Analyse de programmes récursifs

Option Informatique

Anthony Lick
Lycée Janson de Sailly

Introduction

Introduction

```
1  let rec fact n = match n with
2    | 0 -> 1
3    | _ -> n * fact (n-1)
4  ;;
```

Exemple

On rappelle ci-dessus le code OCaml de la fonction factorielle.

Cette fonction est très proche de la définition mathématique associée et il est facile de voir qu'elle termine pour tout entier positif passé en argument.

Introduction

```
1  let rec fact n = match n with
2    | 0 -> 1
3    | _ -> n * fact (n-1)
4  ;;
```

Exemple

En fait, elle **termine** car il n'existe pas de suite strictement décroissante dans $\mathbb{N}$, et elle est **correcte** par principe de récurrence : le cas terminal est correct car $0! = 1$, et ensuite une récurrence immédiate sur $\mathbb{N}$ prouve que la fonction calcule bien $n!$.

En ce qui concerne sa **complexité**, elle vérifie une relation de récurrence de la forme $C(n) = C(n-1)+O(1)$, dont la solution est $C(n) = O(n)$.

Principes de l'analyse des fonctions récursives

Terminaison

La **terminaison** se montre en exhibant une quantité, fonction des paramètres de la fonction récursive, et dont les valeurs décroissent à chaque appel récursif. Ces valeurs sont dans un ensemble muni d'un **ordre bien fondé**, pas nécessairement $\mathbb{N}$.

Correction

La **correction**, en général assez immédiate, repose sur un principe proche du principe de **récurrence**.

Complexité

La **complexité** s'étudie en évaluant la formule récurrente donnée par le coût du traitement dans le corps de la fonction auquel s'ajoute le coût des **appels récursifs**.

Terminaison

Relations d'ordre

Définition

Une **relation d'ordre** $\preceq$ sur un ensemble E est une relation :

- **réflexive** : $\forall x \in E,\ w \preceq x$.
- **transitive** : $\forall (x, y, z) \in E^3$, si $x \preceq y$ et $y \preceq z$, alors $x \preceq z$.
- **antisymétrique** : $\forall (x, y) \in E^2$, si $x \preceq y$ et $y \preceq x$, alors $x = y$.

De plus, si $\forall (x, y) \in E^2$, on a soit $x \preceq y$ soit $y \preceq x$, alors la relation est dite **totale**.

Relations d'ordre

Définition

Une **élément minimal** de E est un élément x tel que $\forall y \in E,\ y \preceq x \Rightarrow x = y$.

Un **plus petit élément** de E (nécessairement unique) est un élément $x \in E$ tel que $\forall y \in E,\ x \preceq y$.

Définition

Un ensemble ordonné $(E, \preceq)$ est dit **bien fondé** (on dit aussi que c'est l'ordre qui est bien fondé) s'il n'existe pas de suite de cet ensemble strictement décroissante (pour $\preceq$).

Si de plus l'ordre est total, E est dit **bien ordonné**.

Exemples

Exemple

L'ensemble $(\mathbb{N}, \leq)$ est bien ordonné.

L'ensemble $(\mathbb{Q}_+, \leq)$ ne l'est pas, car la suite $(\frac{1}{n})_{n \in \mathbb{N}^*}$ est strictement décroissante.

Ordres bien fondés sur $\mathbb{N}^2$

Exemple

L'**ordre produit** sur $\mathbb{N}^2$, défini par :

$$(a, b) \preceq_\times (c, d) \iff a \leq c \text{ et } b \leq d$$

est bien fondé.

Attention, ce n'est pas un ordre total.

Ordres bien fondés sur $\mathbb{N}^2$

Exemple

L'**ordre lexicographique** sur $\mathbb{N}^2$, défini par :

$$(a,b) \preceq_{\text{lex}} (c,d) \iff a < c \quad \text{ou} \quad (a = c \text{ et } b \leq d)$$

est bien fondé.

De plus, c'est un ordre total.

Preuve

En effet, partant d'un couple (a,b), il n'existe que b couples de la forme (a,x) avec $x < b$, donc une suite strictement décroissante de longueur au moins $b+2$ atteint un couple (c,d) avec $c < a$. On conclut car $\mathbb{N}$ est lui-même bien fondé.

Ensembles bien ordonnés

Proposition

Un ensemble ordonné E est bien ordonné si et seulement si toute partie non vide admet un plus petit élément.

Preuve

$\Rightarrow$ Soit $A \subset E$ non vide, avec E bien ordonné. Montrons que A admet un plus petit élément. Soit $x_0 \in A$. Si x_0 n'est pas le plus petit élément, il existe $x_1 \in A$ tel que $x_1 \prec x_0$. En réitérant le processus, on construit une suite strictement décroissante : $x_k \prec x_{k-1} \prec x_{k-2} \prec \cdots \prec x_0$. Le processus s'arrête car il n'existe pas de suite infinie strictement décroissante dans un ensemble bien ordonné : on a donc trouvé un plus petit élément.

Ensembles bien ordonnés

Proposition

Un ensemble ordonné E est bien ordonné si et seulement si toute partie non vide admet un plus petit élément.

Preuve

$\Leftarrow$ Réciproquement, l'ordre E est nécessairement total (sinon, il existerait une partie à 2 éléments sans plus petit élément). De plus, si $(x_n)_{n\in\mathbb{N}}$ est une suite d'éléments de E, alors $\{x_n \mid n \in \mathbb{N}\}$ possède un plus petit élément, donc la suite n'est pas strictement décroissante.

