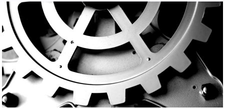
Algorithmique et structures de données
Récursion



Gaël Mahé

slides : Elise Bonzon et Gaël Mahé

Université Paris Descartes

Licence 2

Récursion

Récursion

Récursivité

Récursivité

Une définition **récursive** d'un mot contient ce mot.

Récursivité

Récursivité

Une définition **récursive** d'un mot contient ce mot.

Exemple : un ascendant d'une personne est soit :

- son père
- sa mère
- un ascendant de son père ou de sa mère

Récursivité

Récursivité

Une définition **récursive** d'un mot contient ce mot.

Exemple : un ascendant d'une personne est soit :

- son père
- sa mère
- un ascendant de son père ou de sa mère

"La mère de mon père est un ascendant de mon père, c'est donc mon ascendant."

Récursivité

- Un algorithme est défini **récursivement**
  lorsque sa définition fait référence à l'algorithme lui-même.
- Pour décrire l'algorithme sur une donnée *d*,
  on utilise l'algorithme lui même
  sur un sous ensemble de *d* ou sur une donnée plus petite.
- Les langages de programmation autorisent en général la récursivité :
  Le compilateur transforme le code
  en une version plus proche de l'approche itérative.

Récursivité

- Un algorithme est défini **récursivement**
  lorsque sa définition fait référence à l'algorithme lui-même.
- Pour décrire l'algorithme sur une donnée d,
  on utilise l'algorithme lui même
  sur un sous ensemble de d ou sur une donnée plus petite.
- Les langages de programmation autorisent en général la récursivité :
  Le compilateur transforme le code
  en une version plus proche de l'approche itérative.

Récursivité

- Un algorithme est défini **récursivement**
  lorsque sa définition fait référence à l'algorithme lui-même.
- Pour décrire l'algorithme sur une donnée d,
  on utilise l'algorithme lui même
  sur un sous ensemble de d ou sur une donnée plus petite.
- Les langages de programmation autorisent en général la récursivité :
  Le compilateur transforme le code
  en une version plus proche de l'approche itérative.

Récursivité

- Un algorithme est défini **récursivement**
  lorsque sa définition fait référence à l'algorithme lui-même.
- Pour décrire l'algorithme sur une donnée d,
  on utilise l'algorithme lui même
  sur un sous ensemble de d ou sur une donnée plus petite.
- Les langages de programmation autorisent en général la récursivité :
  Le compilateur transforme le code
  en une version plus proche de l'approche itérative.

Règles

- **Ne jamais ré-appliquer l'algorithme sur des données plus grandes**
- **Toujours effectuer un test de terminaison**

Exemple

En considérant le type abstrait liste récursive du TD1 :

Algorithme 1 : contient

début

/* ENTRÉE : une liste récursive ℓ, un élément x */

/* SORTIE : ℓ contient-elle x ? */

si $\mathrm{estVide}(\ell)$ **alors** Retour faux

sinon si $tete(\ell) = x$ **alors** Retour vrai

sinon Retour $\mathrm{contient}(\mathrm{retire}(\ell), x)$

fin

Récursion et induction

Récursivité terminale

- L'appel récursif est la dernière instruction à être évaluée.
- Cette instruction consiste en un simple appel à la fonction, et jamais à un calcul ou une composition
- À la compilation, la récursion term. se transforme facilement en itération

Exemple 1 : algorithme précédent.
Exemple 2 :

Algorithme 2 : EstPuissanceDe2

début
 /* ENTRÉE : un entier $n \neq 0$ */
 /* SORTIE : n est-il une puissance de 2 ? */
 si $n = 1$ **alors** Retour vrai
 sinon si n *impair* **alors** Retour faux
 sinon Retour $\mathrm{EstPuissanceDe2}(n \div 2)$
fin

Intérêt : pas besoin d'attendre le résultat de l'appel récursif pour terminer l'exécution de l'appel en cours.

Récursif terminal $\rightarrow$ itératif

Algorithme 3 : contient : version récursive

début

/* ENTRÉE : une liste récursive ℓ, un élément x */
/* SORTIE : ℓ contient-elle x ? */
si $\mathrm{estVide}(\ell)$ **alors** Retour faux
sinon si $tete(\ell) = x$ **alors** Retour vrai
sinon Retour $\mathrm{contient}(\mathrm{retire}(\ell), x)$

fin

Algorithme 4 : contient : version itérative

début

/* ENTRÉE : une liste récursive ℓ, un élément x */
/* SORTIE : ℓ contient-elle x ? */
tant que *not* $\mathrm{estVide}(\ell)$ **et** $tete(\ell) \neq x$ **faire**
   $\ell \leftarrow \mathrm{retire}(\ell)$
si $\mathrm{estVide}(\ell)$ **alors** **retourner** *faux*
sinon **retourner** *vrai*

fin

Récursif terminal $\rightarrow$ itératif

Algorithme 5 : EstPuissanceDe2 : version récursive

début
- /* ENTRÉE : un entier n */
- /* SORTIE : n est-il une puissance de 2 ? */
- **si** $n = 1$ **alors** Retour vrai
- **sinon si** n *impair* **alors** Retour faux
- **sinon** Retour $\mathrm{EstPuissanceDe2}(n \div 2)$

fin

Algorithme 6 : EstPuissanceDe2 : version itérative

début
- /* ENTRÉE : un entier n */
- /* SORTIE : n est-il une puissance de 2 ? */
- **tant que** n *pair* **faire**
  - $n \leftarrow n \div 2$
- **si** $n = 1$ **alors** **retourner** *vrai*
- **sinon** **retourner** *faux*

fin

Récursif $\rightarrow$ itératif (cas général)

Algorithme 7 : Expression récursive terminale d'une fonction f

début

/* ENTRÉE : x, SORTIE : $f(x)$ */
si *condition1*(x) **alors** **retourner** $f_1(x)$
...
sinon si *condition_p*(x) **alors** **retourner** $f_p(x)$
sinon **retourner** $f\big(T(x)\big)$

fin

Algorithme 8 : Expression itérative de la même fonction f

début

/* ENTRÉE : x, SORTIE : $f(x)$ */
tant que *not condition1*(x) **et** ... **et** *not condition_p*(x) **faire**
$x \leftarrow T(x)$
si *condition1*(x) **alors** **retourner** $f_1(x)$
...
sinon si *condition_p-1*(x) **alors** **retourner** $f_{p-1}(x)$
sinon **retourner** $f_p(x)$

fin

Récursion et induction

Ex : l'appel récursif n'est pas la dernière instruction

Algorithme 9 : Une fonction récursive non terminale f

début
  /* ENTRÉE : x, SORTIE : action $f(x)$ */
  si *condition1*(x) **alors** $f_1(x)$
  ...
  sinon si *condition_p*(x) **alors** $f_p(x)$
  sinon
    avant(x)
    $f\left(T(x)\right)$
    apres(x)
fin

Exemple d'exécution

Soit x_0 tel que

- $T\big(T(x_0)\big)$ vérifie condition 1
- x_0 et $T(x_0)$ ne vérifient aucune condition 1 à p

Exécution :

$f(x_0) \longrightarrow avant(x_0)$

$\quad f\big(T(x_0)\big) \longrightarrow avant\big(T(x_0)\big)$

$\qquad f\Big(T\big(T(x_0)\big)\Big) \longrightarrow f_1\Big(T\big(T(x_0)\big)\Big)$

