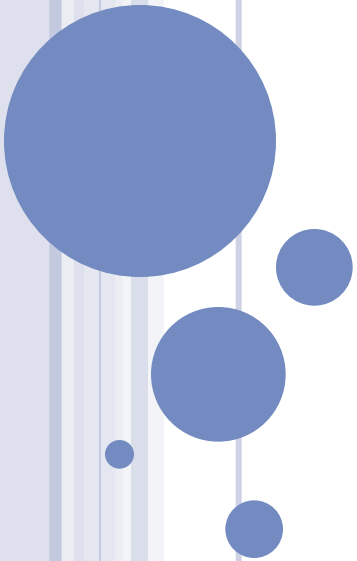


Tris

Tri par insertion

Tri par fusion

Quel est le problème à résoudre ?

- Soit une liste de nombres : '(5 2 14 1 6)
- On souhaite la trier : '(1 2 5 6 14)

Algorithmes de tri

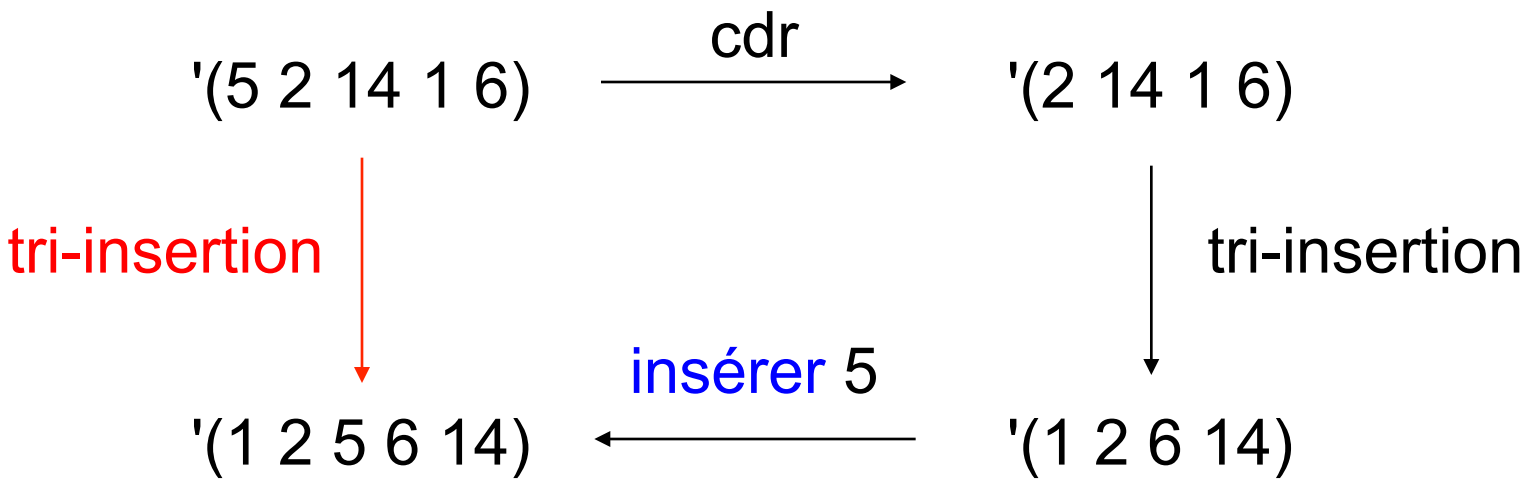
- Tris par sélection du minimum
  - tri-minimum (TP)
  - tri-bulles (TD)
- Tri par insertion
- Tri par fusion
- Tri rapide
- Tri par tas

Principes des tris par sélection

- On cherche le minimum de la liste,
  puis on recommence avec le reste de la liste
- Tri du minimum
  - fonction minimum
  - fonction enlève
- Tri bulles
  - fonction bulle, qui sélectionne le minimum
    et l'enlève de la liste en un seul passage

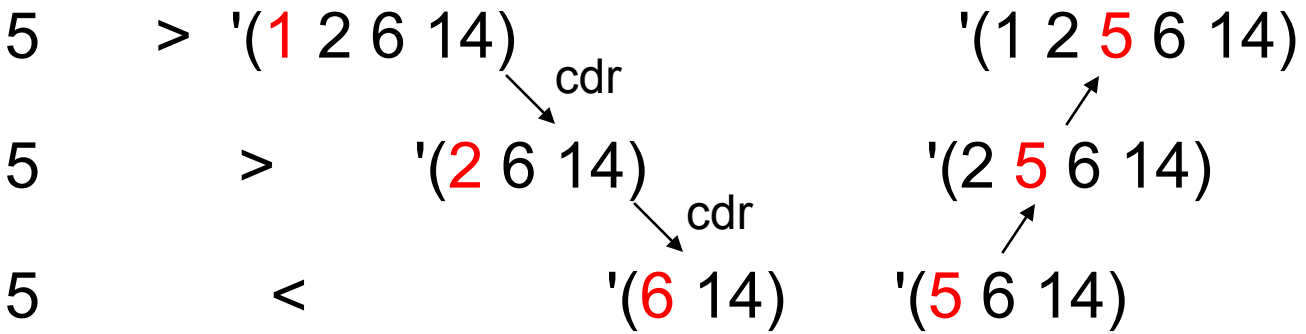
Tri par insertion : la méthode

- Principe : on trie récursivement le cdr de la liste, puis on y insère le car
- Exemple :