Finalement, l'ensemble est bien ordonné.

Induction

Induction

On va maintenant voir le principe d'induction, qui est à un ensemble bien fondé ce que la récurrence (forte) est à $\mathbb{N}$.

Définition

On appelle **prédicat** sur un ensemble E une application de E dans l'ensemble des booléens.

Théorème

Soit $(E, \preceq)$ un ensemble bien fondé. Notons $\mathcal{M}$ l'ensemble de ses éléments minimaux. Si le prédicat $\mathcal{P}$ vérifie :

- $\forall x \in \mathcal{M},\ \mathcal{P}(x)$,
- $\forall x \in E \setminus \mathcal{M},\ (\forall y \prec x,\ \mathcal{P}(y)) \Rightarrow \mathcal{P}(x)$.

Alors, $\forall x \in E, \mathcal{P}(x)$.

Preuve

Par l'absurde, supposons que les deux propriétés ci-dessus soient vraies et qu'il existe $x_0 \in E$ tel que $\mathcal{P}(x_0)$ soit faux.

Par la première propriété, on a $x_0 \notin \mathcal{M}$.

Alors il existe $x_1 \in E$ tel que $x_1 \prec x_0$ et tel que $\mathcal{P}(x_1)$ est faux (contraposée de la seconde propriété).

De plus, $x_1 \notin \mathcal{M}$ par la première propriété.

On recommence avec x_1, et on construit ainsi une suite infinie strictement décroissante, ce qui est impossible car E est bien fondé.

Terminaison d'une fonction récursive

Théorème

Soit f une fonction OCaml récursive.
Soit φ une application de l'ensemble des arguments $\mathcal{A}$ de f vers un ensemble bien fondé E. Supposons que :

- la fonction f termine pour tous les arguments de l'ensemble $\mathcal{M}_{\mathcal{A}} = \{x \in \mathcal{A} \mid \varphi(x) \in \mathcal{M}\}$ où $\mathcal{M}$ est l'ensemble des minimaux de E ;
- $\forall x \in \mathcal{A} \setminus \mathcal{M}_{\mathcal{A}}$, le calcul de $f(x)$ ne requiert qu'un nombre fini (éventuellement aucun) d'appels récursifs à f, sur des arguments y vérifiant $\varphi(y) \prec \varphi(x)$, et la terminaison de ces appels entraine celle de $f(x)$.

Alors, la fonction f termine sur tout argument de $\mathcal{A}$.

Terminaison d'une fonction récursive

Preuve

Il suffit de démontrer par induction la propriété sur E suivante :

$$\mathcal{P}(z) : \text{ les appels } f(x) \text{ avec } \varphi(x) = z \text{ terminent.}$$

Terminaison d'une fonction récursive

Définition

Les éléments $x \in \mathcal{A}$ pour lesquels $f(x)$ ne nécessite aucun appel récursif à f sont appelés les **cas de base** ou **cas terminaux** (les éléments de $\mathcal{M}_{\mathcal{A}}$ sont terminaux, mais ce ne sont pas forcément les seuls).

Remarque

On aura souvent des arguments dans un ensemble bien fondé, et la fonction φ sera l'identité.

Pour des structures plus complexes, l'application φ est souvent un indicateur à valeurs dans $\mathbb{N}$ comme la taille d'une liste ou d'un tableau.

Terminaison d'une fonction récursive

```
1  let rec fact n = match n with
2    | 0 -> 1
3    | _ -> n * fact (n-1)
4  ;;
```

Exemple

La fonction `fact` termine.

```
1  let rec longueur l= match l with
2    | [] -> 0
3    | _::q -> 1 + longueur q
4  ;;
```

Exemple

La fonction ci-dessus donnant la longueur d'une liste termine. En effet, on prend comme application φ celle qui à une liste associe sa longueur.

Terminaison d'une fonction récursive

```
1  let rec binome n p = match (n,p) with
2    | 0,_ -> 0
3    | _,0 -> 1
4    | _ -> (n * binome (n-1) (p-1))/p
5  ;;
```

Exemple

Le calcul des coefficients binomiaux par la fonction ci-dessus termine.

En effet, on peut prendre $E = \mathbb{N}^2$ muni de l'ordre produit ou de l'ordre lexicographique (les deux conviennent).

On peut aussi prendre $E = \mathbb{N}$ et $\varphi : \left\{ \begin{array}{ccc} \mathbb{N}^2 & \to & \mathbb{N} \\ (x,y) & \mapsto & x+y \end{array} \right.$

Terminaison d'une fonction récursive

```
1  let rec ack n p = match (n,p) with
2    | 0,_ -> p+1
3    | _,0 -> ack (n-1) 1
4    | _ -> ack (n-1) (ack n (p-1))
5  ;;
```

Exemple

La fonction d'Ackermann ci-dessus termine.

En effet, on prend ici $\mathbb{N}^2$ muni de l'ordre lexicographique.

- Les cas de base sont les couples dont la forme $(0, x)$.
- Sur $(n, 0)$, il n'y a qu'un seul appel récursif sur $(n-1, 1) \prec_{\text{lex}} (n, 0)$.
- Sur (n, p) avec $n \neq 0$ et $p \neq 0$, il y a deux appels récursifs sur $(n, p-1)$ et $(n-1, \texttt{ack}(n, p-1))$, qui sont bien plus petits que (n, p) pour l'ordre lexicographique.

