Compléments sur la récursion

Pierre Karpman

Lycée Champollion MP2I

https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

Table des matières

Récursion mutuelle (fonctions)

Principe

Définir *ensemble* plusieurs fonctions récursives : `f1 x` est définie en fonction de `f2 x'` ; `f2 y` est définie en fonction de `f1 y'`

Bien défini si les appels à `f1` émis par `f2 x'` le sont pour des arguments « strictement plus petits » que `x` (et pareil pour les appels à `f2` de `f1`)

Exemple

Exemple (canonique) : 0 est pair, et un nombre entier $n > 0$ est pair (resp. impair) si $n - 1$ est impair (resp. pair). (Une façon efficace de déterminer la parité !)

Syntaxe

OCaml

```
let rec f1 = ... and f2 = ...
```

C

Nécessite de « connaître » le nom de chaque fonction appelée : besoin de déclarer la signature de l'une d'entre elles avant sa définition

Python

Rien à faire…

Exemples

OCaml

```
let rec
    even n = if n = 0 then true else odd (n - 1)
and
    odd n  = if n = 0 then false else even (n - 1)
```

C

```
bool odd(unsigned n);
bool even(unsigned n) {
    if (n == 0) { return true; }
    return odd(n - 1);
}
bool odd(unsigned n) {
    if (n == 0) { return false; }
    return even(n - 1);
}
```

Exemples (suite)

Python

```
def even(n:int):
    if n == 0:
        return True
    return odd(n - 1)

def odd(n:int):
    if n == 0:
        return False
    return even(n - 1)
```

Récursion mutuelle (types)

En OCaml, on peut aussi définir des *types* mutuellement récursifs sur le même principe

Exemple

```
type evenNat = Zero
             | SumEE  of evenNat * evenNat
             | SumOO  of oddNat  * oddNat
             | ProdEE of evenNat * evenNat
             | ProdOE of oddNat  * evenNat
             | ProdEO of evenNat * oddNat
and  oddNat  = One
             | SumOE  of oddNat  * evenNat
             | SumEO  of evenNat * oddNat
             | ProdOO of oddNat  * oddNat

type nat = Even of evenNat | Odd of oddNat
```

Table des matières

Rappel : pile d'appel

Principe

Logiquement et physiquement, la plupart du temps, les données nécessaires à l'exécution d'une fonction (notamment les valeurs des arguments ; des variables locales…) sont placées dans une *pile*.

- Un appel de fonction (y compris un appel récursif) *empile* les données nécessaires sur le sommet courant de la pile.
- Un retour d'appel de fonction (ayant terminé son exécution) *dépile* les données du sommet courant de la pile.

Visualisation (rappel)

- En C : avec un débuggeur et la *backtrace*
- En OCaml : dans `utop` avec `#trace`

Exemple : factorielle

5!

Empilements et dépilement dans un calcul récursif de 5!

```
utop # #trace fact;;
fact is now traced.
utop # fact 5;;
fact <-- 5
fact <-- 4
fact <-- 3
fact <-- 2
fact <-- 1
fact --> 1
fact --> 2
fact --> 6
fact --> 24
fact --> 120
- : int = 120
```

Exemple : even 5

5 est-il un nombre pair ?

Empilement et dépilements dans un calcul de even 5 pour les fonctions mutuellement récursives even & odd.

```
even <-- 5
odd <-- 4
even <-- 3
odd <-- 2
even <-- 1
odd <-- 0
odd --> false
even --> false
odd --> false
even --> false
odd --> false
even --> false
```

Exemple : `odd 3`

3 est-il un nombre impair ?

Pareil, pour `odd 3`.

```
odd <-- 3
even <-- 2
odd <-- 1
even <-- 0
even --> true
odd --> true
even --> true
odd --> true
```

Arbre d'appel

Principe

Logiquement (mais *pas* physiquement), une fonction récursive effectuant plusieurs appels récursifs a une structure d'appel en forme d'*arbre* (non trivial)

Exemple

Implémentation naïve du calcul des termes de la suite « de Fibonacci »

```
let rec fib n = if n < 2 then n else fib (n - 1) + fib (n - 2)
```

Attention

De tels fonctions ont souvent un coût **exponentiel** en « quelque chose » (qui si pas « très petit » donne un coût total important ; *cf.* plus loin)

Il faut absolument éviter les appels récursifs multiples quand c'est possible.

Parcours d'arbre

Principe

Les appels récursifs multiples sont empilés et dépilés sur la pile, qui est une structure linéaire (pas *arborescente*)

L'ordre d'empilement (resp. dépilement) des appels est donné par un *parcours* d'arbre en profondeur *préfixe* (resp. *postfixe*) (plus de détails plus tard)

On peut visualiser ces parcours en utilisant à nouveau `#trace`.

