Complexité (suite)

Philippe Lac
(philippe.lac@ac-clermont.fr)
Malika More
(malika.more@u-clermont1.fr)

IREM Clermont-Ferrand

Stage Algorithmique

Année 2010-2011

1. Les nombres de Fibonacci

2. Les tris

3. Pour aller plus loin

1 Les nombres de Fibonacci

2 Les tris

3 Pour aller plus loin

Nombres de Fibonacci

Définition

- $F_0 = 1$
- $F_1 = 1$
- $F_n = F_{n-2} + F_{n-1}$ pour $n \geq 2$

Question

Quelle est la complexité des algorithmes de calcul des nombres de Fibonacci ?

Algorithme récursif

Fonction `Fib` (n)

début

  si $n < 2$ **alors**

    retourner : 1

  fin

  retourner : Fib($n-1$)+Fib($n-2$)

fin

Algorithme récursif

Fonction `Fib`(n)

---

début

  si $n < 2$ **alors**

    retourner : 1

  fin

  retourner : Fib($n-1$)+Fib($n-2$)

fin

Pendant le calcul de F_n

- a_n : nombre d'appels à la fonction `Fib`
- s_n : nombre d'additions

Exemples

- $a_0 = a_1 = 0$
- $a_2 = 2$
- $a_3 = 4$
- $s_0 = s_1 = 0$
- $s_2 = 1$
- $s_3 = 2$
- Etc.

Algorithme récursif

Fonction `Fib`(n)

début
  si $n < 2$ **alors**
    retourner : 1
  fin
  retourner : Fib($n-1$)+Fib($n-2$)
fin

Pendant le calcul de F_n

- a_n : nombre d'appels à la fonction `Fib`
- s_n : nombre d'additions

Relations de récurrence

- $a_n = 1 + a_{n-1} + 1 + a_{n-2}$
- $s_n = s_{n-1} + 1 + s_{n-2}$

Algorithme récursif

Fonction `Fib`(n)

début
  si $n < 2$ **alors**
    retourner : 1
  fin
  retourner : Fib($n-1$)+Fib($n-2$)
fin

Suites auxiliaires

- $a'_n = a_n + 2$
- $s'_n = s_n + 1$

Des grands classiques

- $a'_0 = a'_1 = 2$
- $a'_n = a'_{n-1} + a'_{n-2}$
- $s'_0 = s'_1 = 1$
- $s'_n = s'_{n-1} + s'_{n-2}$

Algorithme récursif

Fonction `Fib` (n)

début

  si $n < 2$ **alors**

    retourner : 1

  fin

  retourner : Fib$(n-1)$+Fib$(n-2)$

fin

Suites auxiliaires

- $a'_n = a_n + 2$
- $s'_n = s_n + 1$

Dérécursivisation

- $a'_n = K_a \times \left(\frac{1+\sqrt{5}}{2}\right)^n$ pour $n \geq 2$
- $s'_n = K_s \times \left(\frac{1+\sqrt{5}}{2}\right)^n$ pour $n \geq 2$

Algorithme récursif

Fonction `Fib(`n`)`

début

  si $n < 2$ **alors**

    retourner : 1

  fin

  retourner : Fib($n-1$)+Fib($n-2$)

fin

Pendant le calcul de F_n

- a_n : nombre d'appels à la fonction `Fib`
- s_n : nombre d'additions

Pour $n \geq 2$

- $a_n = K_a \times \left(\frac{1+\sqrt{5}}{2}\right)^n - 2$
- $s_n = K_s \times \left(\frac{1+\sqrt{5}}{2}\right)^n - 1$

Algorithme récursif

Fonction `Fib`(n)

---

début
  si $n < 2$ **alors**
    retourner : 1
  fin
  retourner : Fib($n-1$)+Fib($n-2$)
fin

Hypothèses

- Chaque appel à `Fib` prend un temps constant
- Chaque addition prend un temps constant
- (À prendre avec précautions)

Temps de calcul de F_n

- Combinaison de a_n et s_n
- Du type $K \times \left(\frac{1+\sqrt{5}}{2}\right)^n$
- Croissance de type exponentiel par rapport à n

Algorithme récursif

Fonction `Fib(`n`)`

début
  si $n < 2$ **alors**
    retourner : 1
  fin
  retourner : Fib($n-1$)+Fib($n-2$)
fin

Vérification expérimentale

$\longrightarrow$ `Scilab`

Algorithme itératif

Fonction `Fib`(n)

début
  si $n < 2$ **alors**
    retourner : 1
  sinon
    Donner à x la valeur 1
    Donner à y la valeur 1
    for i **de** 2 **à** n **do**
      Donner à *temp* la valeur $x + y$
      Donner à x la valeur y
      Donner à y la valeur *temp*
    end
    retourner : y
  fin
fin

Algorithme itératif

```
Fonction Fib(n)
début
    si n < 2 alors
        retourner : 1
    sinon
        Donner à x la valeur 1
        Donner à y la valeur 1
        for i de 2 à n do
            Donner à temp la valeur x + y
            Donner à x la valeur y
            Donner à y la valeur temp
        end
        retourner : y
    fin
fin
```

Pendant le calcul de F_n

- Dans la boucle : 1 addition et 3 affectations
- $n - 1$ passages dans la boucle
- Au total : $n - 1$ additions et $3(n - 1)$ affectations

Hypothèses

- Chaque affectation prend un temps constant
- Chaque addition prend un temps constant
- (À prendre avec précautions)

Algorithme itératif

Fonction `Fib`(n)

début
  si $n < 2$ **alors**
    retourner : 1
  sinon
    Donner à x la valeur 1
    Donner à y la valeur 1
    for i **de** *2* **à** n **do**
      Donner à *temp* la valeur $x + y$
      Donner à x la valeur y
      Donner à y la valeur *temp*
    end
    retourner : y
  fin
fin

Pendant le calcul de F_n

- Dans la boucle : 1 addition et 3 affectations
- $n - 1$ passages dans la boucle
- Au total : $n - 1$ additions et $3(n - 1)$ affectations

Hypothèses

- Chaque affectation prend un temps constant
- Chaque addition prend un temps constant
- (À prendre avec précautions)

Algorithme itératif

Fonction `Fib`(n)

début
- **si** $n < 2$ **alors**
  - **retourner** : 1
- **sinon**
  - Donner à x la valeur 1
  - Donner à y la valeur 1
  - **for** i **de** 2 **à** n **do**
    - Donner à *temp* la valeur $x + y$
    - Donner à x la valeur y
    - Donner à y la valeur *temp*
  - **end**
  - **retourner** : y
- **fin**

fin

Pendant le calcul de F_n

- Dans la boucle : 1 addition et 3 affectations
- $n - 1$ passages dans la boucle
- Au total : $n - 1$ additions et $3(n - 1)$ affectations

