Chapitre 9 : Tris

1) Tri par sélection
2) Tri par insertion
3) Tri rapide (récursif)
4) Principe du Diviser pour Regner (rappel)
5) Tri fusion (diviser pour régner)
6) "Master Theorem" pour résoudre des récurrences obtenues en diviser pour régner

3) Tri rapide

Nous allons maintenant voir un algorithme de tri récursif.

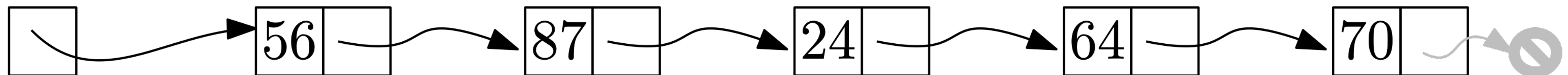
Ce sera un tri sur une liste et non pas un tableau, car il est un peu plus simple de l'écrire ainsi, mais la version tableau existe également.

3) Tri rapide

Début: tous les nombres sont dans une liste.

Principe:

- **choisir** un élément comme étant le **pivot** (disons, le premier élément)

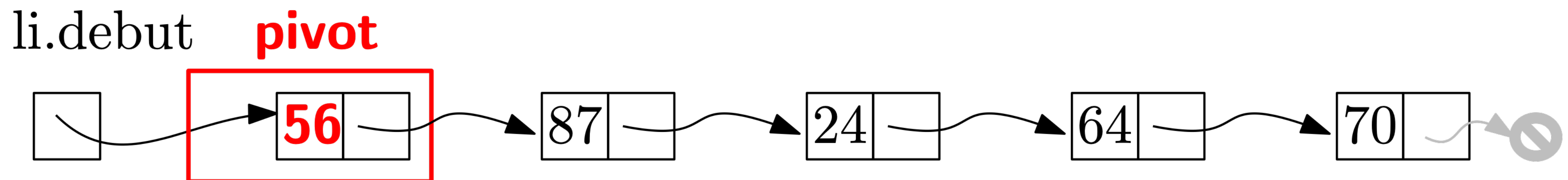


3) Tri rapide

Début: tous les nombres sont dans une liste.

Principe:

- **choisir** un élément comme étant le **pivot** (disons, le premier élément)

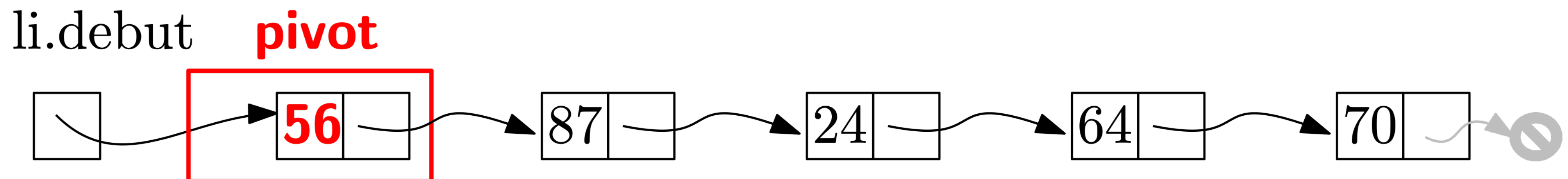


3) Tri rapide

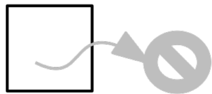
Début: tous les nombres sont dans une liste.

Principe:

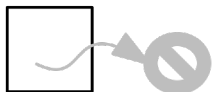
- répartir les autres éléments en **deux listes: les plus petits que le pivot** d'une part, **et les plus grands** d'autre part



Liste des plus petits que le pivot



Liste des plus grands que le pivot

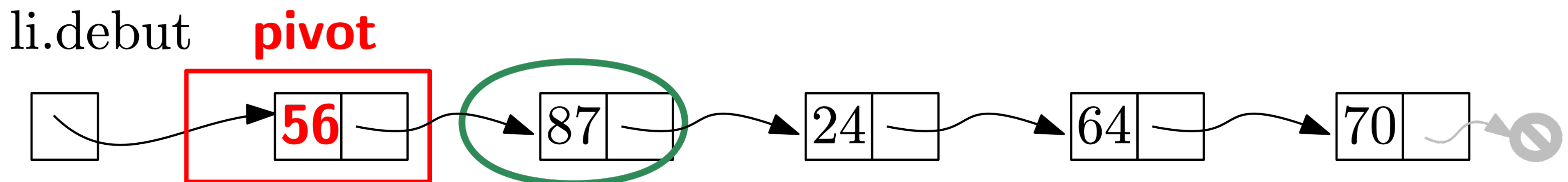


3) Tri rapide

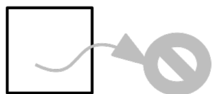
Début: tous les nombres sont dans une liste.