Insertion dans une liste triée : la méthode

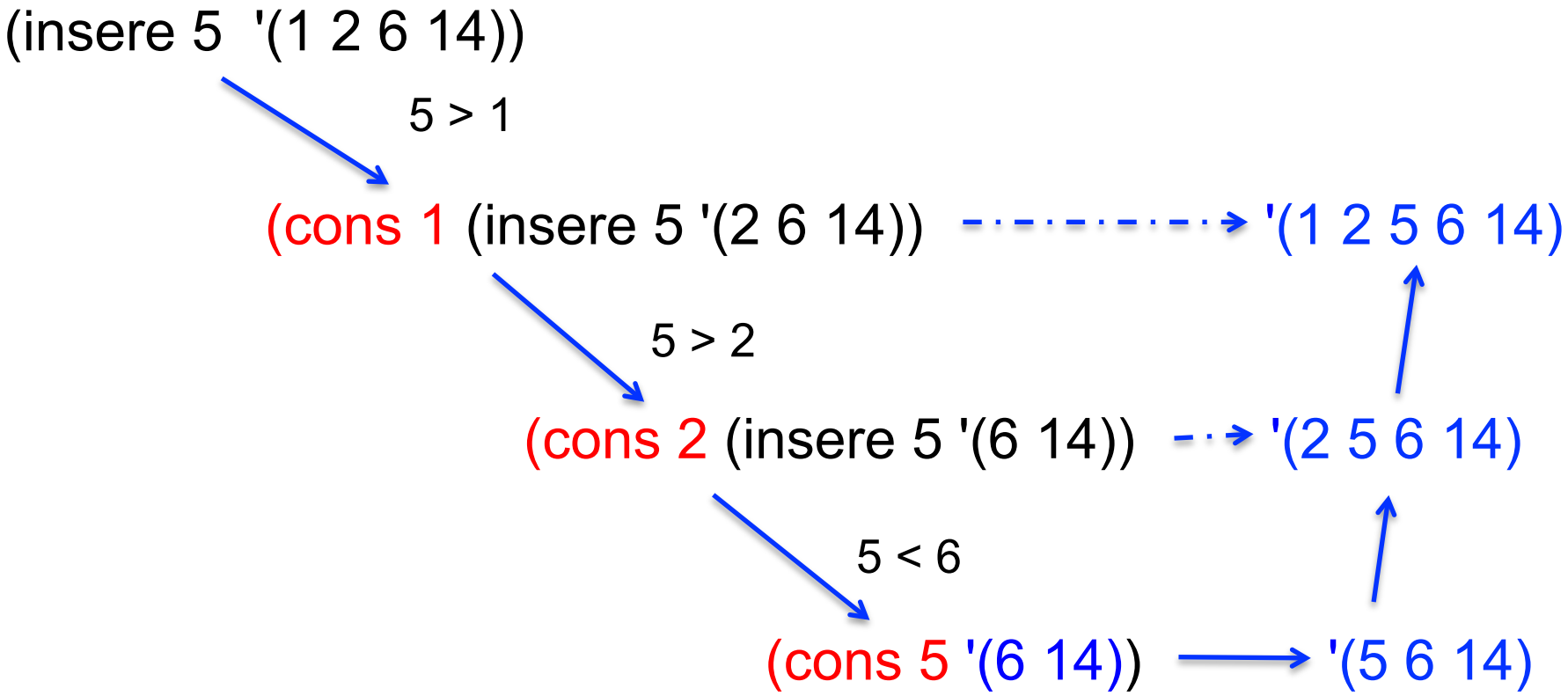
- Principe : on compare l'élément à insérer avec le car de la liste
- Exemple : insérer 5 dans '(1 2 6 14)



INSERTION DANS UNE LISTE TRIÉE : LA FONCTION

```
(define insere ; → liste de nombres triée
  (lambda (n l) ; n nombre, l liste de nombres triée
      (if     (null? l)
              (list n)
              (if     (< n (car l))
                      (cons n l)
                      (cons (car l) (insere n (cdr l)))
              ))))
```