$\qquad apres\big(T(x_0)\big) \hookleftarrow$

$\quad apres(x_0) \hookleftarrow$

Exemple d'exécution

Soit x_0 tel que

- $T\big(T(x_0)\big)$ vérifie condition 1
- x_0 et $T(x_0)$ ne vérifient aucune condition 1 à p

Exécution :

$$
\begin{array}{l}
f(x_0) \longrightarrow avant(x_0) \\
\qquad\qquad f\big(T(x_0)\big) \longrightarrow avant\big(T(x_0)\big) \\
\qquad\qquad\qquad\qquad f\Big(T\big(T(x_0)\big)\Big) \longrightarrow f_1\Big(T\big(T(x_0)\big)\Big) \\
\qquad\qquad\qquad\qquad apres\big(T(x_0)\big) \hookleftarrow \\
\qquad\qquad apres(x_0) \hookleftarrow
\end{array}
$$

Exemple d'exécution

Soit x_0 tel que

- $T\big(T(x_0)\big)$ vérifie condition 1
- x_0 et $T(x_0)$ ne vérifient aucune condition 1 à p

Exécution :

$f(x_0) \longrightarrow avant(x_0)$

$\qquad f\big(T(x_0)\big) \longrightarrow avant\big(T(x_0)\big)$

$\qquad\qquad f\Big(T\big(T(x_0)\big)\Big) \longrightarrow f_1\Big(T\big(T(x_0)\big)\Big)$

$\qquad\qquad apres\big(T(x_0)\big) \hookleftarrow$

$\qquad apres(x_0) \hookleftarrow$

Exemple d'exécution

Soit x_0 tel que

- $T\big(T(x_0)\big)$ vérifie condition 1
- x_0 et $T(x_0)$ ne vérifient aucune condition 1 à p

Exécution :

$f(x_0) \longrightarrow avant(x_0)$

$f\big(T(x_0)\big) \longrightarrow avant\big(T(x_0)\big)$

$f\Big(T\big(T(x_0)\big)\Big) \longrightarrow f_1\Big(T\big(T(x_0)\big)\Big)$

$apres\big(T(x_0)\big) \hookleftarrow$

$apres(x_0) \hookleftarrow$

Exemple d'exécution

Soit x_0 tel que

- $T\big(T(x_0)\big)$ vérifie condition 1
- x_0 et $T(x_0)$ ne vérifient aucune condition 1 à p

Exécution :

$f(x_0) \longrightarrow \mathit{avant}(x_0)$

$\qquad f\big(T(x_0)\big) \longrightarrow \mathit{avant}\big(T(x_0)\big)$

$\qquad\qquad f\Big(T\big(T(x_0)\big)\Big) \longrightarrow f_1\Big(T\big(T(x_0)\big)\Big)$

$\qquad\qquad \mathit{apres}\big(T(x_0)\big) \hookleftarrow$

$\qquad \mathit{apres}(x_0) \hookleftarrow$

Exemple d'exécution

Soit x_0 tel que

- $T\big(T(x_0)\big)$ vérifie condition 1
- x_0 et $T(x_0)$ ne vérifient aucune condition 1 à p

Exécution :

$$
\begin{array}{lllll}
f(x_0) \longrightarrow avant(x_0) & & & & \\
 & f\big(T(x_0)\big) \longrightarrow avant\big(T(x_0)\big) & & & \\
 & & f\Big(T\big(T(x_0)\big)\Big) \longrightarrow f_1\Big(T\big(T(x_0)\big)\Big) & & \\
 & & apres\big(T(x_0)\big) \hookleftarrow & & \\
 & apres(x_0) \hookleftarrow & & &
\end{array}
$$

Dérécursivation

Algorithme 10 : Expression itérative de la même fonction f

début

/* ENTRÉE : x, SORTIE : action $f(x)$ */

$P \leftarrow \text{pilevide}$

$\text{empile}((x, "appel"), P)$

tant que $not\, \text{estvide}(P)$ **faire**

  $(y, etat) \leftarrow \text{sommet}(P); \text{depile}(P)$

  si $etat = "appel"$ **alors**

    si $condition1(y)$ **alors** $f_1(y)$

    ...

    sinon si $condition\_p(y)$ **alors** $f_p(y)$

    sinon

      avant(y)

      $\text{empile}((y, "retour"), P)$

      $\text{empile}((T(y), "appel"), P)$

  si $etat = "retour"$ **alors**

    apres(y)

fin

Récursion et induction

Nombre de Fibonacci

- $\text{fibo}\colon \mathbb{N} \to \mathbb{N}$
- $\text{fibo}(0) = 1$
- $\text{fibo}(1) = 1$
- $\text{fibo}(n) = \text{fibo}(n-1) + \text{fibo}(n-2)$

Nombre de Fibonacci : version récursive

Algorithme 11 : FibRec

début
  /* ENTRÉES: Un nombre n */
  /* SORTIE: Le nombre de Fibonacci de n */
  si $n = 0$ *ou* $n = 1$ **alors**
    Retour 1
  sinon
    Retour FibRec($n - 1$) + FibRec($n - 2$)
fin

Nombre de Fibonacci : version récursive

Algorithme 12 : FibRec

début

/* ENTRÉES: Un nombre n */

/* SORTIE: Le nombre de Fibonacci de n */

si $n = 0$ *ou* $n = 1$ **alors**

Retour 1

sinon

Retour FibRec$(n-1)$ + FibRec$(n-2)$

fin

- Cette fonction nécessite 2 appels récursifs : fonction **dyadique**
- Le résultat est assez rapide pour $n = 5$
  mais beaucoup plus long pour $n = 30$

Nombre de Fibonacci : version récursive

Algorithme 13 : FibRec

début
/* ENTRÉES: Un nombre n */
/* SORTIE: Le nombre de Fibonacci de n */
si $n = 0$ *ou* $n = 1$ **alors**
Retour 1
sinon
Retour FibRec$(n-1)$ + FibRec$(n-2)$
fin

- Cette fonction nécessite 2 appels récursifs : fonction **dyadique**
- Le résultat est assez rapide pour $n = 5$
  mais beaucoup plus long pour $n = 30$

FibRec(5)?

Complexité du nombre de Fibonacci récursif

- Opérations significatives : nombre d'appel de la fonction FibRec
- Le nombre de d'appels croît exponentiellement en fonction de n :
  - FibRec(0) : 1 appel
  - FibRec(1) : 1 appel
  - FibRec(2) : 3 appels
  - FibRec(3) : 5 appels
  - FibRec(4) : 9 appels
  - FibRec(5) : 15 appels

$\Rightarrow$ $\mathrm{Ap}(\mathrm{FibRec}(n)) = 1 + \mathrm{Ap}(\mathrm{FibRec}(n-1)) + \mathrm{Ap}(\mathrm{FibRec}(n-2))$

- On va chercher un encadrement de $\mathrm{Ap}(\mathrm{FibRec}(n))$, puis une forme close (fonction de n)

