensuite la fonction finale de **tri-rapide**, avec comme choix de pivot le premier élément de la liste.

```
1 #lang racket
2 (define (tri-rapide L)
3   (if (null? L)
4       L
5       (let ((p (car L))
6             (R (cdr L)))
7         (append (tri-rapide (ope-list <= R '() p))
8                 (list p) (ope-list > R '() p)))))
```