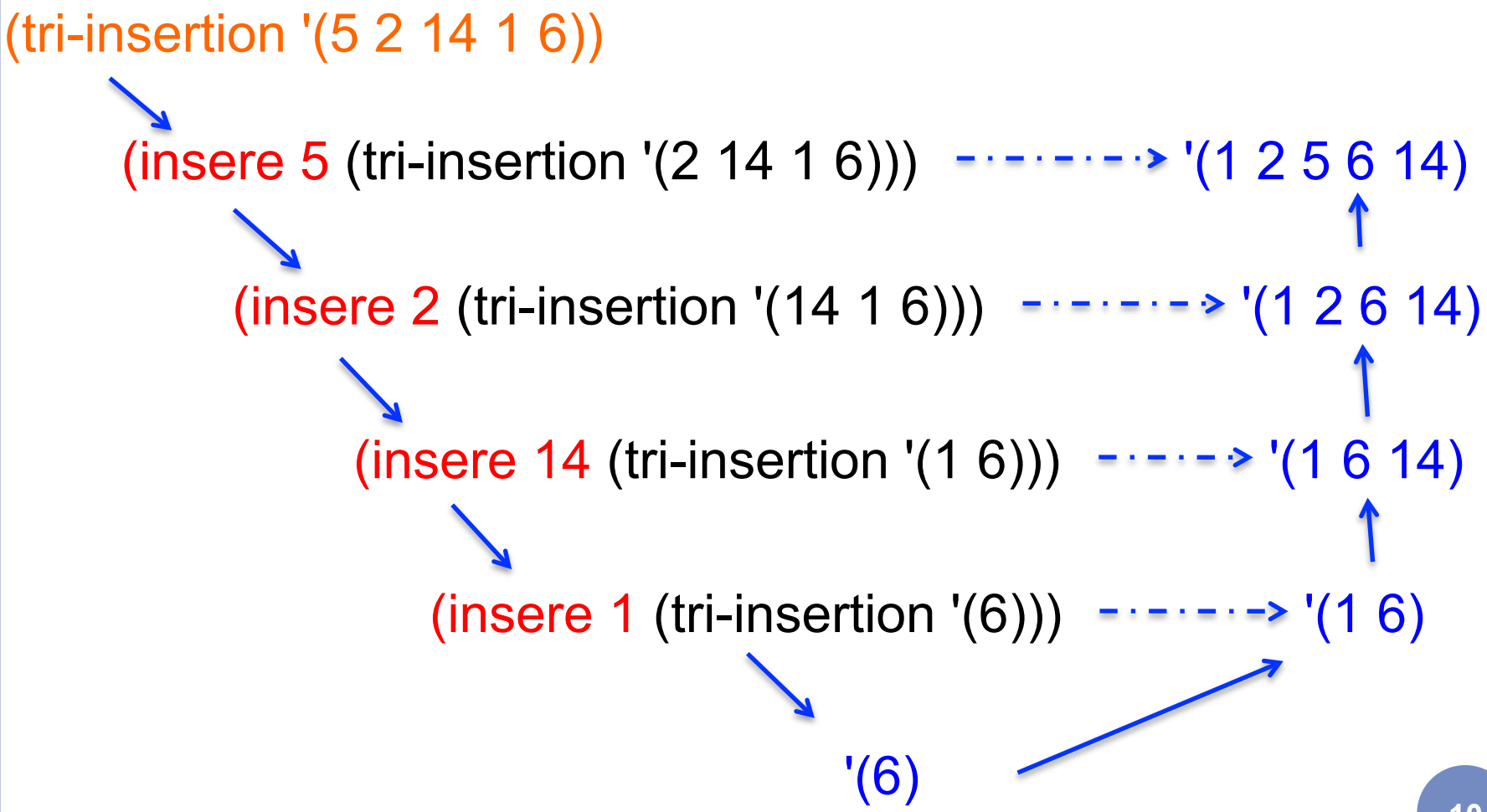
Insertion dans une liste triée : illustration de la fonction



TRI PAR INSERTION : LA FONCTION

```
(define tri-insertion ; → liste de nombres triée
   (lambda (l) ; l liste de nombres non vide
     (if (null? (cdr l))
       l
       (insere (car l) (tri-insertion (cdr l))))))
```

Tri par insertion : illustration de la fonction



Tri par fusion : l'approche « Diviser pour régner »

- Structure récursive :
  pour résoudre un problème donné,
  l'algorithme s'appelle lui-même récursivement
  une ou plusieurs fois sur des sous problèmes
  très similaires

- Le paradigme « diviser pour régner » donne lieu à trois étapes à chaque niveau de récursivité :
  diviser, régner, combiner