Temps de calcul de F_n

- Du type $K \times n$
- Croissance de type linéaire par rapport à n

Vérification expérimentale

$\longrightarrow$ `Scilab`

Intermède

Remarque

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} b \\ a+b \end{pmatrix}$$

Conséquence

$$\begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n-1} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \text{ pour } n \geq 2$$

Multiplication matricielle

Nombre d'opérations élémentaires

$$\begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} \begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{pmatrix}$$

$$=$$

$$\begin{pmatrix} a_{1,1}b_{1,1} + a_{1,2}b_{2,1} & a_{1,1}b_{2,1} + a_{1,2}b_{2,2} \\ a_{2,1}b_{1,1} + a_{2,2}b_{2,1} & a_{2,1}b_{1,2} + a_{2,2}b_{2,2} \end{pmatrix}$$

$\longrightarrow$ 4 additions et 8 multiplications

Version matricielle

Fonction `Fib`(n)

début

  si $n < 2$ **alors**

    retourner : 1

  sinon

    Donner à x la valeur $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$

    Donner à y la valeur $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$

    Donner à x la valeur `ExpRap`$(x, n-1)$

    Donner à y la valeur xy

    Donner à z la valeur $y[2, 1]$

    retourner : z

  fin

fin

Version matricielle

Fonction `Fib`(n)

début
  si $n < 2$ **alors**
    retourner : 1
  sinon
    Donner à x la valeur $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$
    Donner à y la valeur $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$
    Donner à x la valeur `ExpRap`($x, n-1$)
    Donner à y la valeur xy
    Donner à z la valeur $y[2, 1]$
    retourner : z
  fin
fin

Pendant le calcul de F_n

- Calcul de `ExpRap`($x, n-1$)
- Calcul de xy
- Affectations, etc.

Calcul de `ExpRap`($x, n-1$)

Dans le pire des cas :

- $|n-1|$ divisions
- $2|n-1|$ multiplications
- Attention : multiplications matricielles !

Version matricielle

Fonction `Fib`(n)

début

  si $n < 2$ **alors**

    retourner : 1

  sinon

    Donner à x la valeur $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$

    Donner à y la valeur $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$

    Donner à x la valeur `ExpRap`$(x, n-1)$

    Donner à y la valeur xy

    Donner à z la valeur $y[2, 1]$

    retourner : z

  fin

fin

Pendant le calcul de F_n

- Calcul de `ExpRap`$(x, n-1)$
- Calcul de xy
- Affectations, etc.

Calcul de `ExpRap`$(x, n-1)$

Dans le pire des cas :

- $|n-1|$ divisions
- $2|n-1|$ multiplications
- Attention : multiplications matricielles !

Version matricielle

Fonction `Fib`(n)

début

- **si** $n < 2$ **alors**
  - **retourner** : 1
- **sinon**
  - Donner à x la valeur $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$
  - Donner à y la valeur $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$
  - Donner à x la valeur $\texttt{ExpRap}(x, n-1)$
  - Donner à y la valeur xy
  - Donner à z la valeur $y[2, 1]$
  - **retourner** : z
- **fin**

fin

Pendant le calcul de F_n

- Calcul de $\texttt{ExpRap}(x, n-1)$
- Calcul de xy
- Affectations, etc.

Calcul de $\texttt{ExpRap}(x, n-1)$

Dans le pire des cas :

- $|n-1|$ divisions
- $8|n-1|$ additions
- $16|n-1|$ multiplications

Version matricielle

```
Fonction Fib(n)
début
    si n < 2 alors
        retourner : 1
    sinon
        Donner à x la valeur ((0, 1), (1, 1))
        Donner à y la valeur ((1), (1))
        Donner à x la valeur ExpRap(x, n − 1)
        Donner à y la valeur xy
        Donner à z la valeur y[2, 1]
        retourner : z
    fin
fin
```

où la matrice donnée à x est $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ et celle donnée à y est $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$.

Pendant le calcul de F_n

- Calcul de ExpRap$(x, n-1)$
- Calcul de xy
- Affectations, etc.

Hypothèses

- Chaque affectation prend un temps constant
- Chaque opération arithmétique prend un temps constant
- (À prendre avec précautions)

Version matricielle

Fonction `Fib`(n)

début

  si $n < 2$ **alors**

    retourner : 1

  sinon

    Donner à x la valeur $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$

    Donner à y la valeur $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$

    Donner à x la valeur $\texttt{ExpRap}(x, n-1)$

    Donner à y la valeur xy

    Donner à z la valeur $y[2,1]$

    retourner : z

  fin

fin

Pendant le calcul de F_n

- Calcul de $\texttt{ExpRap}(x, n-1)$
- Calcul de xy
- Affectations, etc.

Temps de calcul de F_n

- Du type $K \times |n|$
- Croissance de type logarithmique par rapport à n

Vérification expérimentale

$\longrightarrow$ `Scilab`

Encore Fibonacci

```
Fonction Fib(n, adresse du tableau T)
début
    % F_n pas encore calculé %
    si T[n] = 0 alors
        si n ≤ 1 alors
            Donner à T[n] la valeur 1
        sinon
            Donner à T[n] la valeur Fib(n − 1,
            adresse de T)+Fib(n − 2, adresse
            de T)
        fin
    fin
    % F_n contenu dans T[n] %
    retourner : T[n]
fin
```

Encore Fibonacci

Fonction `Fib(`*n,adresse du tableau T*`)`

début

  `%` F_n `pas encore calculé %`

  si $T[n] = 0$ **alors**

    si $n \leq 1$ **alors**

      Donner à $T[n]$ la valeur 1

    sinon

      Donner à $T[n]$ la valeur `Fib`$(n-1$, adresse de $T)$+`Fib`$(n-2$, adresse de $T)$

    fin

  fin

  `%` F_n `contenu dans` $T[n]$ `%`

  retourner : $T[n]$

fin

- Un algorithme récursif qui évite de calculer plusieurs fois les mêmes valeurs en remplissant un tableau avec les valeurs déjà calculées
- Sa complexité est en $\mathcal{O}(n)$, puisque chaque case du tableau est remplie une seule fois

Encore Fibonacci

Fonction `Fib(`*n,adresse du tableau T*`)`

début

  `%` F_n `pas encore calculé %`

  si $T[n] = 0$ **alors**

    si $n \leq 1$ **alors**

      Donner à $T[n]$ la valeur 1

    sinon

      Donner à $T[n]$ la valeur `Fib`$(n-1$, adresse de $T)$+`Fib`$(n-2$, adresse de $T)$

    fin

  fin

  `%` F_n `contenu dans` $T[n]$ `%`

  retourner : $T[n]$

fin

- Cet algorithme utilise et modifie un tableau créé en dehors de la fonction elle-même
  - Variable »globale »
  - Passage « par valeur » du paramètre T
- Cet algorithme restera théorique parce-qu'il semble que `Scilab` n'autorise pas le passage par adresse, ni la modification des variables globales...