Principe:

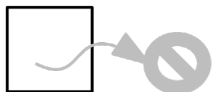
- répartir les autres éléments en **deux listes: les plus petits que le pivot** d'une part, **et les plus grands** d'autre part



Liste des plus petits que le pivot



Liste des plus grands que le pivot

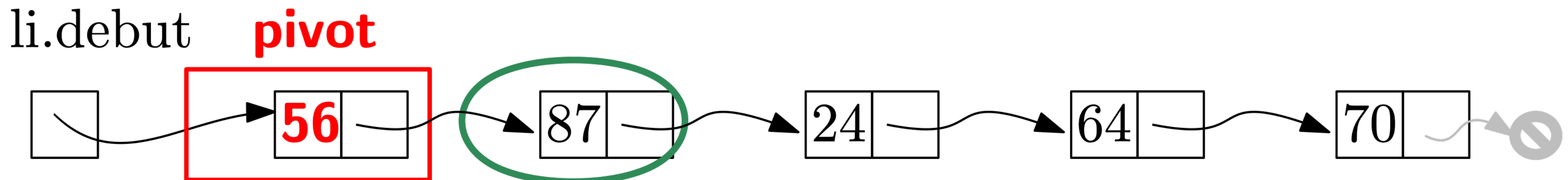


3) Tri rapide

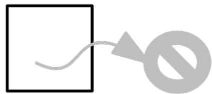
Début: tous les nombres sont dans une liste.

Principe:

- répartir les autres éléments en **deux listes: les plus petits que le pivot** d'une part, **et les plus grands** d'autre part



Liste des plus petits que le pivot



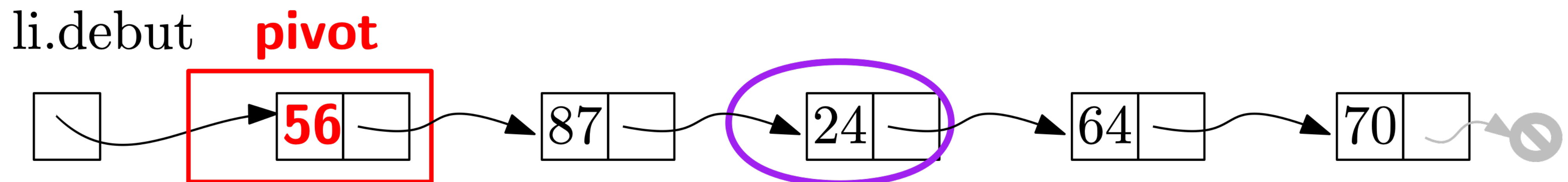
Liste des plus grands que le pivot

3) Tri rapide

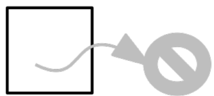
Début: tous les nombres sont dans une liste.

Principe:

- répartir les autres éléments en **deux listes: les plus petits que le pivot** d'une part, **et les plus grands** d'autre part



Liste des plus petits que le pivot



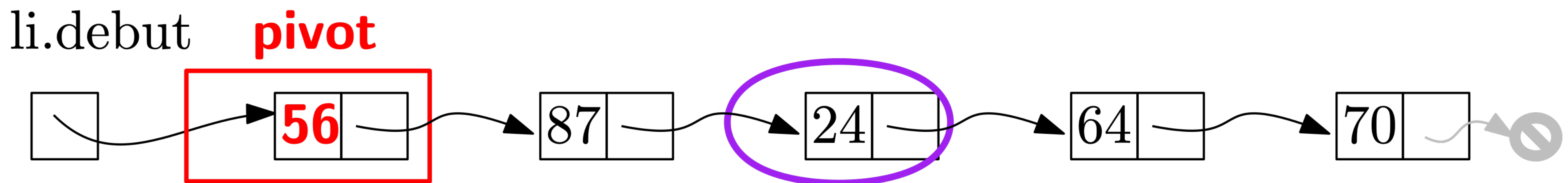
Liste des plus grands que le pivot

3) Tri rapide

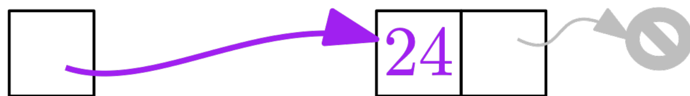
Début: tous les nombres sont dans une liste.