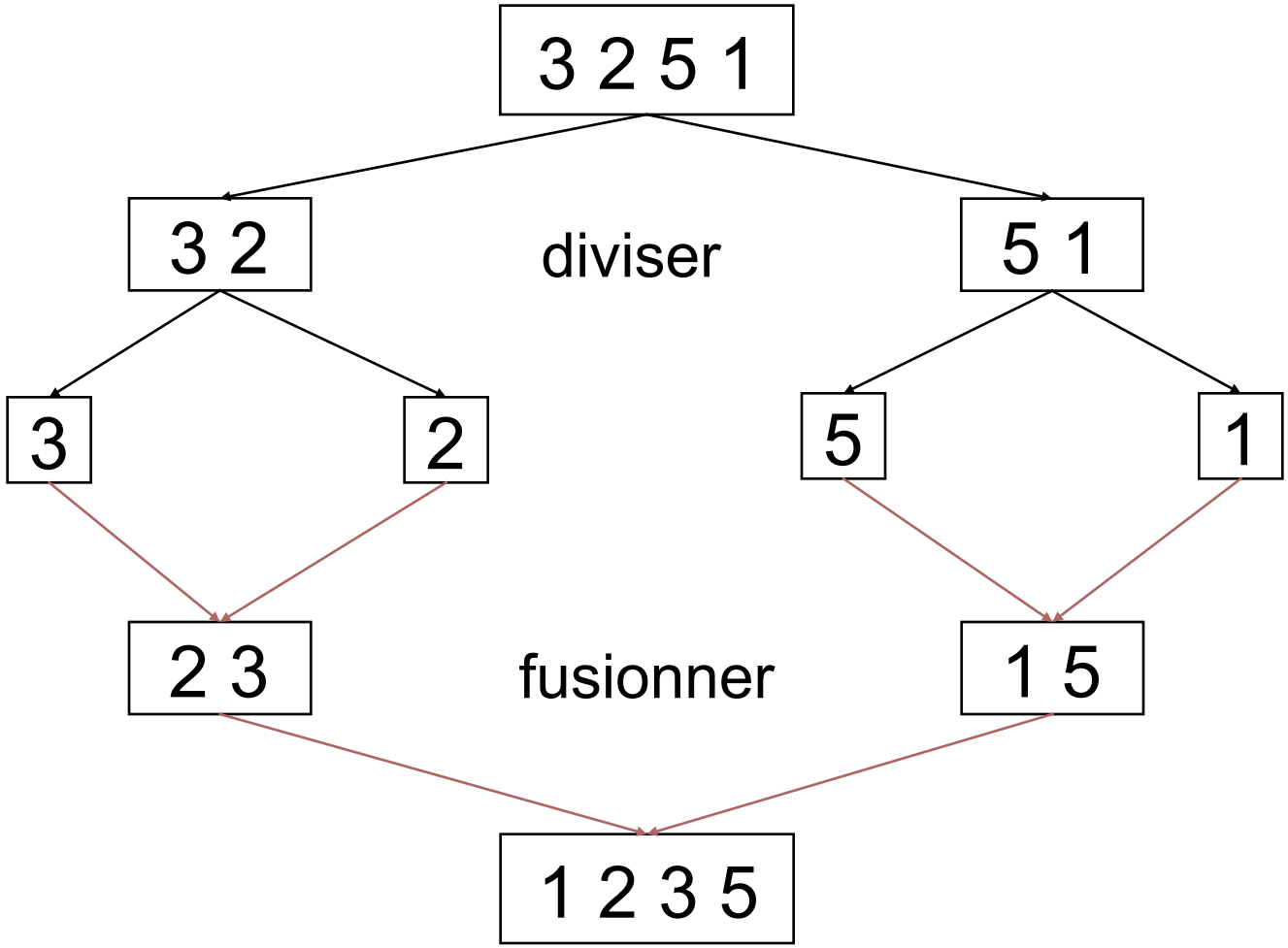
Diviser pour régner : 3 étapes

- Diviser le problème en un certain nombre de sous-problèmes
- Régner sur les sous-problèmes en les résolvant récursivement
  Si la taille d'un sous-problème est assez réduite, on peut le résoudre directement
- Combiner les solutions des sous-problèmes en une solution complète pour le problème initial

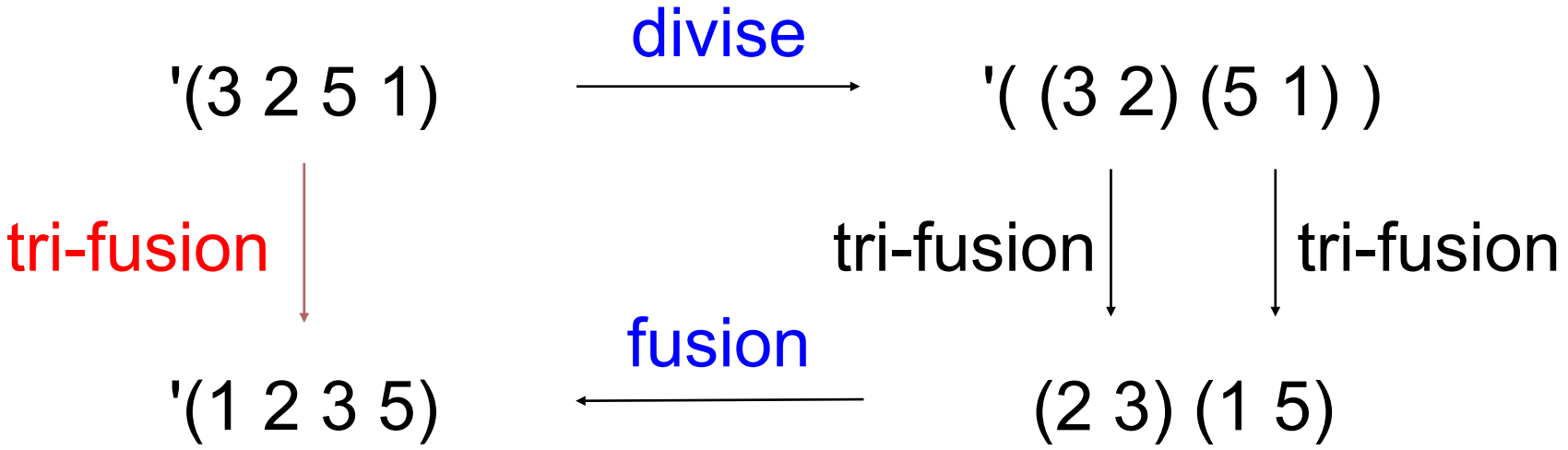
Tri par fusion : le principe

- Diviser : diviser la liste de n éléments à trier en deux sous-listes de n/2 éléments
- Régner : trier les deux sous-listes récursivement à l'aide du tri par fusion
- Combiner : fusionner les deux sous-listes triées pour produire la réponse triée