Fibonacci avancé

Variante de la définition par récurrence

- $F_0 = 1$
- $F_1 = 1$
- Pour tout $k \geq 1$
  - $F_{2k} = F_k^2 + F_{k-1}^2$
  - $F_{2k+1} = (2F_{k-1} + F_k) \times F_k = (2F_{k+1} - F_k) \times F_k$

Remarque

Preuve facile par récurrence

Toujours Fibonacci

```
Fonction Fib(n)
début
    si n ≤ 1 alors
        Donner à F la valeur 1
    sinon
        si n impair alors
            Donner à k la valeur (n−1)/2
            Donner à F1 la valeur Fib(k − 1)
            Donner à F2 la valeur Fib(k)
            Donner à F la valeur (2F1 + F2) × F2
        sinon
            Donner à k la valeur n/2
            Donner à F1 la valeur Fib(k − 1)
            Donner à F2 la valeur Fib(k)
            Donner à F la valeur F1^2 + F2^2
        fin
    fin
    retourner : F
fin
```

Toujours Fibonacci

Fonction `Fib`(n)
début
  si $n \leq 1$ **alors**
    Donner à F la valeur 1
  sinon
    si *n impair* **alors**
      Donner à k la valeur $\frac{n-1}{2}$
      Donner à F_1 la valeur `Fib`$(k-1)$
      Donner à F_2 la valeur `Fib`(k)
      Donner à F la valeur $(2F_1 + F_2) \times F_2$
    sinon
      Donner à k la valeur $\frac{n}{2}$
      Donner à F_1 la valeur `Fib`$(k-1)$
      Donner à F_2 la valeur `Fib`(k)
      Donner à F la valeur $F_1^2 + F_2^2$
    fin
  fin
  retourner : F
fin

- Un algorithme récursif dans lequel la variable n est divisée par 2 à chaque appel récursif
- La hauteur de la pile des appels récursifs est donc en $\mathcal{O}(\log n)$
- Mais on calcule plusieurs fois les mêmes valeurs, comme dans l'algorithme récursif naïf
- L'ensemble des appels récursifs forme un arbres binaire, dont le nombre de nœuds est exponentiel en sa hauteur, i.e. en $\mathcal{O}(n)$

Toujours Fibonacci

Fonction `Fib`(n)

début
  si $n \leq 1$ **alors**
    Donner à F la valeur 1
  sinon
    si *n impair* **alors**
      Donner à k la valeur $\frac{n-1}{2}$
      Donner à F_1 la valeur $\texttt{Fib}(k-1)$
      Donner à F_2 la valeur $\texttt{Fib}(k)$
      Donner à F la valeur $(2F_1 + F_2) \times F_2$
    sinon
      Donner à k la valeur $\frac{n}{2}$
      Donner à F_1 la valeur $\texttt{Fib}(k-1)$
      Donner à F_2 la valeur $\texttt{Fib}(k)$
      Donner à F la valeur $F_1^2 + F_2^2$
    fin
  fin
  retourner : F
fin

- La complexité de cet algorithme est du même ordre que le nombre d'appels récursifs, donc en $\mathcal{O}(n)$
- On peut se ramener à une complexité en $\mathcal{O}(\log n)$ en utilisant un tableau global pour stocker les valeurs déjà calculées, comme précédemment

Fibonacci, un dernier

Fonction `Fib2`(n)

début

- **si** $n = 0$ **alors**
  - Donner à u la valeur $(0, 1)$
- **sinon**
  - **si** $n = 1$ **alors**
    - Donner à u la valeur $(1, 1)$
  - **sinon**
    - **si** n *impair* **alors**
      - Donner à k la valeur $\frac{n-1}{2}$
      - Donner à v la valeur `Fib2`(k)
      - Donner à u la valeur $(v[1]^2 + v[2]^2, (2v[1] + v[2]) \times v[2])$
    - **sinon**
      - Donner à k la valeur $\frac{n}{2}$
      - Donner à v la valeur `Fib2`(k)
      - Donner à u la valeur $((2v[2] - v[1]) \times v[2], v[1]^2 + v[2]^2)$
    - **fin**
  - **fin**
- **fin**
- **retourner** : u

fin

Fibonacci, un dernier

Fonction `Fib2`(n)

début

  si $n = 0$ **alors**

    Donner à u la valeur $(0, 1)$

  sinon

    si $n = 1$ **alors**

      Donner à u la valeur $(1, 1)$

    sinon

      si *n impair* **alors**

        Donner à k la valeur $\frac{n-1}{2}$

        Donner à v la valeur `Fib2`(k)

        Donner à u la valeur $(v[1]^2 + v[2]^2, (2v[1] + v[2]) \times v[2])$

      sinon

        Donner à k la valeur $\frac{n}{2}$

        Donner à v la valeur `Fib2`(k)

        Donner à u la valeur $((2v[2] - v[1]) \times v[2], v[1]^2 + v[2]^2)$

      fin

    fin

  fin

  retourner : u

fin

- Un algorithme récursif dans lequel on calcule deux valeurs successives (F_{n-1}, F_n)
- La hauteur de la pile des appels récursifs est toujours en $\mathcal{O}(\log n)$
- Mais on ne calcule plus plusieurs fois les mêmes valeurs
- La complexité de cet algorithme est donc en $\mathcal{O}(\log n)$

Fibonacci, un dernier

Fonction `Fib2`(n)

début
- **si** $n = 0$ **alors**
  - Donner à u la valeur $(0, 1)$
- **sinon**
  - **si** $n = 1$ **alors**
    - Donner à u la valeur $(1, 1)$
  - **sinon**
    - **si** n *impair* **alors**
      - Donner à k la valeur $\frac{n-1}{2}$
      - Donner à v la valeur `Fib2`(k)
      - Donner à u la valeur $(v[1]^2 + v[2]^2, (2v[1] + v[2]) \times v[2])$
    - **sinon**
      - Donner à k la valeur $\frac{n}{2}$
      - Donner à v la valeur `Fib2`(k)
      - Donner à u la valeur $((2v[2] - v[1]) \times v[2], v[1]^2 + v[2]^2)$
    - **fin**
  - **fin**
- **fin**
- **retourner** : u

fin

- On utilise les deux formules pour F_{2k+1} selon sa position dans le vecteur
- Pour récupérer F_n, on appelle `Fib2`(n)[2], pour $n \geq 0$
- C'est diablement compliqué...