Principe:

- répartir les autres éléments en **deux listes: les plus petits que le pivot** d'une part, **et les plus grands** d'autre part



Liste des plus petits que le pivot



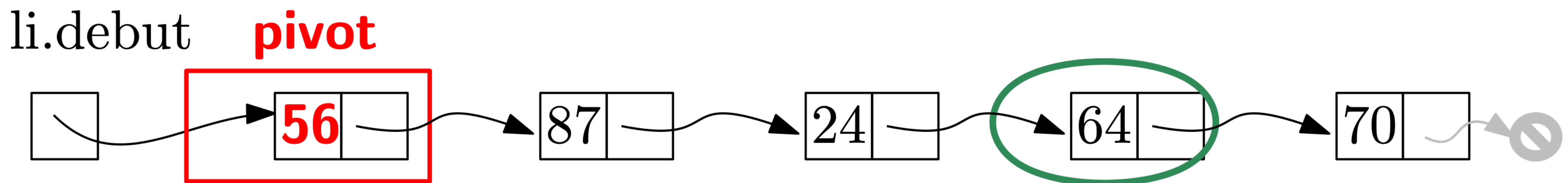
Liste des plus grands que le pivot

3) Tri rapide

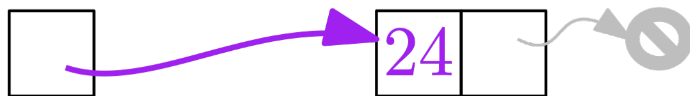
Début: tous les nombres sont dans une liste.

Principe:

- répartir les autres éléments en **deux listes: les plus petits que le pivot** d'une part, **et les plus grands** d'autre part



Liste des plus petits que le pivot



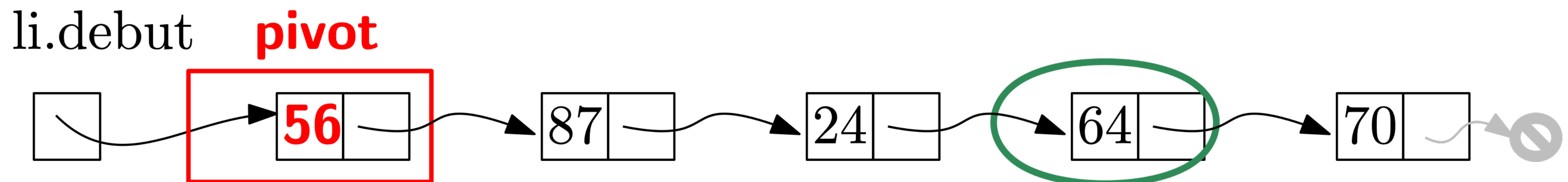
Liste des plus grands que le pivot

3) Tri rapide

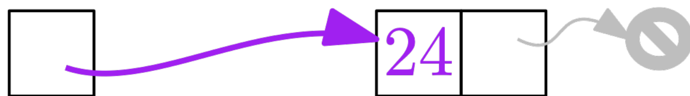
Début: tous les nombres sont dans une liste.

Principe:

- répartir les autres éléments en **deux listes: les plus petits que le pivot** d'une part, **et les plus grands** d'autre part



Liste des plus petits que le pivot



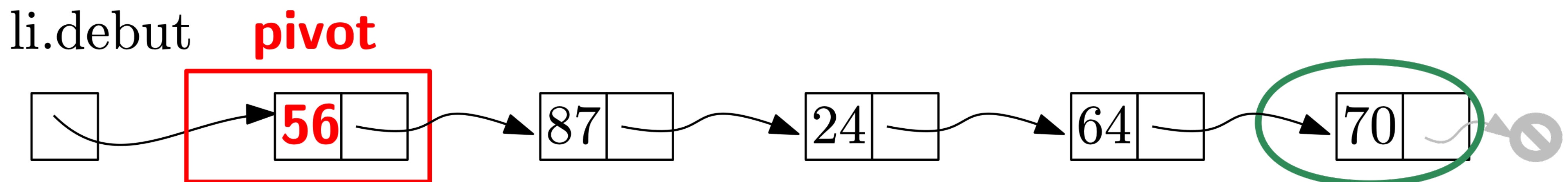
Liste des plus grands que le pivot

3) Tri rapide

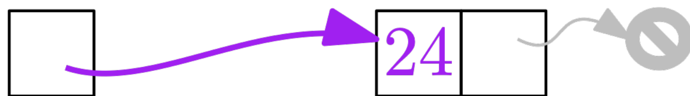
Début: tous les nombres sont dans une liste.