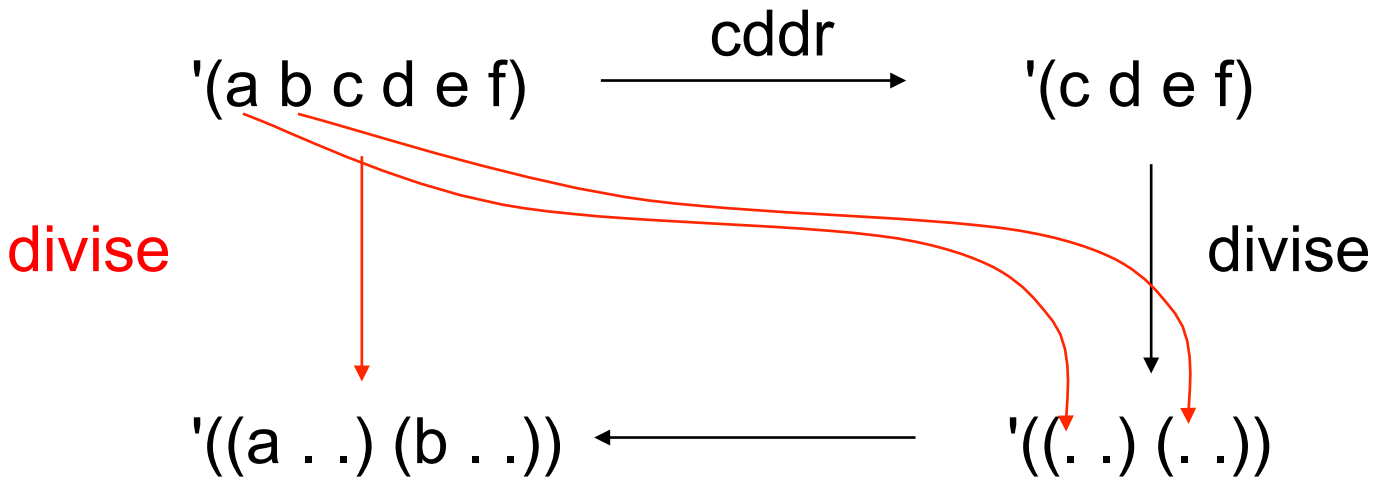
Un exemple



TRI PAR FUSION : LA MÉTHODE



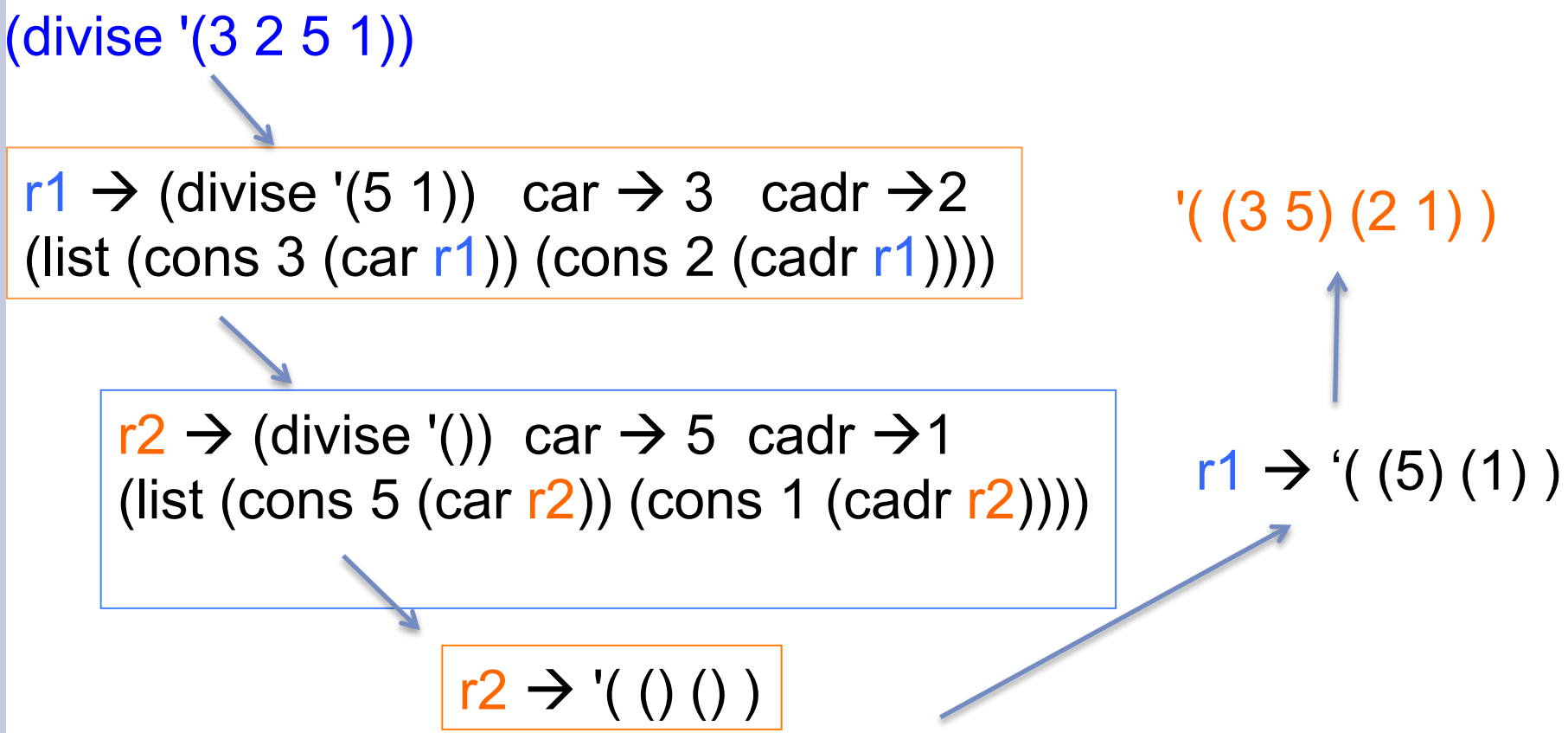
Diviser la liste en deux sous-listes : la méthode