Terminaison d'une fonction récursive

```
1  let rec ack n p = match (n,p) with
2    | 0,_ -> p+1
3    | _,0 -> ack (n-1) 1
4    | _ -> ack (n-1) (ack n (p-1))
5  ;;
```

Remarque

La fonction d'Ackermann croît très vite (plus vite que n'importe quelle fonction usuelle mathématique).

On pourra essayer de calculer `ack i j` pour $i, j \leq 3$, et vérifier que `ack 4 0` vaut 13, `ack 4 1` vaut 65533, et `ack 4 2` vaut $2^{65536} - 3$.

La terminaison n'implique pas que le calcul effectif de la fonction soit possible.

Terminaison d'une fonction récursive

```
1  let rec morris n p = match (n,p) with
2    | 0,_ -> 1
3    | _ -> morris n (p-1) + morris (n-1) (morris n p)
4  ;;
```

Exemple

La fonction de Morris ne termine pas.

En effet, `morris n p` fait appel à `morris n p`.

Terminaison d'une fonction récursive

```
1  let rec syracuse n = match n with
2    | 0 | 1 -> 1
3    | n when n mod 2 = 0 -> syracuse (n/2)
4    | _ -> syracuse (3*n+1)
5  ;;
```

Exemple

Pour conclure, précisons qu'il n'est pas toujours facile de trouver un "bon ensemble" bien fondé et une application φ associée.

La terminaison de la fonction de Syracuse est un problème ouvert.

Correction

Correction

Correction

Montrer la **correction** d'une fonction signifie montrer qu'elle calcule ce qu'elle est censée calculer.

Tout le préambule mathématique introduit permet de répondre facilement à cette question : on va montrer la correction par **récurrence**, ou plus généralement par **induction**, comme pour la terminaison.

Correction

Théorème

On reprend l'ensemble E bien fondé et φ l'application du théorème de terminaison.

Considérons sur E le prédicat suivant :

$$\mathcal{P}(z) : \text{ les } f(x) \text{ pour } \varphi(x) = z \text{ ont la bonne valeur.}$$

Supposons que :

- $\forall x \in \mathcal{M}_{\mathcal{A}},\ \mathcal{P}(\varphi(x))$.
- $\forall x \in \mathcal{A} \setminus \mathcal{M}_{\mathcal{A}}$, le calcul de $f(x)$ ne requiert qu'un nombre fini d'appels récursifs d'arguments $(y_i)_{i \leq N}$ tels que $\varphi(y_i) \prec \varphi(x)$ et $(\forall i \leq N,\ \mathcal{P}(\varphi(y_i))) \Rightarrow \mathcal{P}(\varphi(x))$.

Alors, $\forall x \in \mathcal{A},\ \mathcal{P}(\varphi(x))$.

Correction

Preuve

Immédiate.

Remarque

En général, prouver la **correction** d'une fonction récursive sera relativement immédiat.

Exemple

Il est facile de voir que la fonction `syracuse` renvoie 1 si jamais elle termine.

Correction

```
let mini l = (* renvoie le minimum de l et le reste de la liste *)
  let rec aux m reste l = match l with
    | [] -> m, reste
    | x::q when m<=x -> aux m (x::reste) q
    | x::q -> aux x (m::reste) q
    in aux (List.hd l) [] (List.tl l)
;;

let rec tri_selection l = match l with
  | [] -> []
  | _ -> let m,q=mini l in m::(tri_selection q)
;;
```

Exemple

Étudions maintenant le tri par sélection sur les listes.

La fonction `mini` prend une liste non vide en entrée, et renvoie un couple constitué de son plus petit élément et de la liste de ses autres éléments, dans un ordre arbitraire.

Correction

```
1  let mini l = (* renvoie le minimum de l et le reste de la liste *)
2    let rec aux m reste l = match l with
3      | [] -> m, reste
4      | x::q when m<=x -> aux m (x::reste) q
5      | x::q -> aux x (m::reste) q
6      in aux (List.hd l) [] (List.tl l)
7  ;;
8
9  let rec tri_selection l = match l with
10   | [] -> []
11   | _ -> let m,q=mini l in m::(tri_selection q)
12 ;;
```

Exemple

La terminaison de aux se montre en considérant φ la taille de la liste `l`.

Correction

```
1   let mini l = (* renvoie le minimum de l et le reste de la liste *)
2     let rec aux m reste l = match l with
3       | [] -> m, reste
4       | x::q when m<=x -> aux m (x::reste) q
5       | x::q -> aux x (m::reste) q
6       in aux (List.hd l) [] (List.tl l)
7   ;;
8
9   let rec tri_selection l = match l with
10    | [] -> []
11    | _ -> let m,q=mini l in m::(tri_selection q)
12  ;;
```

Exemple

La correction de aux se fait en considérant la propriété suivante sur $\mathbb{N}$:

$\mathcal{P}(z)$: si `l` est une liste de taille z, et si `m` est plus petit que les éléments de `reste`, alors `aux m reste l` renvoie le couple constitué du minimum parmi les éléments de `m::l@reste`, et d'une liste formée des mêmes éléments, moins ce minimum.

Correction

```
1  let mini l = (* renvoie le minimum de l et le reste de la liste *)
2    let rec aux m reste l = match l with
3      | [] -> m, reste
4      | x::q when m<=x -> aux m (x::reste) q
5      | x::q -> aux x (m::reste) q
6      in aux (List.hd l) [] (List.tl l)
7  ;;
8
9  let rec tri_selection l = match l with
10   | [] -> []
11   | _ -> let m,q=mini l in m::(tri_selection q)
12 ;;
```

Exemple

L'appel de `mini` à `aux` montre qu'elle est correcte.