1 Les nombres de Fibonacci

2 Les tris

3 Pour aller plus loin

Algorithme 28 : Tri par sélection

Entrée : Un tableau T de n entiers
Résultat : Le tableau T trié

```
début
  variables locales : Des entiers k, i, imax, temp
  pour k de n à 2 par pas de −1 faire
    % recherche de l'indice du maximum : %
    Donner à imax la valeur 1
    pour i de 2 à k par pas de 1 faire
      si T[imax] < T[i] alors
        Donner à imax la valeur i
      fin
    fin
    % échange : %
    Donner à temp la valeur T[k]
    Donner à T[k] la valeur T[imax]
    Donner à T[imax] la valeur temp
  fin
fin
```

Tri par sélection

début
    pour k **de** n **à** 2 **par pas de** -1 **faire**
        Donner à *imax* la valeur 1
        pour i **de** 2 **à** k **par pas de** 1 **faire**
            si $T[imax] < T[i]$ **alors**
                Donner à *imax* la valeur i
            fin
        fin
        Donner à *temp* la valeur $T[k]$
        Donner à $T[k]$ la valeur $T[imax]$
        Donner à $T[imax]$ la valeur *temp*
    fin
fin

Tri par sélection

début
  pour k **de** n **à** 2 **par pas de** -1 **faire**
    Donner à *imax* la valeur 1
    pour i **de** 2 **à** k **par pas de** 1 **faire**
      si $T[imax] < T[i]$ **alors**
        Donner à *imax* la valeur i
      fin
    fin
    Donner à *temp* la valeur $T[k]$
    Donner à $T[k]$ la valeur $T[imax]$
    Donner à $T[imax]$ la valeur *temp*
  fin
fin

Pendant le tri

- Recherches de l'indice du maximum
- Échanges

Coût des échanges

« **pour** k **de** n **à** 2 **par pas de** -1 »

- $n-1$ échanges, c.-à-d. $3(n-1)$ affectations.

Tri par sélection

début
  pour k **de** n **à** 2 **par pas de** -1 **faire**
    Donner à *imax* la valeur 1
    pour i **de** 2 **à** k **par pas de** 1 **faire**
      si $T[imax] < T[i]$ **alors**
        Donner à *imax* la valeur i
      fin
    fin
    Donner à *temp* la valeur $T[k]$
    Donner à $T[k]$ la valeur $T[imax]$
    Donner à $T[imax]$ la valeur *temp*
  fin
fin

Pendant le tri

- Recherches de l'indice du maximum
- Échanges

Coût des échanges

« **pour** k **de** n **à** 2 **par pas de** -1 »

- $n-1$ échanges, c.-à-d. $3(n-1)$ affectations.

Tri par sélection

début
  pour k **de** n **à** 2 **par pas de** -1 **faire**
    Donner à *imax* la valeur 1
    pour i **de** 2 **à** k **par pas de** 1 **faire**
      si $T[imax] < T[i]$ **alors**
        Donner à *imax* la valeur i
      fin
    fin
    Donner à *temp* la valeur $T[k]$
    Donner à $T[k]$ la valeur $T[imax]$
    Donner à $T[imax]$ la valeur *temp*
  fin
fin

Maximum parmi k éléments

- une affectation
- $k-1$ passages dans la boucle sur i
- Dans la boucle sur i
  - une comparaison
  - **au plus** une affectation

Au total

- **Au plus**
  - k affectations
  - $k-1$ comparaisons

Tri par sélection

début
  pour k **de** n **à** 2 **par pas de** -1 **faire**
    Donner à *imax* la valeur 1
    pour i **de** 2 **à** k **par pas de** 1 **faire**
      si $T[imax] < T[i]$ **alors**
        Donner à *imax* la valeur i
      fin
    fin
    Donner à *temp* la valeur $T[k]$
    Donner à $T[k]$ la valeur $T[imax]$
    Donner à $T[imax]$ la valeur *temp*
  fin
fin

Recherches de maximum

« **pour** k **de** n **à** 2 **par pas de** -1 »

- Affectations : au plus

$$n + (n-1) + \ldots + 2$$

$$= \frac{n^2 + n - 2}{2}$$

- Comparaisons :

$$(n-1) + (n-2) + \ldots + 1$$

$$= \frac{n^2 - n}{2}$$

Tri par sélection

```
début
  pour k de n à 2 par pas de −1 faire
    Donner à imax la valeur 1
    pour i de 2 à k par pas de 1 faire
      si T[imax] < T[i] alors
        Donner à imax la valeur i
      fin
    fin
    Donner à temp la valeur T[k]
    Donner à T[k] la valeur T[imax]
    Donner à T[imax] la valeur temp
  fin
fin
```

Pendant le tri

- Affectations : au plus $\frac{n^2+n-2}{2} + 3(n-1)$
- Comparaisons : $\frac{n^2-n}{2}$

Hypothèses

- Chaque affectation prend un temps constant
- Chaque comparaison prend un temps constant
- (À prendre avec précautions)

Tri par sélection

début
  pour k **de** n **à** 2 **par pas de** -1 **faire**
    Donner à *imax* la valeur 1
    pour i **de** 2 **à** k **par pas de** 1 **faire**
      si $T[imax] < T[i]$ **alors**
        Donner à *imax* la valeur i
      fin
    fin
    Donner à *temp* la valeur $T[k]$
    Donner à $T[k]$ la valeur $T[imax]$
    Donner à $T[imax]$ la valeur *temp*
  fin
fin

Pendant le tri

- Affectations : au plus $\frac{n^2+n-2}{2} + 3(n-1)$
- Comparaisons : $\frac{n^2-n}{2}$

Temps de calcul

- La partie de degré 1 en n est négligeable devant la partie en n^2 quand n devient grand
- Du type $K \times n^2$
- Croissance de type quadratique par rapport à n
- On dit que l'algorithme est en $\mathcal{O}(n^2)$

Algorithme 29 : Tri par insertion

Entrée : Un tableau T de n entiers

Résultat : Le tableau T trié

début
    variables locales : Des entiers k, i, v
    pour k **de** 2 **à** n **par pas de** 1 **faire**
        Donner à v la valeur $T[k]$
        Donner à i la valeur $k-1$
        `% décalage des éléments pour l'insertion : %`
        tant que $i \geq 1$ **et** $v < T[i]$ **faire**
            Donner à $T[i+1]$ la valeur $T[i]$
            Donner à i la valeur $i-1$
        fin
        `% insertion proprement dite : %`
        Donner à $T[i+1]$ la valeur v
    fin
fin

Tri par insertion

début
  pour k **de** 2 **à** n **par pas de** 1 **faire**
    Donner à v la valeur $T[k]$
    Donner à i la valeur $k-1$
    tant que $i \geq 1$ **et** $v < T[i]$ **faire**
      Donner à $T[i+1]$ la valeur $T[i]$
      Donner à i la valeur $i-1$
    fin
    Donner à $T[i+1]$ la valeur v
  fin
fin