Complexité du nombre de Fibonacci récursif

$$
\begin{aligned}
\mathrm{Ap}(\mathrm{FibRec}(n)) &= 1 + \mathrm{Ap}(\mathrm{FibRec}(n-1)) + \mathrm{Ap}(\mathrm{FibRec}(n-2)) \\
&\leq 1 + 2 * \mathrm{Ap}(\mathrm{FibRec}(n-1)) \qquad \textit{fonction croissante} \\
&\leq 1 + 2 * (1 + \mathrm{Ap}(\mathrm{FibRec}(n-2)) + \mathrm{Ap}(\mathrm{FibRec}(n-3))) \\
&\leq 1 + 2 + 2 * 2 * \mathrm{Ap}(\mathrm{FibRec}(n-2)) \\
&\vdots \\
&\leq 1 + 2 + 2^2 + \ldots 2^n \\
&\leq \Sigma_{i=0}^{n} 2^i \\
&\leq 2^{n+1} - 1
\end{aligned}
$$

Complexité du nombre de Fibonacci récursif

$$\begin{aligned}
\mathrm{Ap}(\mathrm{FibRec}(n)) &= 1 + \mathrm{Ap}(\mathrm{FibRec}(n-1)) + \mathrm{Ap}(\mathrm{FibRec}(n-2)) \\
&\geq 1 + 2 * \mathrm{Ap}(\mathrm{FibRec}(n-2)) \qquad \textit{fonction croissante} \\
&\geq 1 + 2 * (1 + \mathrm{Ap}(\mathrm{FibRec}(n-3)) + \mathrm{Ap}(\mathrm{FibRec}(n-4))) \\
&\geq 1 + 2 + 2 * 2 * \mathrm{Ap}(\mathrm{FibRec}(n-4)) \\
&\vdots \\
&\geq 1 + 2 + 2^2 + \ldots 2^{n/2} \\
&\geq \Sigma_{i=0}^{n/2} 2^i \\
&\geq 2^{n/2+1} - 1
\end{aligned}$$

$$\Rightarrow 2^{n/2+1} - 1 \leq \mathrm{Ap}(\mathrm{FibRec}(n)) \leq 2^{n+1} - 1$$

Calcul cplxité via séries génératrices (1)

A toute suite $\big(x(n)\big)_{n\in\mathbb{Z}}$
on peut associer une série génératrice (SG) X telle que :

$$X(z) = \sum_{n\in\mathbb{Z}} x(n) z^n$$

pour $z \in D \subset \mathbb{C}$

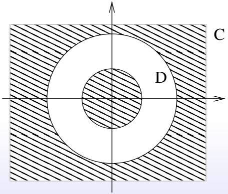


Figure: Domaine de définition d'une série génératrice.

Calcul cplxité via séries génératrices (2)

- Exemple 1 : suite $\mathbb{1}$: $\forall\, n \geq 0, \mathbb{1}(n) = 1$

$$\mathbb{I}(z) = \sum_{n=0}^{+\infty} z^n = \frac{1}{1-z} \quad \text{pour } |z| < 1$$

- Exemple 2 : suite x : $\forall\, n \geq 0, x(n) = a^n$

$$X(z) = \sum_{n=0}^{+\infty} a^n z^n = \frac{1}{1-az} \quad \text{pour } |z| < \frac{1}{|a|}$$

Calcul cplxité via séries génératrices (3)

- La transformation en SG est **linéaire** :

$$\mathrm{SG}[\lambda x + \mu y] = \lambda \mathrm{SG}[x] + \mu \mathrm{SG}[y]$$

- **Théorème du retard** :
  Soit deux suites x et x_k telles que $\forall\, n \in \mathbb{Z},\ x_k(n) = x(n-k)$

$$X_k(z) = z^k X(z)$$

Démonstration :

$$\begin{aligned}
X_k(z) &= \sum_{n\in\mathbb{Z}} x_k(n) z^n \\
&= \sum_{n\in\mathbb{Z}} x(n-k) z^n \\
&\quad \text{changement de variable} \;:\; n-k \to n \\
&= \sum_{n\in\mathbb{Z}} x(n) z^{n+k} \\
&= z^k \sum_{n\in\mathbb{Z}} x(n) z^n
\end{aligned}$$

Calcul cplxité via séries génératrices (4)

Soit $x(n)$ la complexité de *fibo* à l'ordre n :

$$x(n) = 1 + x(n-1) + x(n-2)$$

$$x = \mathbb{1} + x_1 + x_2$$

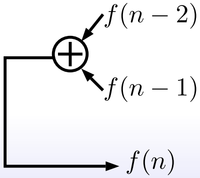
Calcul cplxité via séries génératrices (5)

$$X(z) = \frac{1}{(1-z)(1-z-z^2)}$$

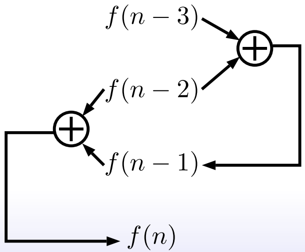
Calcul cplxité via séries génératrices (6)

$$X(z) = \frac{\alpha}{1-z} + \frac{\beta}{1-r_1 z} + \frac{\gamma}{1-r_2 z}$$

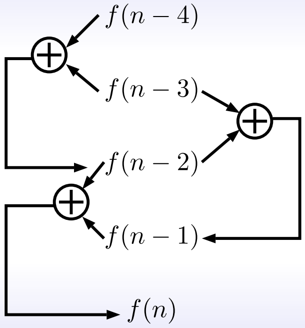
Nombre de Fibonacci : version itérative



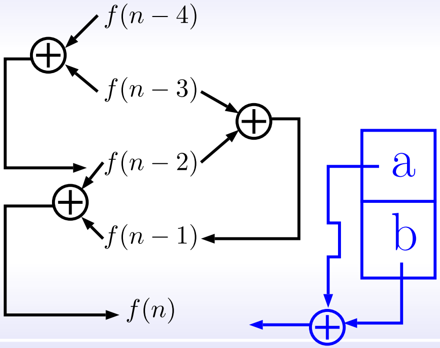
Nombre de Fibonacci : version itérative



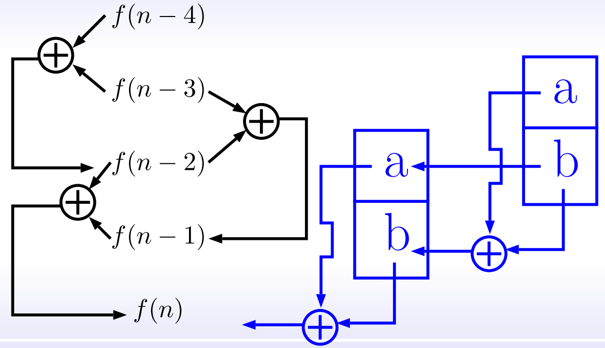
Nombre de Fibonacci : version itérative



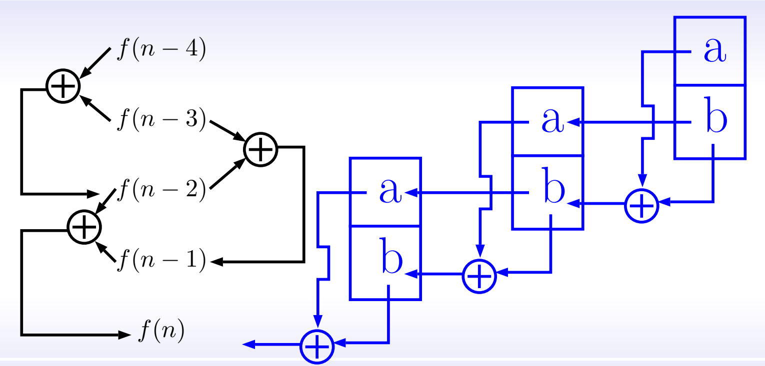
Nombre de Fibonacci : version itérative



Nombre de Fibonacci : version itérative



Nombre de Fibonacci : version itérative



Nombre de Fibonacci : version itérative

Algorithme 14 : FibIter

début

/* ENTRÉES: Un nombre n */
/* SORTIE: Le nombre de Fibonacci de n */
$a \leftarrow 1$
$b \leftarrow 1$
pour $k = 1$ *à* $n - 1$ **faire**
  $b \leftarrow a + b$
  $a \leftarrow b - a$
Retour b

fin

- Complexité faible comparée à celle de la fonction récursive
- Mais écriture non immédiate, elle demande de la reflexion

Récursion et induction

Recherche dichotomique

- Recherche par dichotomie d'un élément x dans un vecteur trié V de dimension n
- Nous avons déjà vu une version itérative
- Version récursive?