Principe:

- répartir les autres éléments en **deux listes: les plus petits que le pivot** d'une part, **et les plus grands** d'autre part



Liste des plus petits que le pivot



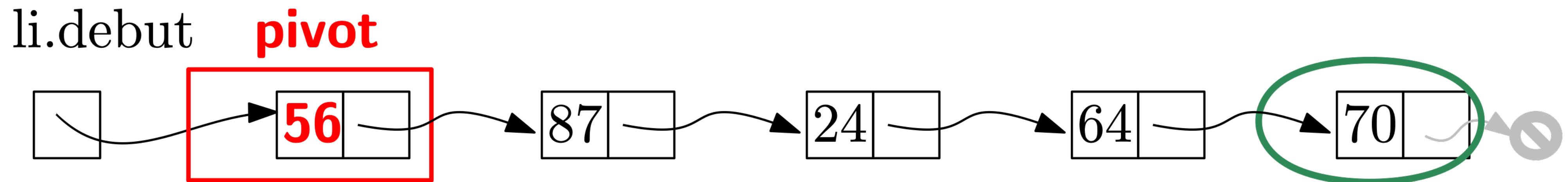
Liste des plus grands que le pivot

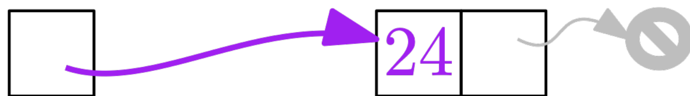
3) Tri rapide

Début: tous les nombres sont dans une liste.

Principe:

- répartir les autres éléments en **deux listes: les plus petits que le pivot** d'une part, **et les plus grands** d'autre part





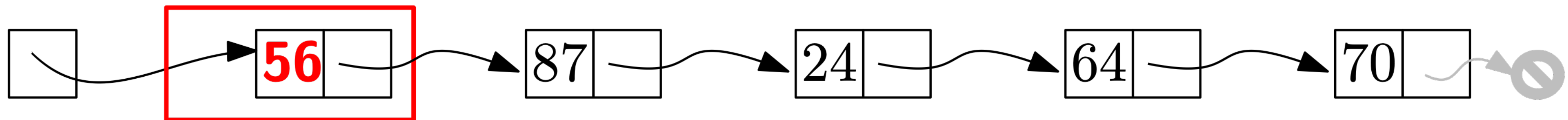
3) Tri rapide

Début: tous les nombres sont dans une liste.

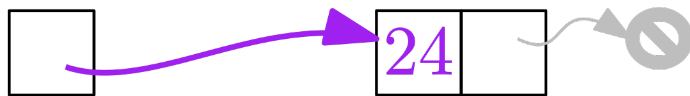
Principe:

- **trier récursivement les deux listes** (moins longues)

li.debut **pivot**



Liste des plus petits que le pivot



Appel récursif pour trier

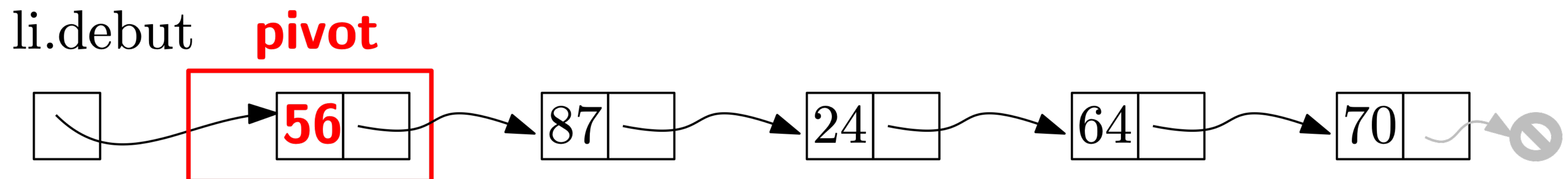
Liste des plus grands que le pivot

3) Tri rapide

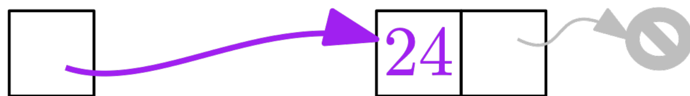
Début: tous les nombres sont dans une liste.

Principe:

- **trier récursivement les deux listes** (moins longues)



Liste des plus petits que le pivot



Liste des plus grands que le pivot

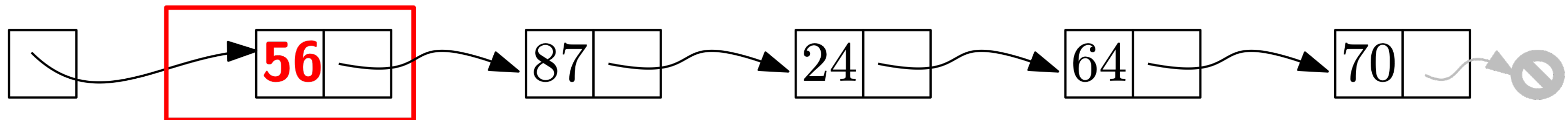
Appel récursif pour trier

3) Tri rapide

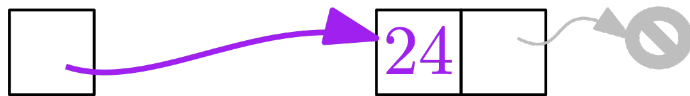
Début: tous les nombres sont dans une liste.