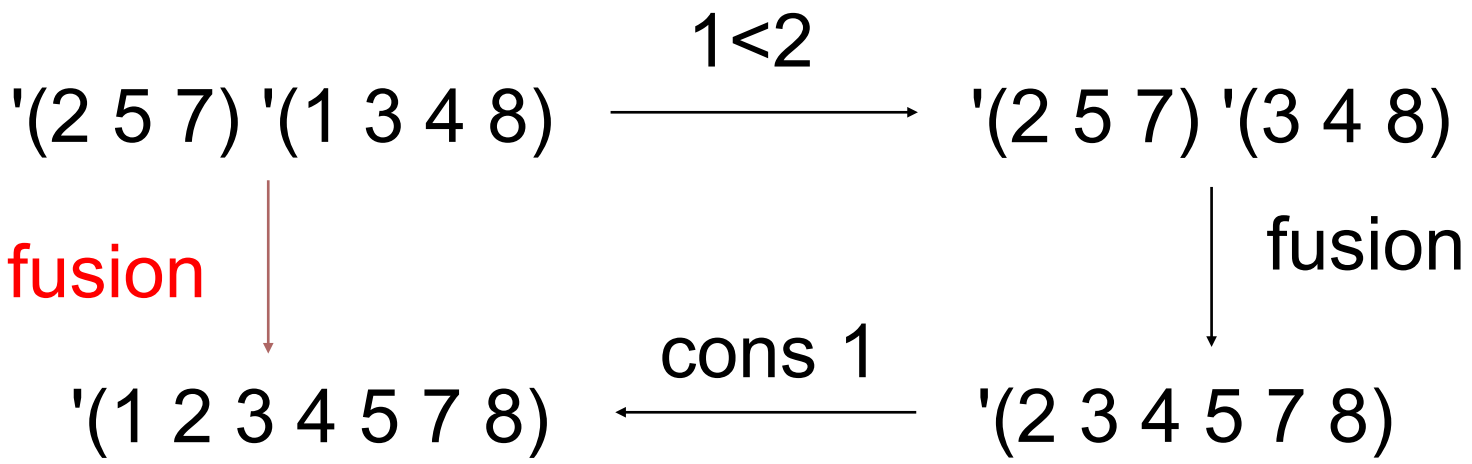
Diviser la liste en deux sous-listes : la fonction

```
(define divise ; → liste de deux listes
  (lambda (l) ; l liste
    (cond      ((null? l) '(() ()))
               ((null? (cdr l)) (list l '()))
               (else (let ((r (divise (cddr l))))
                  (list (cons (car l) (car r))
                        (cons (cadr l) (cadr r)))))))))
```