Correction

```
1   let mini l = (* renvoie le minimum de l et le reste de la liste *)
2     let rec aux m reste l = match l with
3       | [] -> m, reste
4       | x::q when m<=x -> aux m (x::reste) q
5       | x::q -> aux x (m::reste) q
6       in aux (List.hd l) [] (List.tl l)
7   ;;
8
9   let rec tri_selection l = match l with
10    | [] -> []
11    | _ -> let m,q=mini l in m::(tri_selection q)
12  ;;
```

Exemple

Pour l'algorithme de tri en lui-même, on prend φ la taille de la liste `l` et $\mathcal{P}(z)$: "si `l` est de taille z, l'algorithme renvoie la version triée de `l`".

Complexité

Un exemple fondamental : le tri fusion

Le tri fusion

Nous avons vu plusieurs algorithmes de tris fonctionnant en $O(n^2)$ dans le pire des cas.

La question naturelle est alors : peut-on faire mieux ?

Nous voyons maintenant un algorithme de tri, appelé **tri fusion**, plus efficace dans le pire des cas.

Un exemple fondamental : le tri fusion

Diviser pour régner

L'algorithme du tri fusion utilise une méthode classique lorsqu'on programme par récurrence : la méthode **diviser pour régner**.

Cela consiste à découper l'instance de notre problème en sous-problèmes plus simples, à résoudre (récursivement) les sous-problèmes, puis à recombiner ces solutions pour obtenir la solution du problème initial.

Le prochain chapitre sera dédié à l'étude plus générale de ce genre d'algorithmes.

Un exemple fondamental : le tri fusion

Principe

Le principe du **tri fusion** pour trier une liste l est le suivant :

- si la liste est de taille 0 ou 1, elle est triée ;
- sinon, il suffit de :
  - couper la liste en deux parties de même taille (à 1 près),
  - de trier récursivement ces deux listes,
  - de fusionner les deux listes triées obtenues.

Un exemple fondamental : le tri fusion

Implémentation

Il nous faut donc au préalable écrire deux fonctions :

- Une fonction `fission`, qui divise une liste en deux listes de même taille à un élément près.
- Une fonction `fusion`, qui prend en entrée deux listes triées dans l'ordre croissant, et renvoie la liste triée constituée des éléments des deux listes.

Une fois ces deux fonctions écrites, l'implémentation du tri n'est pas difficile (attention à ne pas oublier le cas terminal).

Implémentation du tri fusion

```
1  let rec fission l = match l with
2    | [] | [_] -> l, []
3    | x::y::q -> let a,b=fission q in x::a, y::b
4  ;;
5
6  let rec fusion l1 l2 = match l1, l2 with
7    | [],_ -> l2
8    | _, [] -> l1
9    | x::q1, y::_ when x<=y -> x::(fusion q1 l2)
10   | _,x::q2 -> x::(fusion l1 q2)
11 ;;
12
13 let rec tri_fusion l = match l with
14   | [] | [_] -> l
15   | _ -> let a,b=fission l in fusion (tri_fusion a) (tri_fusion b)
16 ;;
```

Remarques sur le tri fusion

Analyse du tri fusion

On laisse en exercice la preuve de terminaison et de correction de ce tri.

L'intérêt du tri fusion est que, comme on va le voir dans cette partie, sa complexité dans le pire des cas est $O(n \log n)$ où n est la taille de la liste à trier.

Remarques sur le tri fusion

Tri d'une liste en OCaml

La fonction `List.sort` d'OCaml trie une liste. Elle utilise le **tri fusion**.

Elle est un peu plus générique que notre fonction car elle prend en paramètre une fonction de comparaison. C'est une fonction curryfiée à deux argument x et y de même type, qui renvoie :

- 0 si $x = y$;
- -1 si $x < y$;
- 1 si $x > y$.

Remarques sur le tri fusion

Exemple

Voici comment l'utiliser pour trier une liste d'entiers (ou de flottants) suivant l'ordre usuel :

```
# let f = fun x y -> if x = y then 0 else if x<y then -1 else 1 ;;
val f : 'a -> 'a -> int = <fun>
# List.sort f [5; 2; 3; 7; 59; 2; 8 ; 489; 44; 498; 11566; 16] ;;
- : int list = [2; 2; 3; 5; 7; 8; 16; 44; 59; 489; 498; 11566]
```

Complexité des fonctions récursives

Complexité

La **complexité** d'une fonction récursive vérifie une relation de récurrence de la forme $C(n) = \sum C(n_i) + f(n)$ qu'il va falloir résoudre.

Les $C(n_i)$ correspondent à la complexité engendrée par les appels récursifs.

Le terme $f(n)$ correspond au coût de la fonction hors appels récursifs.

Complexité des fonctions récursives

Exemple

Voici les relations obtenues pour divers exemples vus plus tôt :

- Factorielle : $C(n) = C(n-1) + O(1)$.
- Tri par sélection récursif : $C(n) = C(n-1) + O(n)$.
- Tri fusion : $C(n) = C(\lfloor \frac{n}{2} \rfloor) + C(\lceil \frac{n}{2} \rceil) + O(n)$.
- Tours de Hanoï : $C(n) = 2C(n-1) + O(1)$.

Remarque

Souvent, le O pourra être précisé en un Θ.