Tri par insertion

début
  pour k **de** 2 **à** n **par pas de** 1 **faire**
    Donner à v la valeur $T[k]$
    Donner à i la valeur $k-1$
    tant que $i \geq 1$ **et** $v < T[i]$ **faire**
      Donner à $T[i+1]$ la valeur $T[i]$
      Donner à i la valeur $i-1$
    fin
    Donner à $T[i+1]$ la valeur v
  fin
fin

Pendant le tri

- Décalages
- Insertions

Coût des insertions

« **pour** k **de** 2 **à** n **par pas de** 1 »

- $n-1$ affectations

Tri par insertion

début
  pour k **de** 2 **à** n **par pas de** 1 **faire**
    Donner à v la valeur $T[k]$
    Donner à i la valeur $k-1$
    tant que $i \geq 1$ **et** $v < T[i]$ **faire**
      Donner à $T[i+1]$ la valeur $T[i]$
      Donner à i la valeur $i-1$
    fin
    Donner à $T[i+1]$ la valeur v
  fin
fin

Pendant le tri

- Décalages
- Insertions

Coût des insertions

« **pour** k **de** 2 **à** n **par pas de** 1 »

- $n-1$ affectations

Tri par insertion

début
  pour k **de** 2 **à** n **par pas de** 1 **faire**
    Donner à v la valeur $T[k]$
    Donner à i la valeur $k-1$
    tant que $i \geq 1$ **et** $v < T[i]$ **faire**
      Donner à $T[i+1]$ la valeur $T[i]$
      Donner à i la valeur $i-1$
    fin
    Donner à $T[i+1]$ la valeur v
  fin
fin

Décalages parmi k éléments

- deux affectations
- **au plus** $k-1$ passages dans la boucle `tant que`
- Pour la boucle `tant que`
  - deux comparaisons
  - deux affectations

Au total

- **Au plus**
  - $2(k-1)+2=2k$ affectations
  - $2(k-1)+2=2k$ comparaisons

Tri par insertion

début
  pour k **de** 2 **à** n **par pas de** 1 **faire**
    Donner à v la valeur $T[k]$
    Donner à i la valeur $k-1$
    tant que $i \geq 1$ **et** $v < T[i]$ **faire**
      Donner à $T[i+1]$ la valeur $T[i]$
      Donner à i la valeur $i-1$
    fin
    Donner à $T[i+1]$ la valeur v
  fin
fin

Décalages

« **pour** k **de** 2 **à** n **par pas de** 1 »

- Affectations : **au plus**

$$2 \times (2+3+\ldots+n)$$

$$= n^2 + n - 2$$

- Comparaisons : **au plus**

$$n^2 + n - 2$$

Tri par insertion

début
  pour k **de** 2 **à** n **par pas de** 1 **faire**
    Donner à v la valeur $T[k]$
    Donner à i la valeur $k-1$
    tant que $i \geq 1$ **et** $v < T[i]$ **faire**
      Donner à $T[i+1]$ la valeur $T[i]$
      Donner à i la valeur $i-1$
    fin
    Donner à $T[i+1]$ la valeur v
  fin
fin

Pendant le tri

- Affectations : au plus $n^2+n-2+(n-1)$
- Comparaisons : au plus n^2+n-2

Hypothèses

- Chaque affectation prend un temps constant
- Chaque comparaison prend un temps constant
- (À prendre avec précautions)

Tri par insertion

début
  pour k **de** 2 **à** n **par pas de** 1 **faire**
    Donner à v la valeur $T[k]$
    Donner à i la valeur $k-1$
    tant que $i \geq 1$ **et** $v < T[i]$ **faire**
      Donner à $T[i+1]$ la valeur $T[i]$
      Donner à i la valeur $i-1$
    fin
    Donner à $T[i+1]$ la valeur v
  fin
fin

Pendant le tri

- Affectations : au plus $n^2 + 2n - 3$
- Comparaisons : au plus $n^2 + n - 2$

Temps de calcul

- La partie de degré 1 en n est négligeable devant la partie en n^2 quand n devient grand
- Du type $K \times n^2$
- Croissance de type quadratique par rapport à n
- On dit que l'algorithme est en $\mathcal{O}(n^2)$

Algorithme 30 : Tri à bulles

Entrée : Un tableau T de n entiers
Résultat : Le tableau T trié

```
début
    variables locales : Des entiers k, i, temp
    % pour chaque passe : %
    pour k de n à 2 par pas de −1 faire
        % on fait remonter le plus grand : %
        pour i de 2 à k par pas de 1 faire
            si T[i] < T[i − 1] alors
                % échange de T[i] et de T[i − 1] : %
                Donner à temp la valeur T[i]
                Donner à T[i] la valeur T[i − 1]
                Donner à T[i − 1] la valeur temp
            fin
        fin
    fin
fin
```

Tri à bulles

début
  pour k **de** n **à** 2 **par pas de** -1 **faire**
    pour i **de** 2 **à** k **par pas de** 1 **faire**
      si $T[i] < T[i-1]$ **alors**
        Donner à *temp* la valeur $T[i]$
        Donner à $T[i]$ la valeur $T[i-1]$
        Donner à $T[i-1]$ la valeur *temp*
      fin
    fin
  fin
fin

Tri à bulles

```
début
  pour k de n à 2 par pas de −1 faire
    pour i de 2 à k par pas de 1 faire
      si T[i] < T[i − 1] alors
        Donner à temp la valeur T[i]
        Donner à T[i] la valeur T[i − 1]
        Donner à T[i − 1] la valeur temp
      fin
    fin
  fin
fin
```

Pendant le tri

- Échanges

Coût d'un échange

- trois affectations

Tri à bulles

début
  pour k **de** n **à** 2 **par pas de** -1 **faire**
    pour i **de** 2 **à** k **par pas de** 1 **faire**
      si $T[i] < T[i-1]$ **alors**
        Donner à *temp* la valeur $T[i]$
        Donner à $T[i]$ la valeur $T[i-1]$
        Donner à $T[i-1]$ la valeur *temp*
      fin
    fin
  fin
fin

Pendant le tri

- Échanges

Coût d'un échange

- trois affectations

Tri à bulles

début
  pour k **de** n **à** 2 **par pas de** -1 **faire**
    pour i **de** 2 **à** k **par pas de** 1 **faire**
      si $T[i] < T[i-1]$ **alors**
        Donner à *temp* la valeur $T[i]$
        Donner à $T[i]$ la valeur $T[i-1]$
        Donner à $T[i-1]$ la valeur *temp*
      fin
    fin
  fin
fin