Principe:

- **trier récursivement les deux listes** (moins longues)



Liste des plus petits que le pivot



Liste des plus grands que le pivot

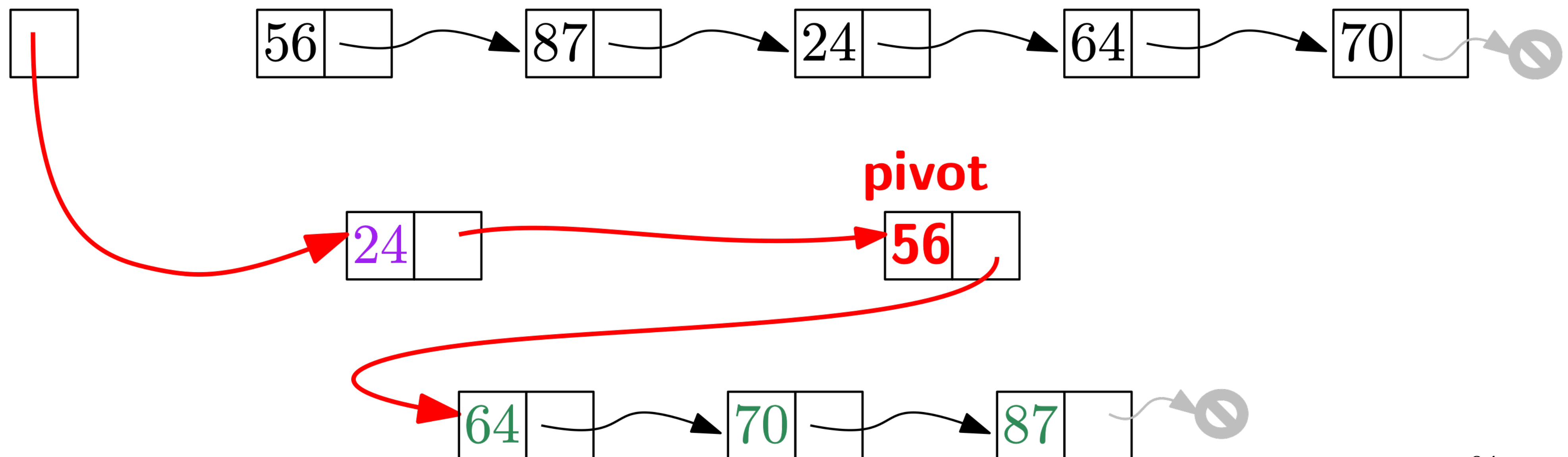
Appel récursif pour trier

3) Tri rapide

Début: tous les nombres sont dans une liste.

Principe:

- **Assembler les 3 morceaux: les petits triés, puis le pivot, puis les grands triés pour former la nouvelle liste.**



3) Tri rapide

```
Fun (liste, liste) separeListe(liste li):
        liste listePlusPetits:=NewListe()
        liste listePlusGrands:=NewListe()
        nombre pivot:=li.debut.data // valeur 1er maillon
        adresse adrMaillonEnCours:=li.debut.suiv // 2e maillon
        Tant que adrMaillonEnCours != -1:
                nombre valeurMaillon:=adrMaillonEnCours.data
                Si valeurMaillon<pivot:
                        ajouteDebut(listePlusPetits, valeurMaillon)
                Sinon:
                        ajouteDebut(listePlusGrands, valeurMaillon)
                adrMaillonEnCours := adrMaillonEnCours.suiv
        Retourner (listePlusPetits, listePlusGrands)
```

3) Tri rapide

```
Proc assemble(liste liDebut, nombre valeurMilieu, liste liFin):
        /* Les deux listes peuvent être vides. Le but est d'ajouter
         valeurMilieu à la fin de liDebut,  puis d'ajouter liFin juste après ce
        nouveau maillon */

        /* Création du maillon milieu */
        maillon M:=NewMaillon()
        M.data:=valeurMilieu

        /* On relie le maillon M et liFin: */
        M.suiv:=liFin.debut
```

3) Tri rapide

```
Proc assemble(liste liDebut, nombre valeurMilieu, liste liFin):
        /* Puis ajoute le maillon M à la fin de liDebut : */
        Si liDebut.debut == -1: // si liDebut est vide:
                liDebut.debut:=adresseDe(M)
        Sinon:
                adresse adrMaillonEnCours:=liDebut.debut
                Tant que adrMaillonEnCours.suiv!=-1:
                        adrMaillonEnCours:=adrMaillonEnCours.suiv
                adrMaillonEnCours.suiv:=adresseDe(M)
```

3) Tri rapide

```
Proc triRapide(liste li):
    Si li.debut != -1 et li.debut.suiv!=-1:
        /* si la liste a taille au moins 2  */
        nombre pivot:=li.debut.data
        liste listePlusPetits, listePlusGrands
        (listePlusPetits, listePlusGrands):=separeListe(li)
        triRapide(listePlusPetits)
        triRapide(listePlusGrands)
        assemble(listePlusPetits, pivot, listePlusGrands)
        li.debut:=listePlusPetits.debut
```

3) Tri rapide

n la taille de la liste
on compte les op. élém de liste.