Dans ce cas, ce sera aussi vrai pour le résultat.

Complexité des fonctions récursives

Premiers résultats

Les résultats qu'on va voir ici permettent de résoudre des "récurrences simples" comme celles satisfaites par la complexité de factorielle, du tri par sélection, ou de la fonction résolvant les tours de Hanoï.

Premières propriétés

Proposition (Comparaison de sommes)

Soit (b_n) et (b'_n) deux suites réelles positives.

Si $b_n = \Theta(b'_n)$, alors $\sum_{k=0}^{n} b_k = \Theta(\sum_{k=0}^{n} b'_k)$.

Preuve

Les deux sommes sont soit toutes deux convergentes, soit toutes deux divergentes.

Dans le premier cas, elles sont toutes deux un $\Theta(1)$.

Dans le second, soit $C > 0$ et n_0 tels que $\forall n \geq n_0,\ b_n \leq C b'_n$.

Alors c'est aussi le cas pour la somme : $\sum_{k=n_0}^{n} b_k \leq C \sum_{k=n_0}^{n} b'_k$.

Premières propriétés

Proposition (Comparaison de sommes)

Soit (b_n) et (b'_n) deux suites réelles positives.

Si $b_n = \Theta(b'_n)$, alors $\sum_{k=0}^{n} b_k = \Theta(\sum_{k=0}^{n} b'_k)$.

Preuve

Puisque les sommes sont divergentes, la somme de droite est non nulle à partir d'un certain rang (qu'on peut supposer être n_0), et donc pour $C' > C$, il existe un rang n à partir duquel $\sum_{k=0}^{n} b_k \leq C' \sum_{k=n_0}^{n} b'_k \leq C' \sum_{k=0}^{n} b'_k$.

Ainsi, $\sum_{k=0}^{n} b_k = O(\sum_{k=0}^{n} b'_k)$.

On montre de même que $\sum_{k=0}^{n} b'_k = O(\sum_{k=0}^{n} b_k)$.

Premières propriétés

Proposition (Sommations classiques)

Soit $\alpha \geq 0$ et $q > 1$. Alors :

- $\sum_{k=0}^{n} k^\alpha = \Theta(n^{\alpha+1})$
- $\sum_{k=0}^{n} q^k = \Theta(q^n)$

Preuve

- la fonction $t \mapsto t^\alpha$ est croissante, donc :

$$\frac{n^{\alpha+1}}{\alpha+1} = \int_0^n t^\alpha dt \leq \sum_{k=1}^{n} k^\alpha \leq \int_1^{n+1} t^\alpha dt = \frac{(n+1)^{\alpha+1}-1}{\alpha+1}$$

  Les deux termes à gauche et à droite sont des $\Theta(n^{\alpha+1})$, donc $\sum_{k=1}^{n} k^\alpha$ aussi.
- $\sum_{k=0}^{n} q^k = \frac{q^{n+1}-1}{q-1} = \Theta(q^n)$.

Exemples

Exemple

La solution de la récurrence $C(n) = C(n-1) + O(1)$ est $C(n) = O(n)$.

En effet, on a $C(n) - C(n-1) = O(1)$, donc par sommation télescopique, on a :

$$\begin{aligned} C(n) - C(0) &= \sum_{k=0}^{n-1} (C(k+1) - C(k)) \\ &= O(\sum_{k=0}^{n-1} 1) \\ &= O(n) \end{aligned}$$

La complexité de **factorielle** est donc un $O(n)$.

Exemples

Exemple

La solution de la récurrence $C(n) = C(n-1) + O(n)$ est $C(n) = O(n^2)$.

En effet, on a $C(n) - C(n-1) = O(n)$, donc par sommation télescopique, on a :

$$\begin{aligned} C(n) - C(0) &= \sum_{k=0}^{n-1} (C(k+1) - C(k)) \\ &= O(\sum_{k=0}^{n-1} k) \\ &= O(n^2) \end{aligned}$$

La complexité du **tri par sélection** est donc un $O(n^2)$.

Premières propriétés

Théorème

Soit (u_n) vérifiant la relation $u_{n+1} = au_n + f(n)$, avec f une fonction strictement positive, $a > 0$, $u_0 \geq 0$.
Soit $b > 0$ tel que $f(n) = \Theta(b^n)$.

- Si $b < a$, alors $u_n = \Theta(a^n)$.
- Si $b = a$, alors $u_n = \Theta(na^n)$.
- Si $b > a$, alors $u_n = \Theta(b^n)$.

Premières propriétés

Preuve

Posons $v_n = \frac{u_n}{a^n}$ (i.e. $a^n v_n = u_n$).
On a donc $a^{n+1} v_{n+1} = a \times a^n v_n + f(n)$.
Donc $v_{n+1} = v_n + \frac{f(n)}{a^{n+1}}$. Ainsi :

$$v_n = v_0 + \sum_{k=1}^{n} (v_k - v_{k-1}) = v_0 + \sum_{k=0}^{n-1} \frac{f(k)}{a^{k+1}}$$

Donc $u_n = a^n \left(v_0 + \sum_{k=0}^{n-1} \frac{f(k)}{a^{k+1}} \right)$. De plus, $f(k) = \Theta(b^k)$, donc la sommation classique des suites géométriques donne :

$$\sum_{k=0}^{n-1} \frac{f(k)}{a^{k+1}} = \begin{cases} \Theta(1) & \text{si } b < a \\ \Theta(n) & \text{si } b = a \\ \Theta\left((\frac{b}{a})^n \right) & \text{si } b > a \end{cases}$$

Et le résultat s'en suit par comparaison de somme.