Recherche dichotomique itérative

Algorithme 15 : Recherche dichotomique dans un vecteur trié

début
- /* ENTRÉES: Un vecteur V de taille n, un élément x */
- /* SORTIE: i si x apparait au rang i de V, 0 si $x \notin V$ */
- $inf \leftarrow 1$, $sup \leftarrow n$, $i \leftarrow 0$, $trouve' \leftarrow faux$
- **tant que** $inf \leq sup$ et $non(trouve')$ **faire**
  - $med \leftarrow (inf + sup)$ **div** 2
  - **si** $x = V(med)$ **alors**
    - $i \leftarrow med$
    - $trouve' \leftarrow vrai$
  - **sinon**
    - **si** $x < V(med)$ **alors** $sup \leftarrow med - 1$
    - **sinon** $inf \leftarrow med + 1$
- **retourner** i

fin

Recherche dichotomique récursive

Algorithme 16 : RechDich

/* ENTRÉES: Un vecteur V de taille n, un élément x, inf (qui vaut 1 au premier appel), sup (qui vaut n au premier appel) */
/* SORTIE: i si x apparait au rang i de V, 0 si $x \notin V$ */
si $sup < inf$ **alors**
  retourner 0
sinon
  $med \leftarrow (inf + sup)$ **div** 2
  si $x = V(med)$ **alors retourner** med
  sinon
    si $x < V(med)$ **alors retourner** RechDich(V,x, inf, $med - 1$)
    sinon retourner RechDich(V,x, $med + 1$, sup)

C'est une récursivité terminale.

Complexité max de la recherche dichotomique (1)

- Opération significatives : on considère ici les **comparaisons**
- Un appel de *RechDich* sur V de taille n génère :
  - au maximum 2 comparaisons : $x = V(med)$ et $x < V(med)$
  - 1 appel de *RechDich* sur un sous-vecteur de taille maximale $\lceil n/2 \rceil$
- Complexité : $C_{max}(n) = 2 + C_{max}(\lceil n/2 \rceil)$

Complexité max de la recherche dichotomique (2)

- Supposons $n = 2^p$: $C_{max}(2^p) = 2 + C_{max}(2^{p-1})$
- Posons $x(p) = C_{max}(2^p)$
- $x(p) = 2 + x(p-1)$: suite arithmétique de raison 2
- $x(p) = 2p + x(0)$
- $C_{max}(n) = 2\log_2 n + C_{max}(1) = 2\log_2 n + 2$
- $C_{max}(n) = \Theta(\log_2(n))$

Complexité max de la recherche dichotomique (2)

- Supposons $n = 2^p$: $C_{max}(2^p) = 2 + C_{max}(2^{p-1})$
- Posons $x(p) = C_{max}(2^p)$
- $x(p) = 2 + x(p-1)$: suite arithmétique de raison 2
- $x(p) = 2p + x(0)$
- $C_{max}(n) = 2\log_2 n + C_{max}(1) = 2\log_2 n + 2$
- $C_{max}(n) = \Theta(\log_2(n))$

Complexité max de la recherche dichotomique (2)

- Supposons $n = 2^p$: $C_{max}(2^p) = 2 + C_{max}(2^{p-1})$
- Posons $x(p) = C_{max}(2^p)$
- $x(p) = 2 + x(p-1)$: suite arithmétique de raison 2
- $x(p) = 2p + x(0)$
- $C_{max}(n) = 2\log_2 n + C_{max}(1) = 2\log_2 n + 2$
- $C_{max}(n) = \Theta(\log_2(n))$

Complexité max de la recherche dichotomique (2)

- Supposons $n = 2^p$: $C_{max}(2^p) = 2 + C_{max}(2^{p-1})$
- Posons $x(p) = C_{max}(2^p)$
- $x(p) = 2 + x(p-1)$: suite arithmétique de raison 2
- $x(p) = 2p + x(0)$
- $C_{max}(n) = 2\log_2 n + C_{max}(1) = 2\log_2 n + 2$
- $C_{max}(n) = \Theta(\log_2(n))$

Complexité max de la recherche dichotomique (2)

- Supposons $n = 2^p$: $C_{max}(2^p) = 2 + C_{max}(2^{p-1})$
- Posons $x(p) = C_{max}(2^p)$
- $x(p) = 2 + x(p-1)$: suite arithmétique de raison 2
- $x(p) = 2p + x(0)$
- $C_{max}(n) = 2\log_2 n + C_{max}(1) = 2\log_2 n + 2$
- $C_{max}(n) = \Theta(\log_2(n))$

Complexité max de la recherche dichotomique (2)

- Supposons $n = 2^p$: $C_{max}(2^p) = 2 + C_{max}(2^{p-1})$
- Posons $x(p) = C_{max}(2^p)$
- $x(p) = 2 + x(p-1)$: suite arithmétique de raison 2
- $x(p) = 2p + x(0)$
- $C_{max}(n) = 2\log_2 n + C_{max}(1) = 2\log_2 n + 2$
- $C_{max}(n) = \Theta(\log_2(n))$

Récursion et induction

Tri récursif

- Tri récursif :
  1. diviser les données en deux “presque moitiés” ;
  2. appeler sur chaque moitié.
- Nous allons voir deux tris récursifs :
  - Tri par fusion (Merge Sort)
  - Tri rapide (Quick Sort)
- Paradigme **Divide-and-Conquer** (diviser pour régner)

Divide and Conquer =

1. **Diviser** :
   Si les données sont trop grandes pour être traitées de façon directe, alors les diviser en deux ou plusieurs sous-ensemble disjoints ;
2. **Appliquer récursivement** le principe “diviser pour régner” sur chaque sous-ensemble ;
3. **Conquérir** : Fusionner les solutions des sous-ensembles pour obtenir la solution au problème initial.