Complexité: O(n)

```
Fun (liste, liste) separeListe(liste li):
        liste listePlusPetits:=NewListe()
        liste listePlusGrands:=NewListe()
        nombre pivot:=li.debut.data // valeur 1er maillon
        adresse adrMaillonEnCours:=li.debut.suiv // 2e maillon
        Tant que adrMaillonEnCours != -1:
                nombre valeurMaillon:=adrMaillonEnCours.data
                Si valeurMaillon<pivot:
                        ajouteDebut(listePlusPetits, valeurMaillon)
                Sinon:
                        ajouteDebut(listePlusGrands, valeurMaillon)
                adrMaillonEnCours := adrMaillonEnCours.suiv
        Retourner (listePlusPetits, listePlusGrands)
```

3) Tri rapide

```
Proc assemble(liste liDebut, nombre valeurMilieu, liste liFin):
    /* Les deux listes peuvent être vides. Le but est d'ajouter
     valeurMilieu à la fin de liDebut,  puis d'ajouter liFin juste après ce
    nouveau maillon */

    /* Création du maillon milieu */
    maillon M:=NewMaillon()
    M.data:=valeurMilieu

    /* On relie le maillon M et liFin: */
    M.suiv:=liFin.debut
```

Complexité:
O(1) pour cette partie

3) Tri rapide

Complexité:
O(n_1) pour cette partie, où n_1 est la taille de liDebut.

```
Proc assemble(liste liDebut, nombre valeurMilieu, liste liFin):
    /* Puis ajoute le maillon M à la fin de liDebut : */
    Si liDebut.debut == -1: // si liDebut est vide:
        liDebut.debut:=adresseDe(M)
    Sinon:
        adresse adrMaillonEnCours:=liDebut.debut
        Tant que adrMaillonEnCours.suiv!=1:
            adrMaillonEnCours:=adrMaillonEnCours.suiv
        adrMaillonEnCours.suiv:=adresseDe(M)
```

3) Tri rapide

Complexité (avec n taille de la liste)

```
Proc triRapide(liste li):
    Si li.debut != -1 et li.debut.suiv!=-1:
        /* si la liste a taille au moins 2 */
        nombre pivot:=li.debut.data
        listeDC listePlusPetits, listePlusGrands
        (listePlusPetits, listePlusGrands):=separeListe(li)
        triRapide(listePlusPetits)
        triRapide(listePlusGrands)
        assemble(listePlusPetits, pivot, listePlusGrands)
        li.debut:=listePlusPetits.debut
```

O(1)

O(n)

O(n)

3) Tri rapide

Complexité (avec n taille de la liste)

```
Proc triRapide(liste li):
    Si li.debut != -1 et li.debut.suiv!=-1:
        /* si la liste a taille au moins 2 */          O(1)
        nombre pivot:=li.debut.data
        listeDC listePlusPetits, listePlusGrands
        (listePlusPetits, listePlusGrands):=separeListe(li)   O(n)
        triRapide(listePlusPetits)
        triRapide(listePlusGrands)
        assemble(listePlusPetits, pivot, listePlusGrands)     O(n)
        li.debut:=listePlusPetits.debut
```

3) Tri rapide

Rappel: pour calculer la **complexité de la fonction récursive**, il faut utiliser une **formule de récurrence**.

On appelle u_n **la complexité en pire cas de l'algorithme de tri rapide sur une liste de taille n.**

Alors, il suffit d'observer que les listes $\mathrm{listePlusPetits}$ et $\mathrm{listePlusGrands}$, dont on appellera n_1 et n_2 les tailles respectives, vérifient : $n_1 + n_2 = n\text{-}1$, avec aucune garantie sur la répartition entre n_1 et n_2

(par exemple, on peut avoir $n_1 = n\text{-}1$ et $n_2 = 0$)

3) Tri rapide

On obtient: $\boldsymbol{u_n = u_{n_1} + u_{n_2} + O(n)}$

Intuitivement, le cas le moins favorable est lorsque $n_1 = n$-1 et $n_2 = 0$ (ou l'inverse), on a alors:

$$\boldsymbol{u_n = u_{n-1} + u_0 + O(n)}$$

On peut prouver qu'une telle équation mène au fait que u_n est en $O(n^2)$

$\rightarrow$ **La complexité du tri rapide est quadratique.**

Note, pour information: la complexité "en moyenne" (hors-programme) du tri rapide est en $O(n.\log n)$ *.*

4) Introduction au concept Diviser pour Régner

"Diviser pour Régner" est une **technique algorithmique** consistant à:

- ***Diviser:*** *découper un problème initial en sous-problèmes*
- ***Régner:*** *résoudre les sous-problèmes (généralement récursivement)*
- ***Combiner:*** *calculer une solution au problème initial à partir des solutions des sous-problèmes*

Elle peut permettre d'obtenir des algorithmes de meilleur complexité.