Premières propriétés

Exemple

La complexité de l'algorithme résolvant le problème des tours de Hanoï vérifie $C(n) = 2C(n-1)+\Theta(1)$, donc $C(n) = \Theta(2^n)$.

Remarque

Si $f(n) = \Theta(n^\alpha)$ avec α quelconque et $a > 1$, alors $u_n = \Theta(a^n)$.

La démonstration est la même que précédemment.

Récurrences “diviser pour régner”

Diviser pour régner

Pour résoudre certains problèmes, une technique algorithmique appelée “diviser pour régner” peut être très adaptée.

C’est la technique qu’on a utilisé pour le tri fusion, et dont on verra d’autres exemples dans le prochain chapitre.

Elle consiste à :

1. **Diviser** : découper un problème initial en sous-problèmes ;
2. résoudre les sous-problèmes récursivement (ou directement s’ils sont assez petits) ;
3. **Régner** : calculer une solution au problème initial à partir des solutions des sous-problèmes.

Récurrences “diviser pour régner”

Complexité

Dans ce cas, pour calculer la complexité, on obtiendra en général une récurrence de la forme :

$$C(n) = a_1 \cdot C(\lfloor n/b \rfloor) + a_2 \cdot C(\lceil n/b \rceil) + f(n)$$

Abus de notation

Dans ce cas, en posant $a = a_1 + a_2$, on utilise souvent l'abus de notation suivant :

$$C(n) = a \cdot C(n/b) + f(n)$$

Théorème maître

Théorème

Soit C une solution de l'équation suivante :

$$C(n) = a \cdot C(n/b) + f(n)$$

où $a \geq 1$, $b \geq 2$, et f est une fonction croissante telle que $f(n) = \Theta(n^\beta)$ avec $\beta \in \mathbb{R}$.

Alors, en notant $\alpha = \log_b(a)$:

- Si $\alpha > \beta$, $C(n) = \Theta(n^\alpha)$.
- Si $\alpha < \beta$, $C(n) = \Theta(n^\beta)$.
- Si $\alpha = \beta$, $C(n) = \Theta(n^\alpha \cdot \log(n))$.

Remarque

Le théorème est toujours vrai en remplaçant les Θ par des O.

Preuve

On commence par prouver le résultat seulement pour les n de la forme b^k.

On a donc $C(b^k) = a \cdot C(b^{k-1}) + \Theta(b^{\beta k})$.

Cette relation de récurrence se réécrit en :

$$\frac{C(b^k)}{a^k} = \frac{C(b^{k-1})}{a^{k-1}} + \Theta\left(\left(\frac{b^\beta}{a}\right)^k\right)$$

Donc :

$$\frac{C(b^k)}{a^k} - \frac{C(b^{k-1})}{a^{k-1}} = \Theta\left(\left(\frac{b^\beta}{a}\right)^k\right)$$

Preuve

En sommant les termes, on obtient :

$$\sum_{i=1}^{k} \left[\frac{C(b^i)}{a^i} - \frac{C(b^{i-1})}{a^{i-1}} \right] = \sum_{i=1}^{k} \Theta\left(\left(\frac{b^\beta}{a} \right)^i \right)$$

$$\frac{C(b^k)}{a^k} - C(1) = \Theta\left(\sum_{i=1}^{k} \left(\frac{b^\beta}{a} \right)^i \right)$$

Ainsi :

- Si $a > b^\beta$, alors $\sum_{i=1}^{k} \left(\frac{b^\beta}{a} \right)^i = \Theta(1)$, donc $C(b^k) = \Theta(a^k)$.
  Or $a^k = (b^{\log_b(a)})^k = (b^k)^\alpha$. Donc $C(b^k) = \Theta((b^k)^\alpha)$.
- Si $a < b^\beta$, alors $\sum_{i=1}^{k} \left(\frac{b^\beta}{a} \right)^i = \Theta((\frac{b^\beta}{a})^k)$, donc
  $C(b^k) = \Theta((b^\beta)^k) = \Theta((b^k)^\beta)$.
- Si $a = b^\beta$, alors $\sum_{i=1}^{k} \left(\frac{b^\beta}{a} \right)^i = \Theta(k)$, donc
  $C(b^k) = \Theta(a^k \cdot k) = \Theta((b^k)^\alpha \cdot \log(b^k))$.

Preuve

Lemme

Si f est croissante, alors C est croissante.

Preuve

Montrons par récurrence sur $n \in \mathbb{N}^*$ que $C(n) \leq C(n+1)$.

- Initialisation : $C(2) = a \cdot C(1) + f(1) \geq C(1)$
- Hérédité :

$$C(n+1) - C(n) = a(\underbrace{C((n+1)/b) - C(n/b)}_{\geq 0 \text{ par HR}}) + \underbrace{f(n+1) - f(n)}_{\geq 0}$$

Preuve

Soit $n \in \mathbb{N}^*$. $\exists! i \in \mathbb{N}$ tel que $b^i \leq n < b^{i+1}$.

Par croissance de C, on a donc $C(b^i) \leq C(n) < C(b^{i+1})$.

On prouve le théorème pour le cas $\alpha > \beta$ (les autres cas se montrent de manière similaire).