Tri par fusion (Merge Sort)

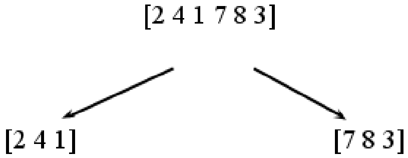
Soit $V : [1, n] \to E$ un vecteur non trié.
Tri par fusion= 3 étapes :

1. **Diviser** V en 2 sous-vecteurs V_1 et V_2 d'environ $\frac{n}{2}$ éléments chacun
2. **Appliquer récursivement** le tri par fusion sur V_1 et V_2
3. **Conquérir :** fusionner V_1 et V_2 dans un vecteur trié :
    - V_1 et V_2 sont triés
    - Répéter :
      comparer le plus petit élément de V_1 avec le plus petit élément de V_2, et insérer le plus petit des deux dans V

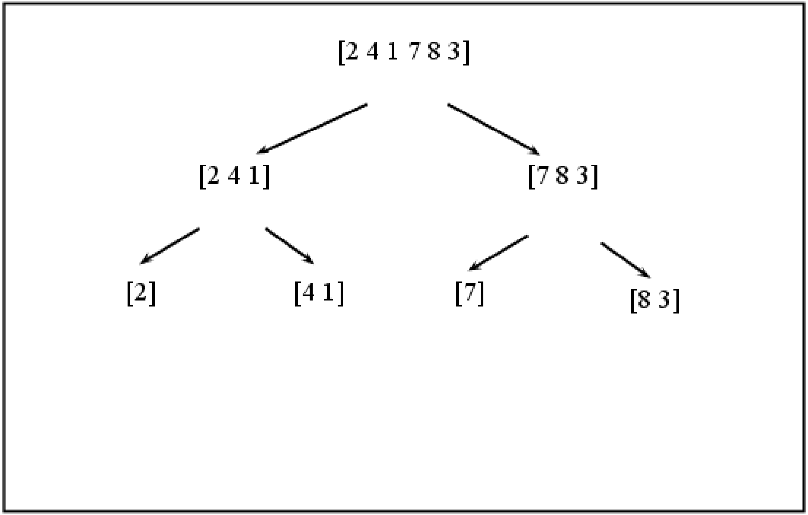
Tri par fusion (Merge Sort)

V = [2 4 1 7 8 3]

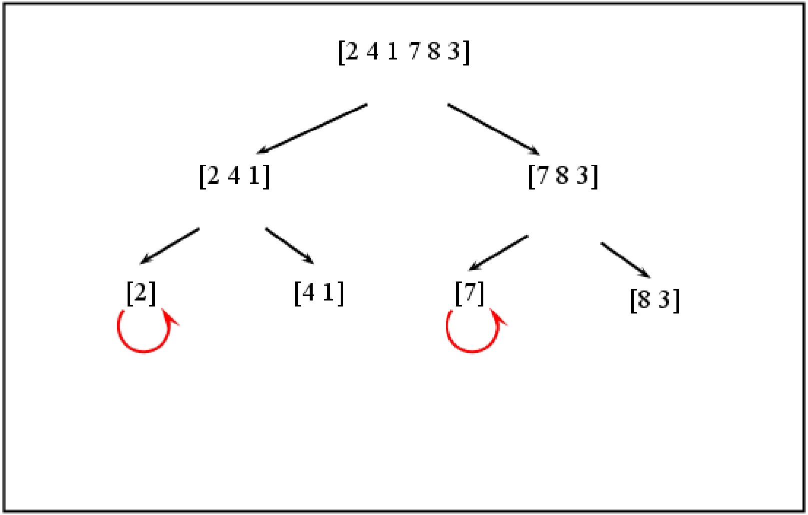
Tri par fusion (Merge Sort)



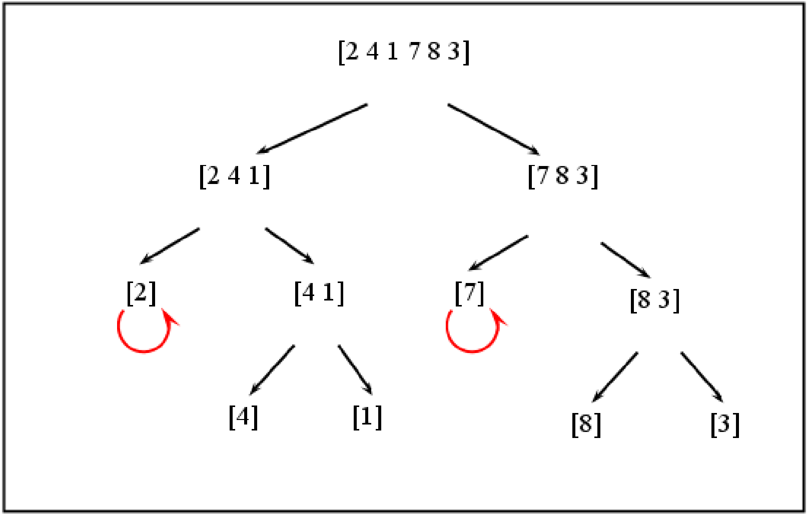
Tri par fusion (Merge Sort)



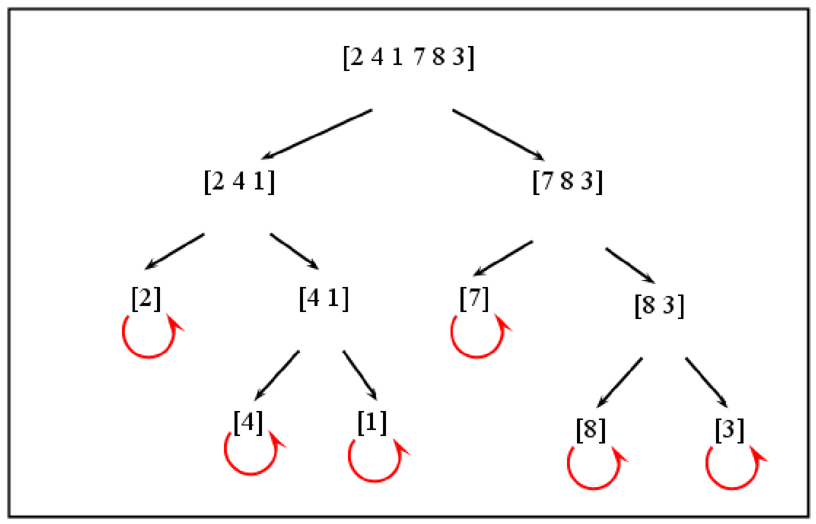
Tri par fusion (Merge Sort)



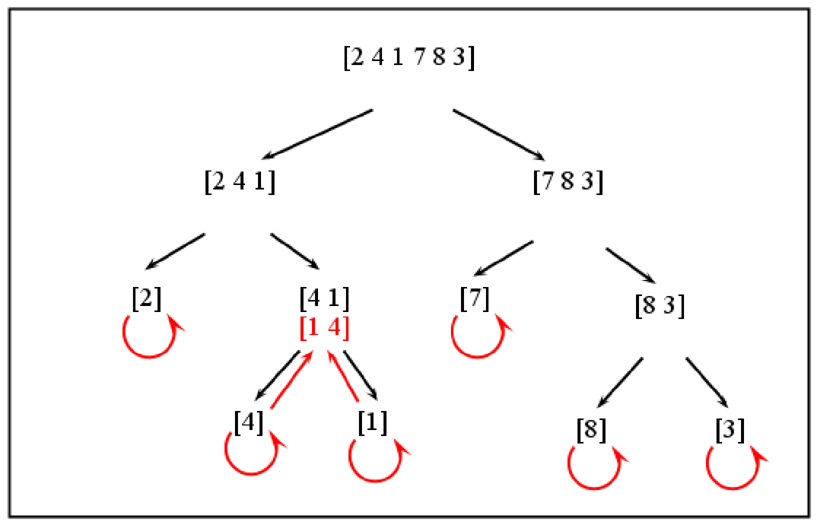
Tri par fusion (Merge Sort)