4) Introduction au concept Diviser pour Régner

Dans sa forme la plus simple et la plus courante, **"diviser pour régner"** consiste à **diviser un problème à résoudre sur une entrée de taille n** en **deux problèmes à résoudre sur des entrées de taille (environ) n/2.**

On les **résout récursivement**, puis on **combine les solutions** avec "peu" d'opérations..

Nous allons voir deux algorithmes utilisant la méthode "Diviser pour Régner":

- **Tri fusion**
- Recherche dichotomique (vu dans le chap 4 récursivité)

5) Tri fusion

Principe:

- Je **coupe la liste en deux listes de même taille** (ou presque).
- Je **trie ces listes de taille moitié, récursivement**
- Je **fusionne ces deux listes triées** selon le principe suivant**: je regarde le premier maillon non-traité de chaque liste**, et **j'insère la valeur la plus petite** parmi ces deux maillons dans ma liste auxiliaire (à la fin).

→ **voir animation** dans le fichier TriFusion.pdf

5) Tri fusion

```
Fun (liste, liste) separeEnDeuxListesTailleEgale(liste li):
        liste liste1:=NewListe()
        liste liste2:=NewListe()
        adresse adrMaillonEnCours:=li.debut
        int numeroListeOuMettreLeProchain:=1
        Tant que adrMaillonEnCours != -1:
                Si numeroListeOuMettreLeProchain == 1:
                        ajouteDebut(adrMaillonEnCours.data, liste1)
                        numeroListeOuMettreLeProchain:=2
                Sinon:
                        ajouteDebut(adrMaillonEnCours.data, liste2)
                        numeroListeOuMettreLeProchain:=1
                adrMaillonEnCours:=adrMaillonEnCours.suiv
        Retourner (liste1, liste2)
```

5) Tri fusion

```
Fun (liste, liste) separeEnDeuxListesTailleEgale(liste li):
    liste liste1:=NewListe()
    liste liste2:=NewListe()
    adresse adrMaillonEnCours:=li.debut
    int numeroListeOuMettreLeProchain:=1
    Tant que adrMaillonEnCours != -1:
        Si numeroListeOuMettreLeProchain == 1:
            ajouteDebut(adrMaillonEnCours.data, liste1)
            numeroListeOuMettreLeProchain:=2
        Sinon:
            ajouteDebut(adrMaillonEnCours.data, liste2)
            numeroListeOuMettreLeProchain:=1
        adrMaillonEnCours:=adrMaillonEnCours.suiv
    Retourner (liste1, liste2)
```

Complexité: O(n)
où n=taille de li

5) Tri fusion

```
Fun adresse fusionne(liste liste1, liste liste2):
    liste listeAux:=NewListe()
    adresse dernierMaillonAjouté:=-1
    adresse adrPremierMaillonRestantListe1:= liste1.debut
    adresse adrPremierMaillonRestantListe2:= liste2.debut
    Tant que adrPremierMaillonRestantListe1 != -1 et
                        adrPremierMaillonRestantListe2 != -1:
        maillon M:=NewMaillon()
        Si dernierMaillonAjouté == -1 :
        /* on n'a pas encore mis de maillon dans la liste */
            listeAux.debut:=adresseDe(M)
        Sinon:
            dernierMaillonAjouté.suiv:=M
```

5) Tri fusion

```
Fun adresse fusionne(liste liste1, liste liste2):
        Tant que adrPremierMaillonRestantListe1 != -1 et  …..
                Si adrPremierMaillonRestantListe1.data <
                                adrPremierMaillonRestantListe2.data:
                        M.data:=adrPremierMaillonResantListe1.data
                        adrPremierMaillonRestantListe1:=
                                adrPremierMaillonRestantListe1.suiv
                Sinon:
                        M.data:=adrPremierMaillonResantListe2.data
                        adrPremierMaillonRestantListe2:=
                                adrPremierMaillonRestantListe2.suiv

                dernierMaillonAjouté:=adresseDe(M)
```

5) Tri fusion

```
Fun adresse fusionne(liste liste1, liste liste2):
    Si adrPremierMaillonRestantListe1 == -1 :
            /* s'il n'y a plus d'éléments restants dans liste1,  on ajoute
            liste2 à la fin de la liste qu'on est en train de construire *
        dernierMaillonAjouté.suiv:=adrPremierMaillonRestantListe2

    Sinon: /* il n'y a plus d'éléments restants dans liste2 */
        dernierMailonAjouté.suiv:= adrPremierMaillonRestantListe1

    Retourner listeAux.debut
```