Boucles sur i

- $k-1$ passages
  - une comparaison
  - **au plus** un échange

Boucles sur i

- Au total
  - $k-1$ comparaisons
  - **au plus** $k-1$ échanges

Tri à bulles

début
  pour k **de** n **à** 2 **par pas de** -1 **faire**
    pour i **de** 2 **à** k **par pas de** 1 **faire**
      si $T[i] < T[i-1]$ **alors**
        Donner à *temp* la valeur $T[i]$
        Donner à $T[i]$ la valeur $T[i-1]$
        Donner à $T[i-1]$ la valeur *temp*
      fin
    fin
  fin
fin

Boucles sur k

« **pour** k **de** n **à** 2 **par pas de** -1 »

- Boucle sur i
  - $k-1$ comparaisons
  - **au plus** $k-1$ échanges

Boucles sur k

- Au total
  - $(n-1)+(n-2)+\ldots+1$ comparaisons
  - **au plus** $(n-1)+(n-2)+\ldots+1$ échanges

Tri à bulles

début
  pour k **de** n **à** 2 **par pas de** -1 **faire**
    pour i **de** 2 **à** k **par pas de** 1 **faire**
      si $T[i] < T[i-1]$ **alors**
        Donner à *temp* la valeur $T[i]$
        Donner à $T[i]$ la valeur $T[i-1]$
        Donner à $T[i-1]$ la valeur *temp*
      fin
    fin
  fin
fin

Pendant le tri

- Affectations : au plus $\frac{3n(n-1)}{2}$
- Comparaisons : $\frac{n(n-1)}{2}$

Hypothèses

- Chaque affectation prend un temps constant
- Chaque comparaison prend un temps constant
- (À prendre avec précautions)

Tri à bulles

début
  pour k **de** n **à** 2 **par pas de** -1 **faire**
    pour i **de** 2 **à** k **par pas de** 1 **faire**
      si $T[i] < T[i-1]$ **alors**
        Donner à *temp* la valeur $T[i]$
        Donner à $T[i]$ la valeur $T[i-1]$
        Donner à $T[i-1]$ la valeur *temp*
      fin
    fin
  fin
fin

Pendant le tri

- Affectations : au plus $\frac{3n(n-1)}{2}$
- Comparaisons : $\frac{n(n-1)}{2}$

Temps de calcul

- La partie de degré 1 en n est négligeable devant la partie en n^2 quand n devient grand
- Du type $K \times n^2$
- Croissance de type quadratique par rapport à n
- On dit que l'algorithme est en $\mathcal{O}(n^2)$

Tri fusion

Fonction `TriFusion` (*T,n,debut,fin*)

Entrée : Un tableau *T* de *n* entiers et des entiers *debut* et *fin*
Résultat : Le tableau *T* trié entre *debut* et *fin*

début
  variables locales : Un entier *milieu*, un tableau *temp* de *n* entiers
  si *debut* $<$ *fin* **alors**
    Donner à *milieu* la valeur $\lfloor \frac{debut+fin}{2} \rfloor$
    `TriFusion` (*T,n,debut,milieu*)
    `TriFusion` (*T,n,milieu+1,fin*)
    `Interclassement` (*T,n,debut,milieu,fin*)
  fin
fin

Fonction `Interclassement` ($T, n, debut, milieu, fin$)

Entrée : Un tableau T de n entiers, des entiers *debut*, *milieu* et *fin*
Résultat : Le tableau T interclassé entre *debut* et *fin*

```
début
    variables locales : Des entiers i, j, k, un tableau temp de n entiers
    Donner à i la valeur debut
    Donner à j la valeur milieu + 1
    pour k de debut à fin par pas de 1 faire
        si (j > fin ou (i ≤ milieu et T[i] < T[j])) alors
            Donner à temp[k] la valeur T[i]
            Donner à i la valeur i + 1
        sinon
            Donner à temp[k] la valeur T[j]
            Donner à j la valeur j + 1
        fin
    fin
    % copier le tableau résultat temp à sa place dans le tableau T %
    pour k de debut à fin par pas de 1 faire
        Donner à T[k] la valeur temp[k]
    fin
fin
```

Tri fusion

Remarque

- Contrairement aux algorithmes précédents, le tri fusion n'est pas un tri « en place », puisque l'interclassement utilise un tableau auxiliaire de même taille que le tableau initial.
- Avec une implémentation astucieuse, on peut améliorer la gestion de la mémoire pour l'interclassement, mais l'algorithme est alors ralenti.
- Le tri par tas est un algorithme de tri en place sophistiqué de complexité $\mathcal{O}(n \log n)$

Tri fusion

Fonction `TriFusion` (*T,n,debut,fin*)

début

  si $debut < fin$ **alors**

    Donner à *milieu* la valeur $\lfloor \frac{debut+fin}{2} \rfloor$

    `TriFusion` (*T,n,debut,milieu*)

    `TriFusion` (*T,n,milieu+1,fin*)

    `Interclassement` (*T,n,debut,milieu,fin*)

  fin

fin

Tri fusion

Fonction `TriFusion` (*T,n,debut,fin*)

début
- **si** $debut < fin$ **alors**
  - Donner à *milieu* la valeur $\lfloor \frac{debut+fin}{2} \rfloor$
  - `TriFusion` (*T,n,debut,milieu*)
  - `TriFusion` (*T,n,milieu+1,fin*)
  - `Interclassement` (*T,n,debut,milieu,fin*)
- **fin**

fin

Pendant le tri

- Comparaison
- Calcul du milieu
- Appels récursifs à la fonction `TriFusion`
- `Interclassement`

Opérations élémentaire

- comparaison
- addition
- division
- affectation
- appel récursif

$\longrightarrow$ hypothèse du coût unitaire

Tri fusion

Fonction `TriFusion` (*T,n,debut,fin*)

début

  si *debut* $<$ *fin* **alors**

    Donner à *milieu* la valeur $\lfloor \frac{debut+fin}{2} \rfloor$

    `TriFusion` (*T,n,debut,milieu*)

    `TriFusion` (*T,n,milieu+1,fin*)

    `Interclassement` (*T,n,debut,milieu,fin*)

  fin

fin

Pendant le tri

- Comparaison
- Calcul du milieu
- Appels récursifs à la fonction `TriFusion`
- `Interclassement`

Opérations élémentaire

- comparaison
- addition
- division
- affectation
- appel récursif

$\longrightarrow$ hypothèse du coût unitaire

Tri fusion

Fonction `Interclassement`(*T,n,debut,milieu,fin*)

début
- Donner à i la valeur *debut*
- Donner à j la valeur $milieu + 1$
- **pour** k **de** *debut* **à** *fin* **par pas de** 1 **faire**
  - **si** $(j > fin$ **ou** $(i \leq milieu$ **et** $T[i] < T[j]))$
  - **alors**
    - Donner à $temp[k]$ la valeur $T[i]$
    - Donner à i la valeur $i + 1$
  - **sinon**
    - Donner à $temp[k]$ la valeur $T[j]$
    - Donner à j la valeur $j + 1$
  - **fin**
- **fin**
- **pour** k **de** *debut* **à** *fin* **par pas de** 1 **faire**
  - Donner à $T[k]$ la valeur $temp[k]$
- **fin**

fin

Interclassement

Deux tableaux de m nombres déjà triés

- deux affectations
- une addition
- première boucle sur k
  - m passages
  - trois tests
  - deux affectations
  - une addition
- deuxième boucle sur k
  - m passages
  - une affectation