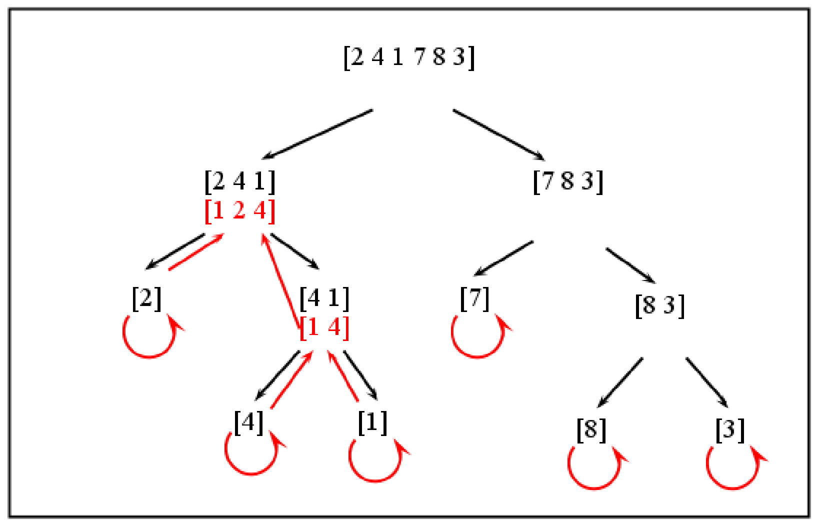
Tri par fusion (Merge Sort)



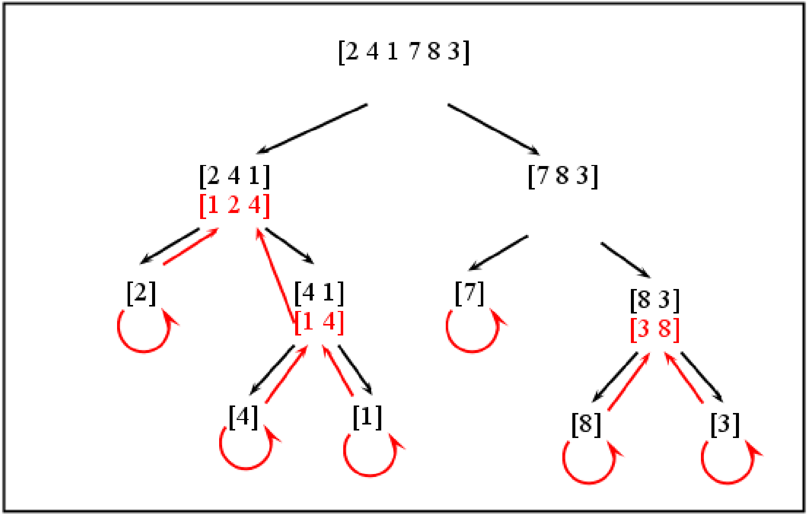
Tri par fusion (Merge Sort)



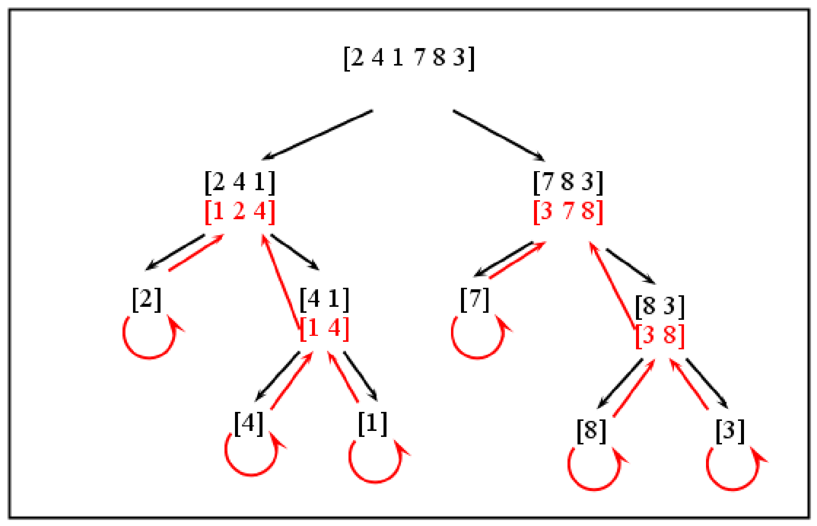
Tri par fusion (Merge Sort)



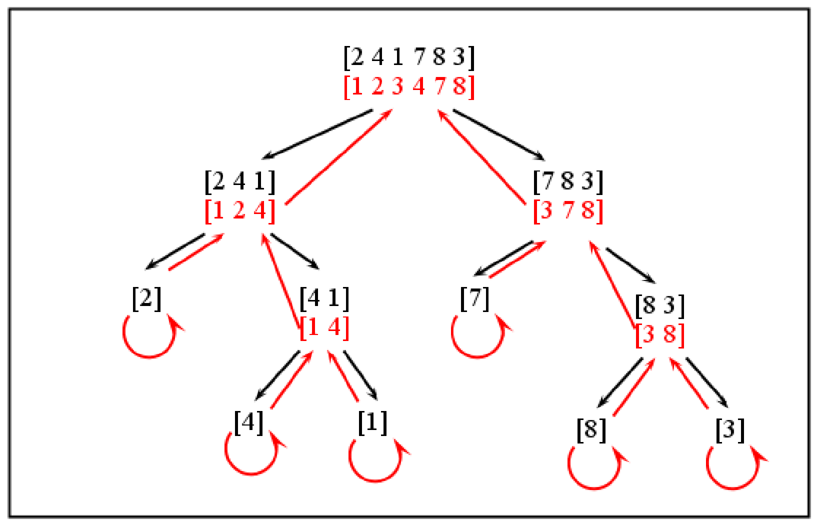
Tri par fusion (Merge Sort)



Tri par fusion (Merge Sort)



Tri par fusion (Merge Sort)



Tri par fusion (Merge Sort)

L'exécution du tri par fusion est représentée par un arbre binaire :

- Chaque nœud représente un appel récursif du tri par fusion et emmagasine
  - Le vecteur non trié avant l'exécution de l'algorithme
  - Le vecteur trié à la fin de l'exécution
- La racine est l'appel initial
- Les feuilles sont les appels pour les vecteurs de taille 1

Tri par fusion (Merge Sort)

Algorithme 17 : TriFusion

début
- /* ENTRÉES: Un vecteur V de taille n */
- /* SORTIE: Le vecteur V trié */
- **si** $n > 1$ **alors**
  - $p \leftarrow n$ div 2
  - **pour** $i = 1$ *à* p **faire**
    - $V_1(i) \leftarrow V(i)$
  - **pour** $i = p + 1$ *à* n **faire**
    - $V_2(i - p) \leftarrow V(i)$
  - $V_1 \leftarrow$ TriFusion(V_1, p)
  - $V_2 \leftarrow$ TriFusion(V_2, $n - p$)
  - Retour Fusion(V_1, V_2)
- **sinon**
  - Retour V

fin

Tri par fusion (Merge Sort)

Algorithme 18 : Fusion

début

/* ENTRÉES: Deux vecteurs triés V_1 et V_2 de taille p et q */
/* SORTIE: Un vecteur V trié */
$i \leftarrow 1$; $j \leftarrow 1$; $k \leftarrow 1$
tant que $j \leq p$ *et* $k \leq q$ **faire**
  si $V_1(j) \leq V_2(k)$ **alors** $V(i) \leftarrow V_1(j)$; $j \leftarrow j + 1$
  sinon $V(i) \leftarrow V_2(k)$; $k \leftarrow k + 1$
  $i \leftarrow i + 1$
pour $t = j$ *à* p **faire**
  $V(i) \leftarrow V_1(t)$; $i \leftarrow i + 1$
pour $t = k$ *à* q **faire**
  $V(i) \leftarrow V_2(t)$; $i \leftarrow i + 1$
Retour V

fin

Complexité du tri par fusion (1)