Exemple `fib`

`fib 3`

Empilements et dépilement dans un calcul récursif naïf de F_3

```
utop # fib 3;;
fib <-- 3
    fib <-- 1
    fib --> 1
    fib <-- 2
        fib <-- 0
        fib --> 0
        fib <-- 1
        fib --> 1
    fib --> 1
fib --> 2
- : int = 2
```

Table des matières

Rappel : objectifs d'une analyse de coût

Objectifs (par défaut)

- Coût temporel pire cas
  - Dans un modèle RAM gentil
- En fonction de la *taille* des entrées
- Généralement exprimé par majoration asymptotique
  - La plus fine possible !

« $T(n) = O(f(n))$ »

Exemples de résultats

- Trier une liste de n entiers machine avec un tri par insertion : coût $O(n^2)$
- Trier une liste de n entiers machine avec un tri fusion : coût $O(n \log n)$
  - Aussi $O(n^2)$, mais…
- Recherche d'élément dans un tableau d'entiers machine de taille n : $O(n)$
- Recherche d'élément par dichotomie dans un tableau trié d'entiers machine de taille n : $O(\log n)$
  - Aussi $O(n)$, mais…

Fonctions récursives avec Un appel récursif : $T(n) \leq T(\delta(n)) + D(n)$

Comme une boucle

- Pas d'arbre de récursion ; relativement simple
- δ : majore la décroissance de la taille ; D : majore le coût marginal pour une taille
- $T(n) \leq D(n) + D(\delta(n)) + \cdots + D(1)$
  - $T(n) \leq \sum_{i=0}^{\infty} D(\delta^i(n))[\delta^i(n) \geq 1]$

Exemple : $T(n) \leq T(n/2) + a$

Par ex. recherche dichotomique dans un tableau trié (récursive) ; exponentiation rapide

- cas de base en coût constant b
- $\delta(n) \mapsto n/2$ (division entière, éventuellement à -1 près…)
- $D(n) \mapsto a$ une constante

$$T(1) = b \qquad T(n) \leq T(n/2) + a$$

- Informel : au plus $\log(n)$ appels réc. et un appel au cas de base : $T(n) \leq a\log(n) + b$

$T(n) \leq T(n/2) + a$: sur des puissances de deux

Hypothèses

- $T(1) = b$
- $T(n) \leq T(n/2) + a$

Objectif

Montrer que $T(2^n) \leq a \log(2^n) + b = an + b$

Par récurrence

- Cas de base : $T(2^0) = T(1) = b \leq b$
- Conservation :

$$
\begin{aligned}
T(2^n) &\leq an + b && \text{Hyp. de récurrence} \\
T(2^{n+1}) &\leq T(2^n) + a && \text{Hyp. initiale} \\
&\leq an + b + a = a(n+1) + b && \text{Substitution + calcul}
\end{aligned}
$$

Un appel récursif : second exemple

Exemple : $T(n) \leq T(n-1) + an$

Par ex. tri par insertion (par ex. TP12)

- Cas de base en coût constant b
- $\delta(n) \mapsto n-1$
- $D(n) \mapsto an$ linéaire

$$T(n) \leq T(n-1) + an$$

- Informel : au plus n appels : $T(n) \leq b + a\sum_{i=1}^{n}(i-1) = an(n+1)/2 + b$
- Par récurrence :
  - Cas de base : $T(1) = b \leq b + a$
  - Conservation :

$$\begin{aligned}
T(n) &\leq an(n+1)/2 + b && \text{Hyp. de récurrence} \\
T(n+1) &\leq T(n) + a(n+1) && \text{Hyp. initiale} \\
&\leq a[n(n+1)/2 + (n+1)] + b && \text{Substitution} \\
&= a[(n+2)(n+1)/2)] + b && n+1 = 2(n+1)/2
\end{aligned}$$

Plusieurs appels récursifs : $T(n) = \sum_{i=1}^{k} T(\delta_i(n)) + D(n)$

Utilisation de l'arbre d'appel

Approche informelle mais souvent suffisante :

- On dessine l'arbre d'appel récursif
- On regroupe & exprime les coûts marginaux par taille d'entrée (par « niveau de *profondeur* »)
- On borne le nombre de niveaux (la *hauteur* de l'arbre)
- On somme le tout (avec pondération)