Diviser la liste en deux sous-listes : illustration de la fonction



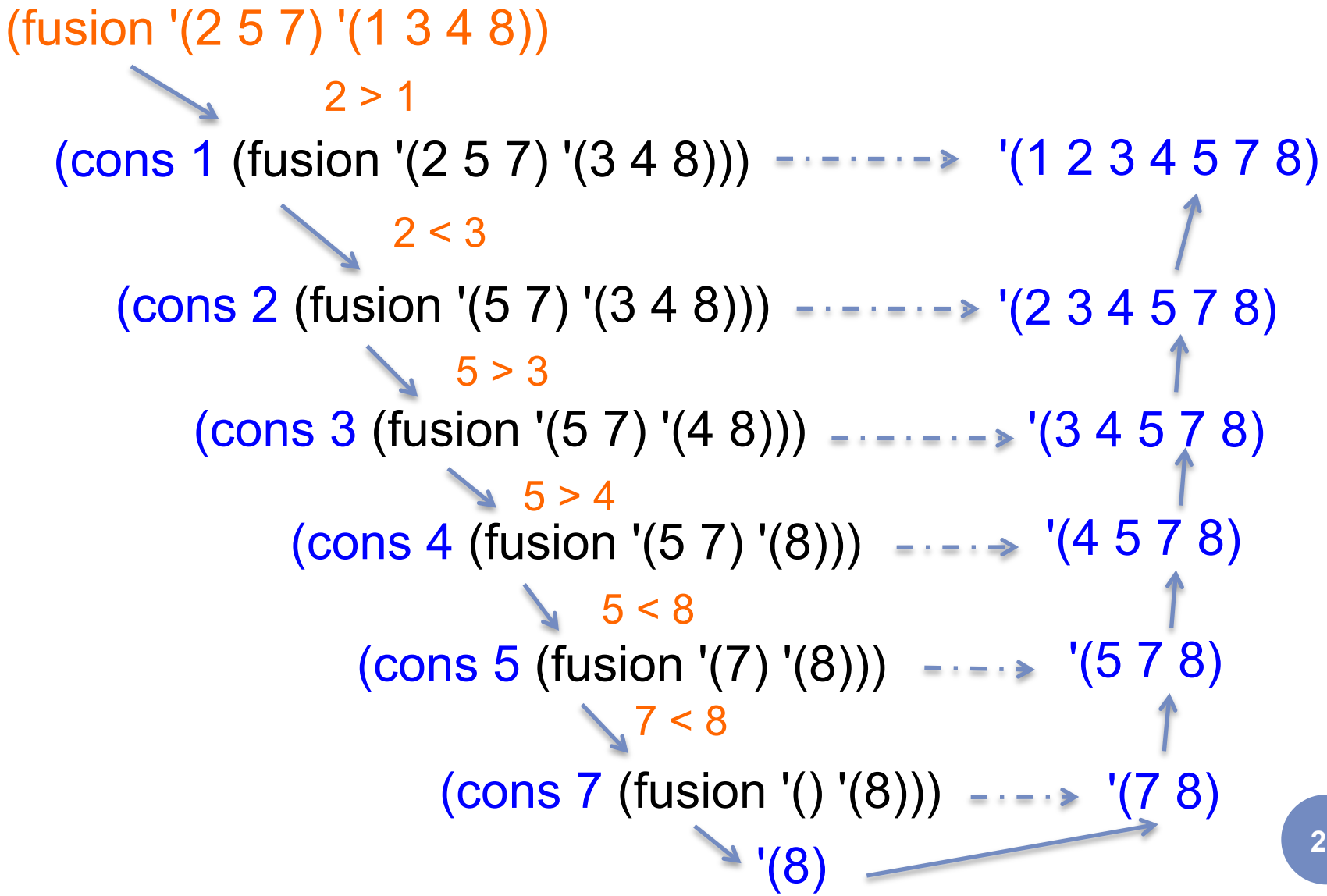
Fusionner deux listes triées : la méthode



Fusionner deux listes triées : la fonction

```
(define fusion ; → liste de nb triée
  (lambda (l1 l2) ; listes de nb triées
    (cond ((null? l1) l2)
          ((null? l2) l1)
          ((< (car l1) (car l2))
              (cons (car l1) (fusion (cdr l1) l2)))
          (else
              (cons (car l2) (fusion l1 (cdr l2)))))))
```

Fusionner deux listes triées : illustration de la fonction



TRI PAR FUSION : LA FONCTION

```
(define tri-fusion ; → liste de nb triée
  (lambda (l) ; liste de nb non vide
    (if (null? (cdr l))
        l
        (let ((r (divise l)))
          (fusion      (tri-fusion (car r))
                       (tri-fusion (cadr r)))))))
```

TRI PAR FUSION : ILLUSTRATION

(tri-fusion '(7 4 9 1))

r1 → (divise '(7 4 9 1)) ⟶ ((7 9) (4 1))

(fusion

(tri-fusion '(7 9))

r2 → (divise '(7 9)) ⟶ '((7) (9))

(fusion (tri-fusion '(7)) ⟶ '(7)

(tri-fusion '(9))) ⟶ '(9)

→ '(7 9)

(tri-fusion '(4 1)))

r3 → (divise '(4 1)) ⟶ '((4) (1))

(fusion (tri-fusion '(4)) ⟶ '(4)

(tri-fusion '(1))) ⟶ '(1)

→ '(1 4)