- Opération significatives : on considère ici les **comparaisons**
- Un appel de *TriFusion* sur V de taille n génère :
  - 1 appel de *TriFusion* sur V_1 de taille $p = \lfloor n/2 \rfloor$
  - 1 appel de *TriFusion* sur V_2 de taille $n - p = \lceil n/2 \rceil$
  - 1 appel à *Fusion* : λn comparaisons, $1/2 < \lambda < 1$
- Complexité : $C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + \lambda n$

Complexité du tri par fusion (2)

- Supposons $n = 2^p$: $C(2^p) = 2C(2^{p-1}) + \lambda 2^p$
- Posons $x(p) = C(2^p)$

Complexité du tri par fusion (3)

$$X(z) = \frac{\lambda}{(1-2z)^2}$$

Or $\frac{1}{(1-z)^2} = \sum_{p \geq 0}(p+1)z^p$
Ainsi,

Tri rapide (Quick Sort)

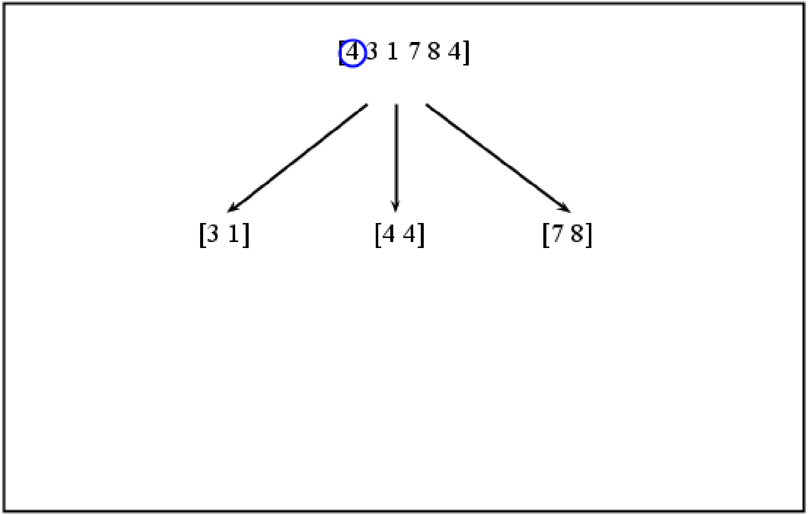
Soit $V : [1, n] \rightarrow E$ un vecteur non trié.
Tri rapide $=$ 3 étapes :

1. **Diviser :** choisir un élément x au hasard (appelé **pivot**), et diviser V en 3 sous-vecteurs L et E et G tels que :
   - L : contient les éléments plus petits que x
   - E : contient les éléments égaux à x
   - G : contient les éléments plus grands que x
2. **Appliquer récursivement** le tri rapide sur L et G
3. **Conquérir :** fusionner L, E et G dans un vecteur trié

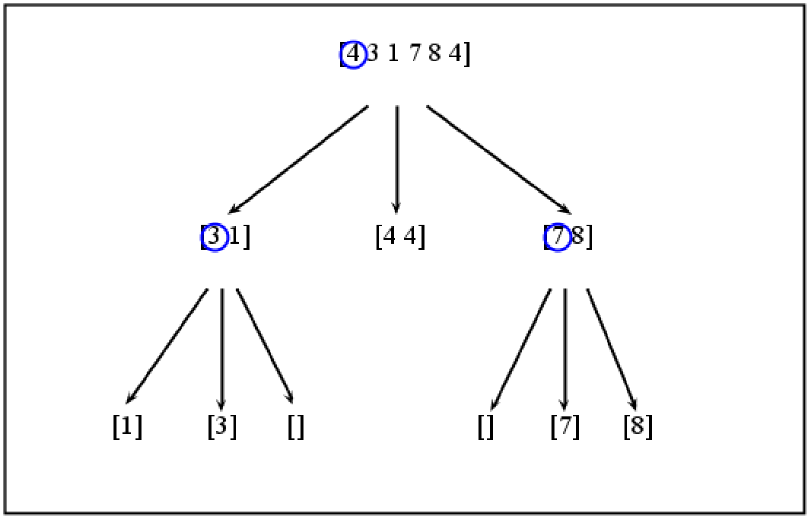
Tri rapide (Quick Sort)

[4 3 1 7 8 4]

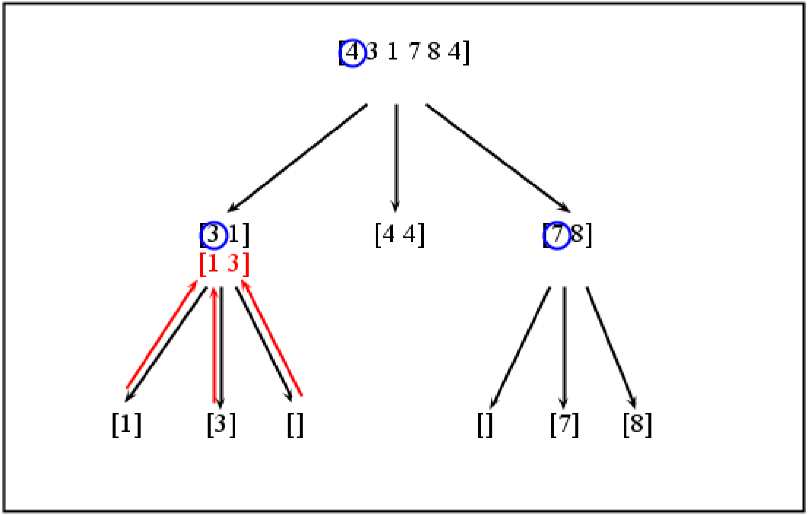
Tri rapide (Quick Sort)



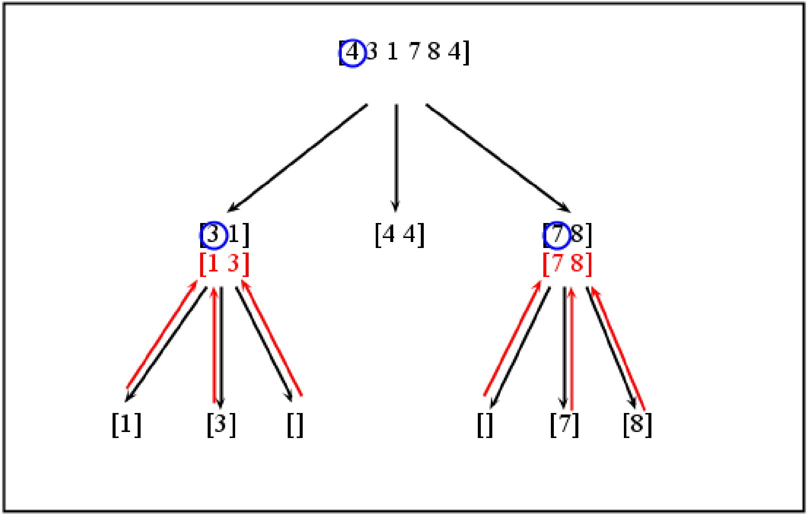
Tri rapide (Quick Sort)