On a $C(b^k) = \Theta((b^k)^\alpha)$, i.e.

$$\exists k_1, k_2 > 0, k_1 \cdot (b^i)^\alpha \leq C(b^i) \leq k_2 \cdot (b^i)^\alpha$$

Donc

$$\frac{k_1}{b^\alpha} \cdot n^\alpha \leq k_1 \cdot b^{i\alpha} \leq C(n) \leq k_2 \cdot b^{(i+1)\alpha} \leq k_2 b^\alpha \cdot n^\alpha$$

Ainsi, $C(n) = \Theta(n^\alpha)$.

Exemple

Exemple

La complexité du tri fusion vérifie les hypothèses du théorème maître, avec :

- $a = b = 2$ (donc $\alpha = \log_b(a) = 1$) ;
- et $f(n) = \Theta(n)$ (donc $\beta = 1$).

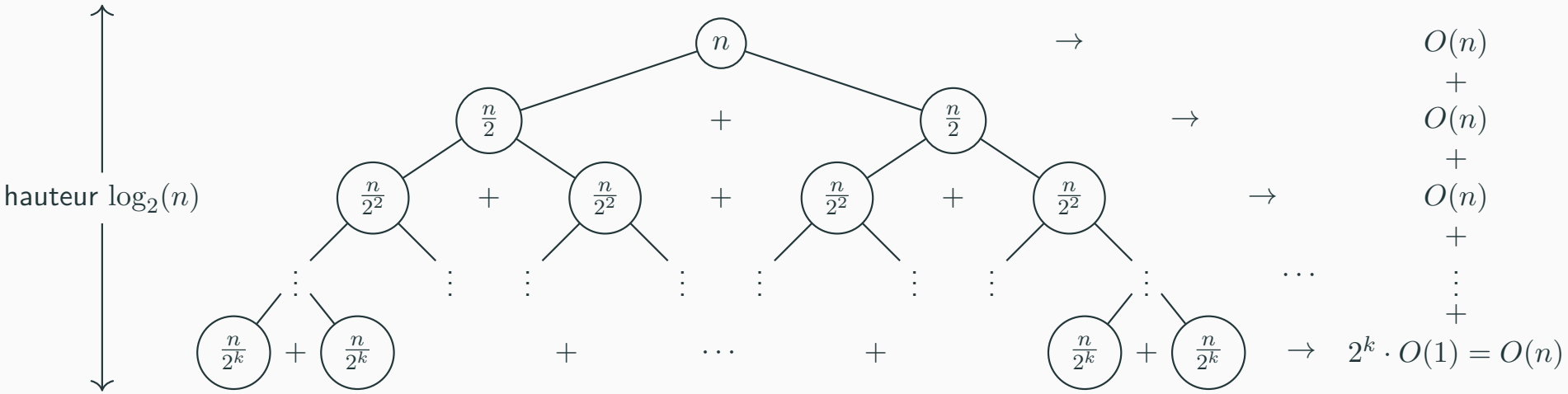
Ainsi, on obtient que la complexité du tri fusion dans le pire des cas est un $\Theta(n \log n)$.

Intuition derrière le théorème maître

Intuition

- Si $\alpha > \beta$, cela signifie qu'on fait beaucoup d'appels récursif, suffisamment pour rendre les coûts hors appels récursifs négligeables.
- Si $\alpha < \beta$, c'est l'inverse : les coûts hors appels récursifs l'emportent, et rendent les coûts des appels récursifs négligeables.
- Si $\alpha = \beta$, alors les coûts s'équilibrent.
  Pour comprendre ce qui se passe, représentons l'arbre des appels récursifs effectués.

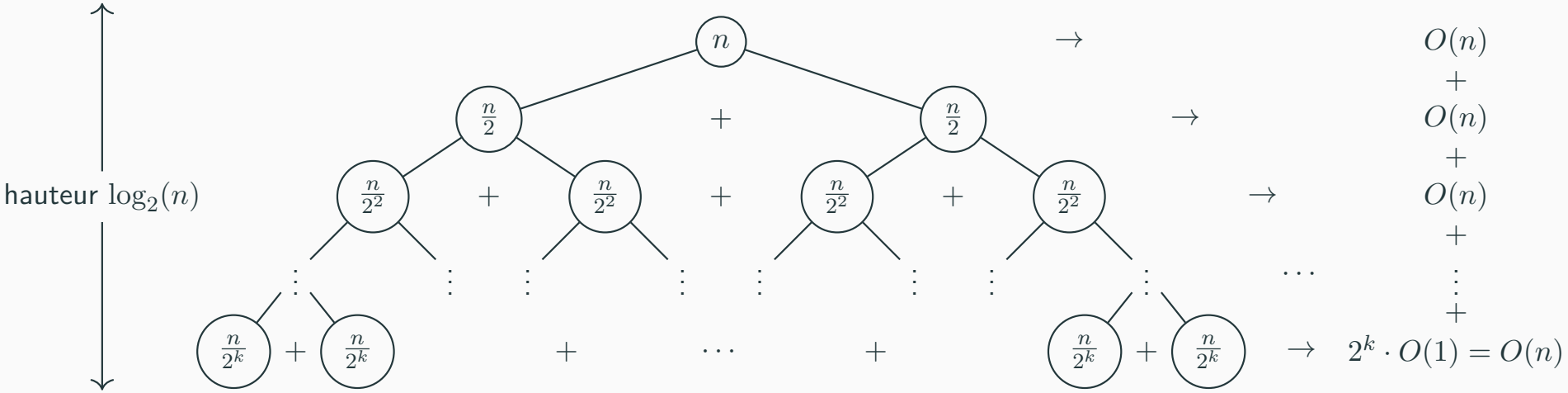
Intuition derrière le théorème maître



Appels récursifs du tri fusion

On représente ci-dessus les appels récursifs du tri fusion, dont la complexité satisfait la relation : $C(n) = 2C(n/2) + O(n)$, sur une liste de taille $n = 2^k$.