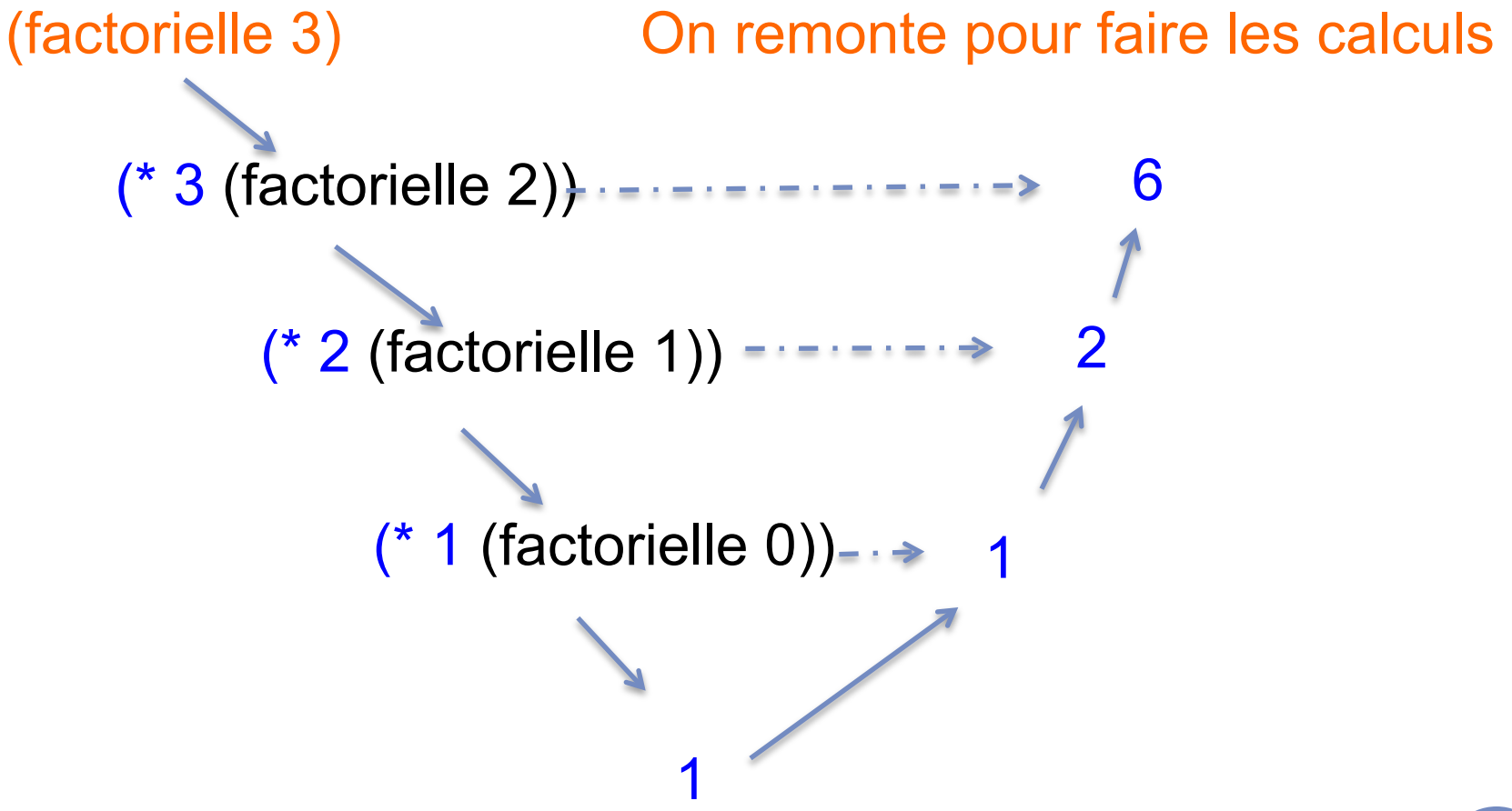
→ '(1 4 7 9)

CALCULS EN REMONTANT OU EN DESCENDANT

- Jusqu'à présent, nous avons toujours effectué les calculs en remontant des appels récursifs
- Exemple : retour sur la fonction factorielle

```
(define factorielle ; → entier positif
        (lambda (n) ; n entier positif
                (if     (= n 0)
                        1
                        (* n (factorielle (- n 1))))))
```

Fonction factorielle : illustration



Introduire un paramètre supplémentaire pour effectuer les calculs en descendant

```
(define factorielle-compteur ; → entier positif
    (lambda (n) ; n entier positif
          (fact n 1)))

; effectue le calcul de factorielle(n) en utilisant un paramètre
   supplémentaire res dans lequel on effectue le calcul
(define fact ; → entier positif
  (lambda (n res) ; entiers positifs
    (if (= n 0)
        res
        (fact (- n 1) (* res n)))))
```

Fonction factorielle-compteur : illustration

(factorielle-compteur 3)

(fact 3 1)

(fact 2 (* 1 3))

(fact 1 (* 3 2))

(fact 0 (* 6 1))

6

Les calculs effectués en descendant sont stockés dans res

On renvoie directement le résultat contenu dans res

Remarques

- La fonction factorielle-compteur est celle qui répond à la spécification.
  Il est indispensable d'écrire une fonction qui répond à la spécification, même si elle ne fait rien d'autre que d'appeler la fonction fact.
  L'utilisateur n'a pas à savoir que nous utilisons un deuxième argument.
- La fonction fact est celle qui fait effectivement tout le travail.

Quel intérêt ?

- Dans la fonction fact, on a une récursivité terminale. Avec certains langages de programmation et certains compilateurs, cette récursivité terminale est « dérécursifiée » afin d'améliorer l'efficacité du programme.
- On se rapproche en effet d'une solution itérative :

```
res ← 1
TantQue n>0 Faire
   res ← res*n
    n ← n-1
FinTantQue
Afficher res
```