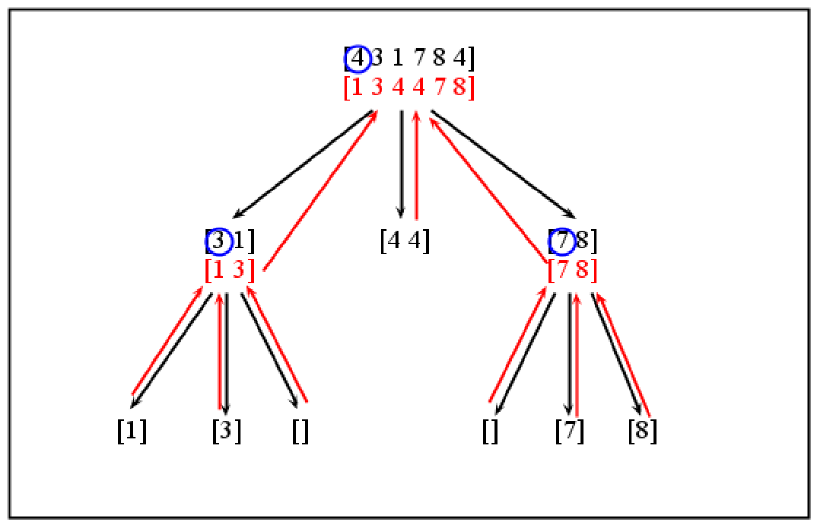
Tri rapide (Quick Sort)



Tri rapide (Quick Sort)



Tri rapide (Quick Sort)



Tri rapide (Quick Sort)

L'exécution du tri rapide est représentée par un arbre :

- Chaque nœud représente un appel récursif du tri rapide et emmagasine
  - Le vecteur non trié avant l'exécution de l'algorithme et son pivot
  - Le vecteur trié à la fin de l'exécution
- La racine est l'appel initial
- Les feuilles sont les appels pour les vecteurs de taille 1

Tri rapide (Quick Sort)

Algorithme 19 : DivQuickSort

début

/* ENTRÉES: Un vecteur V de taille n, l'indice p du pivot */
/* SORTIES: Trois vecteurs L, E et G */
$x \leftarrow V(p)$; $\ell \leftarrow 0$; $e \leftarrow 0$; $g \leftarrow 0$
pour $i = 1$ *à* n **faire**
  si $V(i) < x$ **alors** $\ell \leftarrow \ell + 1$; $L(\ell) \leftarrow V(i)$
  sinon si $V(i) = x$ **alors** $e \leftarrow e + 1$; $E(e) \leftarrow V(i)$
  sinon $g \leftarrow g + 1$; $G(g) \leftarrow V(i)$
Retour (L, ℓ, E, e, G, g)

fin

Tri rapide (Quick Sort)

Algorithme 20 : TriRapide

début

/* ENTRÉES: Un vecteur V de taille n */
/* SORTIE: Le vecteur V trié */
$(L, \ell, E, e, G, g) \leftarrow DivQuickSort(V, n, 1)$
si $\ell > 1$ **alors** $L \leftarrow$ TriRapide(L, ℓ)
si $g > 1$ **alors** $G \leftarrow$ TriRapide(G, g)
pour $i = 1$ *à* ℓ **faire** $V(i) \leftarrow L(i)$
pour $i = (\ell + 1)$ *à* $(e + \ell)$ **faire** $V(i) \leftarrow E(i - \ell)$
pour $i = (\ell + e + 1)$ *à* $(g + e + \ell)$ **faire** $V(i) \leftarrow G(i - \ell - e)$
Retour V

fin

Complexité du tri rapide

- Opération significatives : on considère ici les **comparaisons**
- Complexité temporelle (démonstration en TD) :
  - Complexité moyenne : $\Theta(n \log n)$
  - Complexité pire cas : $\Theta(n^2)$
- Malgré cette très mauvaise complexité en pire cas,
  c'est un des tris les plus rapides en pratique, et un des plus utilisé
- Plus économe en mémoire que le tri par fusion

Comparaison des algorithmes de tri

- On compare la complexité moyenne en terme de comparaison
- Algorithmes lents :
  - tri par comptage ($\Theta(n^2)$)
  - tri à bulles ($\Theta(n^2)$)
  - tri par sélection ($\Theta(n^2)$)
  - tri par insertion ($\Theta(n^2)$)
  - ...
- Algorithmes rapides :
  - tri fusion ($\Theta(n \log n)$) (mais gourmand en place mémoire)
  - tri rapide ($\Theta(n \log n)$) (mais pas au pire des cas)
  - ...
- On peut montrer que la complexité temporelle d'un algorithme de tri ne peut pas être meilleure en moyenne et au pire cas que $n \log n$.

Récursion et induction

Tours de Hanoï

- Soient
  - n plateaux de taille croissante (numérotés 1 à n), et
  - 1 tige A (arrivée)
  - 1 tige D (départ)
  - 1 tige X (auxiliaire)
- Au départ les plateaux sont empilés sur D par taille croissante, le plus grand en bas
- On veut placer ces plateaux, dans le même ordre, sur la tige A
- On ne déplace qu'un plateau à la fois
- On ne peut poser un plateau que sur un plateau de taille supérieure

Tours de Hanoï : Principe de récurrence

- Soit $Han(p, D, A)$ la procédure qui consiste à déplacer les p premiers plateaux de la tige D à la tige A
- Principe de récurrence :
  si on sait faire $Han(p-1, D', A')$ alors on a gagné.
  En effet, il suffit d'effectuer :
  - $Han(p-1, D, X)$: déplace les $p-1$ premiers plateaux de D à X
  - $Deplace(p, D, A)$: déplace le plateau p de D à A
  - $Han(p-1, X, A)$: déplace les $p-1$ premiers plateaux de X à A
- cas facile : 1 seul plateau ($p = 1$)

Tours de Hanoï

Algorithme 21 : Hanoi

début

/* ENTRÉES: Un entier $p =$ nombre de plateaux à déplacer, D, A et X des symboles désignant respectivement les tiges de départ, d'arrivée et auxiliaire */

/* SORTIE: Les p plateaux sont sur la tige A */

si $p = 1$ **alors**

  $deplace(1, D, A)$

  / déplace le plateau 1 de D à A : un seul mouvement*/*

sinon

  $Hanoi(p-1, D, X, A)$

  $deplace(p, D, A)$

  $Hanoi(p-1, X, A, D)$

fin

Tours de Hanoï : complexité

- Soit f_p le nombre de déplacements nécessaires pour un jeu de p plateaux
- Si $p = 1$ alors $f_p = 1$
- Sinon il faut
  - f_{p-1} déplacements pour chaque appel à *Hanoi* (il y en a deux)
  - 1 déplacement pour chaque appel à *deplace*
- Donc $f_p = 2f_{p-1} + 1$, avec $f_1 = 1$
  - $f_p + 1 = 2(f_{p-1} + 1)$
  - $f_p + 1 = 2^x(f_{p-x} + 1) = 2^{p-1}(f_1 + 1) = 2^{p-1} * 2 = 2^p$

Tours de Hanoï : complexité

- Pour déplacer p plateaux il faut donc 2^p mouvements
- Le temps de calcul double lorsqu'on ajoute un plateau
- Inutile donc d'essayer $Hanoi(50, 0, 1)$, il faudrait 10^{15} déplacements
  - A raison d'1 déplacement/seconde, il faudrait 317 000 siècles
  - A raison d'1 déplacement/ms, il faudrait 317 siècles
- Exemple de récursivité non terminale