Exemple : $T(n) \leq 2T(n/2) + an$

Par ex. tri fusion

- Cas de base de coût $\leq b$
- Au plus $\log(n)$ *niveaux*
- Chaque niveau de coût marginal $\leq an$ (bn pour les *feuilles*)

$$T(n) \leq an\log(n) + bn$$

$T(n) \leq 2T(n/2) + an$: sur des puissances de deux

Hypothèses

- $T(1) = b$
- $T(n) \leq 2T(n/2) + an$

Objectif

Montrer que $T(2^n) \leq an2^n + b2^n$

Par récurrence

- Cas de base : $T(2^0) = T(1) = b \leq b$
- Conservation :

$$
\begin{aligned}
T(2^n) &\leq an2^n + b2^n && \text{Hyp. de récurrence} \\
T(2^{n+1}) &\leq 2T(2^n) + a2^{n+1} && \text{Hyp. initiale} \\
&\leq 2(an2^n + b2^n) + a2^{n+1} && \text{Substitution} \\
&= an2^{n+1} + b2^{n+1} + a2^{n+1} \\
&= a(n+1)2^{n+1} + b2^{n+1}
\end{aligned}
$$

Exercice : $T(n) \leq 2T(n/2) + a$

- $T(1) = a$
- $T(n) \leq 2T(n/2) + a$
- Avec l'arbre d'appel, puis sur des puissances de deux

Exemple

Exponentiation rapide pas rapide

$T(n) \leq 2T(n/2) + a$: sur des puissances de deux

Hypothèses

- $T(1) = a$
- $T(n) \leq 2T(n/2) + a$

Objectif

Montrer que $T(2^n) \leq a(2^{n+1} - 1)$

Par récurrence

- Cas de base : $T(2^0) = T(1) = a \leq a$
- Conservation :

$$
\begin{aligned}
T(2^n) &\leq a(2^{n+1} - 1) && \text{Hyp. de récurrence} \\
T(2^{n+1}) &\leq 2T(2^n) + a && \text{Hyp. initiale} \\
&\leq 2a(2^{n+1} - 1) + a && \text{Substitution} \\
&= a2^{n+2} - 2a + a \\
&= a(2^{n+2} - 1)
\end{aligned}
$$

Exercice : $T(n) \leq 3T(n/2) + an$

- $T(1) = a$
- $T(n) \leq 3T(n/2) + an$
- Avec l'arbre d'appel, puis sur des puissances de deux

Exemple

Algorithme de multiplication « de Karatsuba »

Idée sur des polynômes :

- $(p_1X^n + p_0) \times (q_1X^n + q_0) = p_1q_1X^{2n} + (p_1q_0 + p_0q_1)X^n + p_1q_0$
  - Avec $\deg(p_0)$, $\deg(p_1)$, $\deg(q_0)$, $\deg(q_1)$ tous $< n$
- Calculable avec trois produits en degré $< n +$ $O(n)$ additions :
  - H := p_1q_1
  - L := p_0q_0
  - M := $(p_1 + p_0) \times (q_1 + q_0) \rightsquigarrow p_1q_0 + p_0q_1 = M - H - L$

$T(n) \leq 3T(n/2) + an$: sur des puissances de deux

Objectif (dégradé)

Montrer que $T(2^n) \leq a3^n + o(3^n)$
($T(2^n) \leq a2^{\log(3)n}$; $T(n) \leq n^{\log(3)} + \ldots$ pour n une puissance de deux)

Par récurrence

- Cas de base : $T(2^0) = T(1) = a \leq a$
- Conservation :

$$
\begin{aligned}
T(2^n) &\leq a3^n + o(3^n) && \text{Hyp. de récurrence} \\
T(2^{n+1}) &\leq 3T(2^n) + a2^{n+1} && \text{Hyp. initiale} \\
&\leq 3a(3^n + o(3^n)) + a2^{n+1} && \text{Substitution} \\
&= a(3^{n+1} + 2^{n+1} + o(3^n)) \\
&= a(3^{n+1} + o(3^{n+1}))
\end{aligned}
$$

$T(n) \leq 3T(n/2) + an$: sur des puissances de deux *bis*

Quelle formule sans o ?

- Calculer les premiers termes (pour $a = 1$) : 1, 5, 19, 65, 211
- ~~Rechercher une suite candidate sur l'OEIS~~ intuiter que cela correspond à $3^{n+1} - 2^{n+1}$
- Le montrer par récurrence

Par récurrence

- Cas de base : $T(2^0) = T(1) = 1 \leq 1$ (prenons $a = 1$…)
- Conservation :

$$
\begin{aligned}
T(2^n) &\leq 3^{n+1} - 2^{n+1} && \text{Hyp. de récurrence} \\
T(2^{n+1}) &\leq 3T(2^n) + 2^{n+1} && \text{Hyp. initiale} \\
&\leq 3(3^{n+1} - 2^{n+1}) + 2^{n+1} && \text{Substitution} \\
&= 3^{n+2} - 2 \times 2^{n+1} \\
&= 3^{n+2} - 2^{n+2}
\end{aligned}
$$