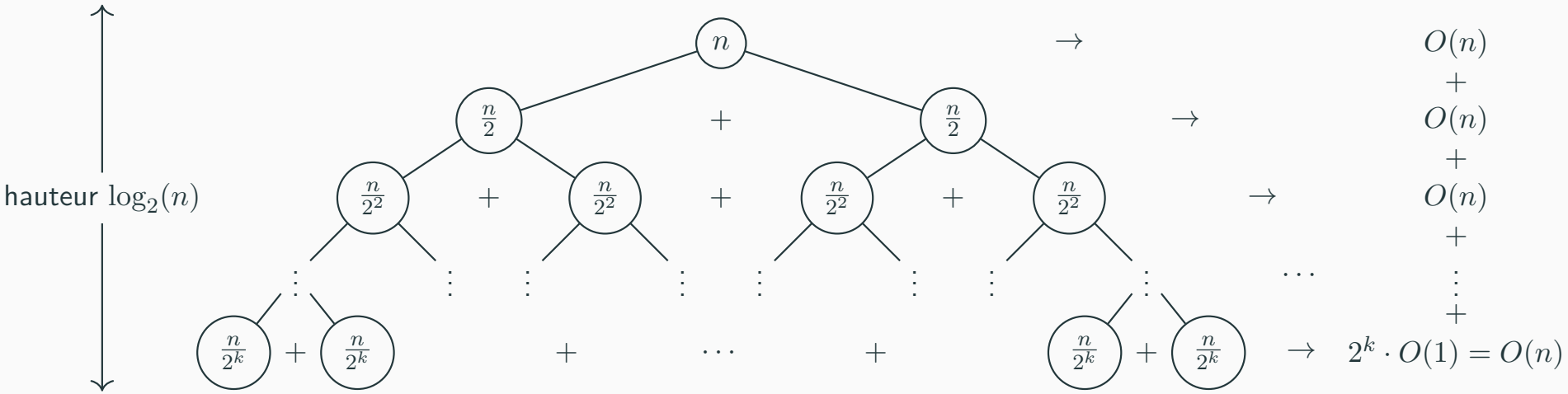
Intuition derrière le théorème maître



Appels récursifs du tri fusion

Chaque **noeud** de cet arbre possède deux **fils** représentant les 2 appels récursifs (on dit que c'est un **arbre binaire**), à l'exception des noeuds tout en bas (qu'on appelle les **feuilles**) qui n'ont pas de fils : ils représentent les cas de base.

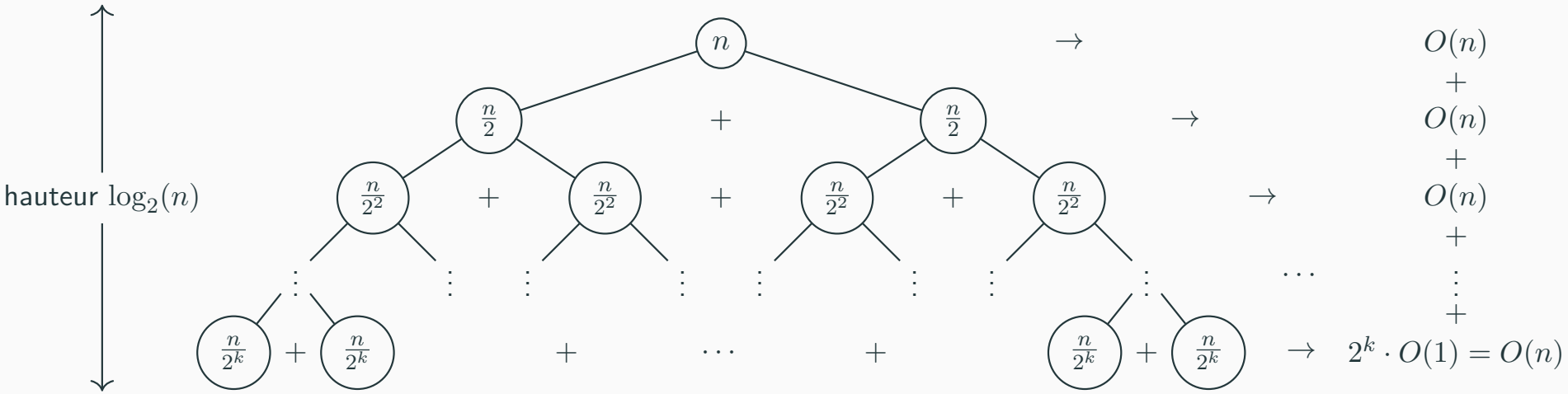
Intuition derrière le théorème maître



Appels récursifs du tri fusion

Cet arbre possède $2^k = n$ feuilles. On peut montrer par récurrence qu'un arbre binaire ayant n feuilles a une hauteur de $\lceil \log_2(n) \rceil$.

Intuition derrière le théorème maître



Appels récursifs du tri fusion

Sur cette figure, on remarque que chaque étage représente tous les appels récursifs d'une certaine taille, et que chacun de ces $O(\log n)$ étages contribue de manière équilibrée à un coût de $O(n)$. D'où la complexité totale en $O(n \log n)$.