5) Tri fusion

```
Proc triFusion(liste li):
    Si li.debut != -1 et li.debut.suiv != -1:
        /* si la liste a taille au moins 2*/
        liste liste1, liste2
        (liste1, liste2):=separeEnDeuxListesTailleEgale(li)
        triFusion(liste1)
        triFusion(liste2)
        li.debut:=fusionne(liste1, liste2)
```

5) Tri fusion

Complexité de separeEnDeuxListesTailleEgale(li) : **O(n)**

Complexité de fusionne(liste1, liste2): **$O(n_1+n_2)$** où n_1 et n_2 sont les tailles de liste1 et liste2.

Soit $\boldsymbol{u_n}$ la complexité en pire cas de triFusion sur une liste de taille n.

On obtient la relation de récurrence:

$$\begin{cases} u_n = u_{\lfloor \frac{n}{2} \rfloor} + u_{\lceil \frac{n}{2} \rceil} + O(n) \approx \mathbf{2u_{\frac{n}{2}} + n}, & si\ n > 1 \\ u_0 = 1, \quad et\ u_1 = 2 \end{cases}$$

On peut prouver que u_n est en $O(n.\log n)$.

→ Tri fusion en $O(n.log\ n)$→ plus rapide que $O(n^2)$

6) Pour aller plus loin (facultatif): "Master Theorem"

On a mentionné que la relation de récurrence suivante:

$$\begin{cases} u_n = u_{\left\lceil \frac{n}{2} \right\rceil} + u_{\left\lfloor \frac{n}{2} \right\rfloor} + O(n) \approx \mathbf{2u_{\frac{n}{2}} + n}, & si\ n > 1 \\ u_0 = 1, \quad et\ u_1 = 2 & \end{cases}$$

… nous permet de prouver que $\boldsymbol{u_n}$ est en $\boldsymbol{O(n.\log n)}$.

Il en va de même pour un cas un peu plus général:

$$\begin{cases} u_n = \mathbf{c.u_{\frac{n}{c}} + O(n)}, & si\ n > 1 \\ u_n = constante, & si\ n\ "petit" \end{cases}$$

où c est une constante ($c = 2$ dans le cas particulier du tri fusion) alors $\boldsymbol{u_n}$ est en $\boldsymbol{O(n.\log n)}$.

6) Pour aller plus loin (facultatif): "Master Theorem"

Il existe un théorème qui nous aide à **calculer les complexités des algorithmes en Diviser pour Régner**

Théorème 4.1 (Théorème général.) *Soient $a \geqslant 1$ et $b > 1$ deux constantes, soit $f(n)$ une fonction et soit $T(n)$ définie pour les entiers non négatifs par la récurrence*

$$T(n) = aT(n/b) + f(n) ,$$

où l'on interprète n/b comme signifiant $\lfloor n/b \rfloor$ ou $\lceil n/b \rceil$. $T(n)$ peut alors être bornée asymptotiquement de la façon suivante.

1) Si $f(n) = O(n^{\log_b a - \varepsilon})$ pour une certaine constante $\varepsilon > 0$, alors $T(n) = \Theta(n^{\log_b a})$.

2) Si $f(n) = \Theta(n^{\log_b a})$, alors $T(n) = \Theta(n^{\log_b a} \lg n)$.

3) Si $f(n) = \Omega(n^{\log_b a + \varepsilon})$ pour une certaine constante $\varepsilon > 0$, et si $af(n/b) \leqslant cf(n)$ pour une certaine constante $c < 1$ et pour tout n suffisamment grand, alors $T(n) = \Theta(f(n))$.

Voyons le cas où a et b sont la même constante

6) Pour aller plus loin (facultatif): "Master Theorem"

Master Theorem, cas particulier :

Si $u_n = \boldsymbol{c.u_{\frac{n}{c}} + f(n)}$, **pour une certaine fonction f, alors:**

Cas 1 : S'il existe ε tel que $\boldsymbol{f(n) = O(n^{1-\varepsilon})}$

alors $u_n = \boldsymbol{\Theta(n)}$

Cas 2: Si $\boldsymbol{f(n) = \Theta(n)}$

alors $u_n = \boldsymbol{\Theta(n \,.\log n)}$

Cas 3: S'il existe ε tel que $\boldsymbol{f(n)} = \Omega\,(n^{1+\varepsilon})$

(+ une petite condition, cf théorème page précédente)

alors $u_n = \boldsymbol{\Theta(f(n))}$

Autres cas : pas couverts par ce théorème (par ex. $f(\mathrm{n}) = \Theta(n.\log n)$)