Tri fusion

Fonction `Interclassement` ($T,n,debut,milieu,fin$)

début
  Donner à i la valeur $debut$
  Donner à j la valeur $milieu + 1$
  pour k **de** $debut$ **à** fin **par pas de** 1 **faire**
    si ($j > fin$ **ou** ($i \leq milieu$ **et** $T[i] < T[j]$))
    alors
      Donner à $temp[k]$ la valeur $T[i]$
      Donner à i la valeur $i + 1$
    sinon
      Donner à $temp[k]$ la valeur $T[j]$
      Donner à j la valeur $j + 1$
    fin
  fin
  pour k **de** $debut$ **à** fin **par pas de** 1 **faire**
    Donner à $T[k]$ la valeur $temp[k]$
  fin
fin

Interclassement

Deux tableaux de m nombres déjà triés

- $3m$ tests
- $3m + 2$ affectations
- $m + 1$ additions

Interclassement

Deux tableaux de m nombres déjà triés

- Au total
  - $7m + 3$ opérations élémentaires

Tri fusion

Fonction `TriFusion` (*T,n,debut,fin*)

début
  si $debut < fin$ **alors**
    Donner à *milieu* la valeur $\lfloor \frac{debut+fin}{2} \rfloor$
    `TriFusion` (*T,n,debut,milieu*)
    `TriFusion` (*T,n,milieu+1,fin*)
    `Interclassement` (*T,n,debut,milieu,fin*)
  fin
fin

Pour simplifier

- $n = 2^k$

Tri fusion

Fonction `TriFusion` (*T,n,debut,fin*)

début
  si $debut < fin$ **alors**
    Donner à *milieu* la valeur $\lfloor \frac{debut+fin}{2} \rfloor$
    `TriFusion` (*T,n,debut,milieu*)
    `TriFusion` (*T,n,milieu+1,fin*)
    `Interclassement` (*T,n,debut,milieu,fin*)
  fin
fin

Coût du tri

- Trier un tableau de 2^k nombres = interclasser deux tableaux de 2^{k-1} nombres déjà triés + trier deux tableaux de 2^{k-1} nombres
- Trier un tableau de 2^{k-1} nombres = interclasser deux tableaux de 2^{k-2} nombres déjà triés + trier deux tableaux de 2^{k-2} nombres
- Etc.
- Trier un tableau de 2^0 nombres : une comparaison

Tri fusion

Fonction `TriFusion` (*T,n,debut,fin*)

début

  si *debut* $<$ *fin* **alors**

    Donner à *milieu* la valeur $\lfloor \frac{debut+fin}{2} \rfloor$

    `TriFusion` (*T,n,debut,milieu*)

    `TriFusion` (*T,n,milieu+1,fin*)

    `Interclassement` (*T,n,debut,milieu,fin*)

  fin

fin

Coût du tri

- $T_i =$ Coût du tri d'un tableau de 2^i nombres
- $T_0 = 1$
- $T_{i+1} = 7 \times 2^i + 3 + 2 \times T_i + 3$

Dérécursivisation

- $T_k = (7k+1)2^{k-1} + 6k$
- Résultat à prendre avec précautions

Tri fusion

Fonction `TriFusion` (*T,n,debut,fin*)

début

  si $debut < fin$ **alors**

    Donner à *milieu* la valeur $\lfloor \frac{debut+fin}{2} \rfloor$

    `TriFusion` (*T,n,debut,milieu*)

    `TriFusion` (*T,n,milieu+1,fin*)

    `Interclassement` (*T,n,debut,milieu,fin*)

  fin

fin

Coût du tri

- T_i = Coût du tri d'un tableau de 2^i nombres
- $T_0 = 1$
- $T_{i+1} = 7 \times 2^i + 3 + 2 \times T_i + 3$

Dérécursivisation

- $T_k = (7k + 1)2^{k-1} + 6k$
- Résultat à prendre avec précautions

Tri fusion

Fonction `TriFusion` (*T,n,debut,fin*)

début

  si $debut < fin$ **alors**

    Donner à *milieu* la valeur $\lfloor \frac{debut+fin}{2} \rfloor$

    `TriFusion` (*T,n,debut,milieu*)

    `TriFusion` (*T,n,milieu+1,fin*)

    `Interclassement` (*T,n,debut,milieu,fin*)

  fin

fin

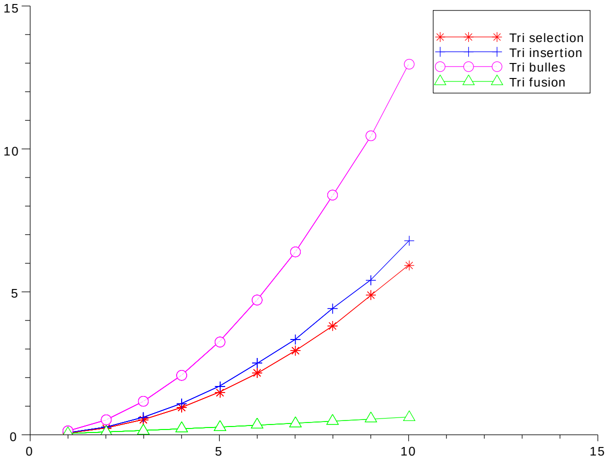
Coût du tri

- $T_k = (7k+1)2^{k-1} + 6k$ opérations élémentaires

Temps de calcul

- Le terme en k est négligeable devant le terme en $k \times 2^k$ quand k devient grand
- On revient à $n = 2^k$
- Du type $K \times n \log n$
- On dit que l'algorithme est en $\mathcal{O}(n \log n)$

Vérification expérimentale



1 Les nombres de Fibonacci

2 Les tris

3 Pour aller plus loin

Des tris encore plus rapides ?

On a vu

- Des algorithmes de tri dont la complexité dans le pire des cas est en $\mathcal{O}(n^2)$ (tri par insertion, tri par sélection, tri à bulles)
- Le tri fusion, dont la complexité dans le pire des cas est en $\mathcal{O}(n \log n)$

Peut-on faire mieux ?

Existe-t-il des algorithmes de tri asymptotiquement plus rapides que le tri fusion de plus d'un facteur constant ?

Une précision importante

Ces $\mathcal{O}$ sont en fait des Θ

Les pires des cas se produisent effectivement :

- Pour le tri par insertion, et le tri à bulles, le pire des cas se produit quand le tableau est déjà trié dans l'ordre inverse.
- Pour le tri par sélection, toutes les exécutions sont à peu près les mêmes
- Pour le tri fusion, si la longueur du tableau est une puissance de deux, c'est un peu plus rapide

Remarque

$f(n) \underset{n\to+\infty}{=} \Theta(g(n))$ ssi $\exists N, c, C\ \forall n > N\ |c \cdot g(n)| < |f(n)| < |C \cdot g(n)|$

Tris par comparaisons

Définition

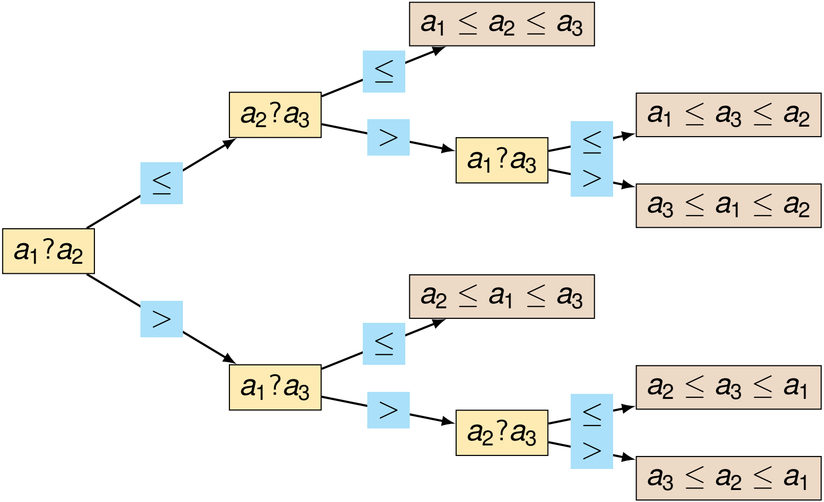
- Entrée $a_1, a_2, \ldots, a_n$
- Les informations sur les objets à trier sont obtenues uniquement par des comparaisons $a_i \overset{?}{\leq} a_j$
- (On suppose $a_1, a_2, \ldots, a_n$ tous distincts)

Tous les algorithmes de tri qu'on a vus sont des tris par comparaisons

Modèle abstrait

- Arbre de décision
- Représenter toutes les comparaisons effectuées par l'algorithme sur des entrées d'une taille donnée

Tri par insertion de 3 éléments a_1, a_2, a_3



Arbre de décision pour un tri sur n éléments

Structure

- Nœud interne : comparaison $a_i?a_j$
- Arbre binaire : deux réponses possibles pour chaque comparaison
- Feuille : résultat du tri (étiquetée par une permutation de $1, \dots, n$)

Exécution de l'algorithme

- Chemin de la racine à une feuille
- Nombre de comparaisons = nombre de nœuds internes

Algorithme correct

Toutes les $n!$ permutations de $1, \dots, n$ sont des feuilles

Nombre de comparaisons

Pire des cas

- Plus long chemin de la racine à une feuille
- Hauteur de l'arbre de décision

Théorème

Un arbre de décision permettant de trier n éléments est de hauteur $\Omega(n \log n)$

Remarque (Notation Ω)

$f(n) \underset{n\to+\infty}{=} \Omega(g(n))$ ssi $\exists N, c$ t.q. $\forall n > N\ c\,|g(n)| \leq |f(n)|$

Preuve du théorème

Théorème

Un arbre de décision permettant de trier n éléments est de hauteur $\Omega(n \log n)$

- hauteur de l'arbre h
- un arbre binaire de hauteur h possède au plus 2^h feuilles
- cet arbre de décision permet de trier n éléments
- toutes les $n!$ permutations apparaissent comme feuilles de l'arbre
- donc $n! \leq 2^h$ ou encore $h \geq \log_2(n!)$
- formule de Stirling $n! > \left(\frac{n}{e}\right)^n$
- finalement $h \geq \log_2\left(\left(\frac{n}{e}\right)^n\right) = n\log_2(n) - n\log_2(e) = \Omega(n \log n)$

La réponse

Théorème

Un arbre de décision permettant de trier n éléments est de hauteur $\Omega(n \log n)$

Corollaire

- Aucun tri par comparaisons ne possède une complexité meilleure que $\mathcal{O}(n \log n)$
- Le tri fusion a une complexité optimale

Remarque

Ce résultat ne concerne que les tris par comparaisons, puisqu'il est basé sur le modèle des arbres de décision.

D'autres tris

Deux exemples qui ne sont pas des tris par comparaisons

- Tri par dénombrement
  - permet de trier des nombres bornés
- Tri par base
  - permet de trier des nombres de longueur bornée

Avant-goût

On va voir que ces tris ont une complexité linéaire

Remarque

Mais ce sont des tris spécialisés, qui ne permettent pas de trier n'importe quelles données

Tri par dénombrement

Hypothèse

Les entiers à trier appartiennent à $\{0, \ldots, k-1\}$

Principe

- Créer un tableau `Compte` de longueur k rempli de 0
- Parcourir le tableau `T` à trier
- Pour tout i, incrémenter `Compte[T[i]]`
- Ensuite, parcourir de nouveau le tableau `T` en le remplissant de `Compte[0]` fois le nombre 0, suivis de `Compte[1]` fois le nombre 1, etc.

Tri par dénombrement

Complexité

Pourvu que k ne soit pas trop grand (c.-à-d. $k = \mathcal{O}(n)$), le tri par dénombrement est en $\mathcal{O}(n)$ car essentiellement on parcourt deux fois le tableau à trier.

Remarque

Une contradiction ?
Non, car ce n'est pas un tri par comparaisons !
Cet algorithme n'effectue d'ailleurs **aucune** comparaison...

Tri par base

Hypothèse

Les entiers à trier ont une longueur fixe

Exemple

329		720		720		329
457		355		329		355
657		436		436		436
839	$\Rightarrow$	457	$\Rightarrow$	839	$\Rightarrow$	457
436		657		355		657
720		329		457		720
355		839		657		839
		$\uparrow$		$\uparrow$		$\uparrow$

Tri par base

Principe

- On trie d'abord selon le chiffre des unités
- Puis selon le chiffre suivant, etc.
- On termine par le chiffre de plus grand poids
- Il est essentiel d'utiliser pour les étapes intermédiaires un tri stable

Remarque

Un tri stable est un tri qui conserve l'ordre relatif des indices des valeurs égales

Tri par base

Complexité

- Tout dépend de celle du tri stable utilisé...
- Il semble raisonnable d'utiliser un tri par dénombrement, puisque les chiffres dans une base donnée sont peu nombreux.
- La complexité du tri par base est alors en $\mathcal{O}(n)$, puisqu'on applique un nombre fixe de fois un algorithme linéaire.

FIN