Diviser pour régner

C. Charignon

L'expression « diviser pour régner » est utilisée lorsqu'on résout un problème en le divisant en problèmes plus petit (souvent deux problèmes deux fois plus petits), qu'on résout récursivement. On peut distinguer trois étapes dans la résolution d'un problème par cette méthode :

1. diviser le problème en problèmes plus petits (c'est l'étape « diviser ») ;
2. résoudre récursivement les petits problèmes ;
3. en déduire une solution du gros problème initial (étape « régner »).

La dichotomie déjà étudiée est un premier exemple de telle méthode. On va donner deux autres exemples dans ce chapitre ; ces deux exemples sont à connaître.

Table des matières

I Cours **2**

1 Puissances divisées **2**
1.1 Principe . . . 2
1.2 Programmation Caml . . . 2
1.3 Complexité . . . 2

2 Tri fusion **2**
2.1 Le principe . . . 2
2.2 Implantation Caml . . . 3
2.3 Complexité . . . 3
2.3.1 Préliminaires . . . 3
2.3.2 Cas où la longueur de la liste est une puissance de deux . . . 4
2.3.3 Cas général . . . 4

II Exercices **5**

1 Diviser pour régner **1**

2 Tri **1**

Première partie
Cours

1 Puissances divisées

1.1 Principe

Exemple : Soit x un flottant, calculons x^8. L'écriture naïve $x^8 = x*x*x*x*x*x*x*x$ conduit à 8 multiplications.

L'écriture $x^8 = \left(\left(x^2\right)^2\right)^2$ n'en utilise plus que 3 !

Voyons comment utiliser la stratégie « diviser pour régner » pour un calcul de puissances. Soit $x \in \mathbb{R}$ et $n \in \mathbb{N}$. Pour calculer x^n, nous allons :

1. **Diviser** : à condition que $n > 0$, posons $q = \left\lfloor \frac{n}{2} \right\rfloor$.
2. **Appel récursif** : on calcule x^q.
3. **Régner** :
   - Si n est pair, alors $n = 2q$, et $x^n = (x^q)^2$;
   - Si n est impair, alors $n = 2q+1$ et $x^n = x\,(x^q)^2$.

1.2 Programmation Caml

Ceci conduit à la fonction suivante :

```
let rec puissance x n=
  if n=0 then 1
  else let y=puissance x (n/2) in
    if n mod 2=0 then
      y*y
    else
      y*y*x
;;
```

1.3 Complexité

On fixe $x \in \mathbb{N}$. Pour tout $n \in \mathbb{N}$, soit C_n le nombre de multiplications pour calculer `puissance x n`. On a :

$$\begin{cases} C_0 = 0 \\ \forall n \in^*,\ C_n \leqslant C_{\left\lfloor \frac{n}{2} \right\rfloor} + 2 \end{cases}$$

On prouve alors par récurrence forte que $\forall n \in \mathbb{N}^*,\ C_n \leqslant 2\log_2(n) + 1$.

2 Tri fusion

2.1 Le principe

On propose maintenant de trier une liste[1] par une méthode de type « diviser pour régner ».
Soit `l` une liste. Pour créer une nouvelle liste contenant les éléments de `l` mais triés, nous allons :

- **Diviser :** couper `l` en deux sous-listes d'à peu près même longueur `l1` et `l2`
- **Appels récursifs** : récupérer ure version triée `l1Triee` et `l2Triee` de chaque petite liste ;
- **Régner** : rassembler `l1Triee` et `l2Triee` en une seule liste triée.

La première opération s'appelle « partition » et la troisième « fusion ».

1. le tri de listes ou de tableaux constitue fait partie des « gammes » d'un informaticien. Il existe de nombreuses méthodes et leur étude constitue un exercice classique et formateur. Ce sera l'objet d'un chapitre entier en seconde année.

2.2 Implantation Caml

On va utiliser deux fonctions préliminaires : une fonction `partition` et une fonction `fusion`.

```
let rec partition l=
  (* renvoie une partition de l en deux listes de longueurs égales à +-1 près. *)
    match l with
        |[] -> ([], [])
        |[x] -> ( [x], [])
        |x::(y::q)-> let (l1,l2)= partition q in
                        (x:: l1, y::l2)
;;
```

```
let rec fusion l1 l2=
  (* l1 et l2 sont deux listes triées.
  Renvoie la liste obtenue en rassemblant l1 et l2 en une liste triée.*)

    match (l1,l2) with
        | ([], l)               -> l
        | (l,[])                -> l
        | (x::l, y::m) when x<y -> x :: fusion l l2
        | (x::l, y::m)          -> y :: fusion l1 m
;;
```

La fonction de tri s'obtient alors facilement :

```
let rec tri l=
    match l with
        |[] -> []
        | [x] -> l
        | _ -> let (l1,l2)=partition l in
                    fusion (tri l1) (tri l2)
;;
```

2.3 Complexité

2.3.1 Préliminaires

Avant de commencer, un petit lemme, utile lorsqu'on découpe un entier en deux entiers :

Lemme 2.1. *Soit $n \in \mathbb{N}$. Alors :*

$$n = \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n+1}{2} \right\rfloor$$

Ainsi, lorsqu'on divise une liste de longueur l en deux partie égales à plus ou moins 1 près, un morceau sera de longueur $\left\lfloor \frac{n}{2} \right\rfloor$ et l'autre de longueur $\left\lfloor \frac{n+1}{2} \right\rfloor$.

Démonstration :

- Si n est impair : soit $k \in \mathbb{N}$ tel que $n = 2k$, alors :

$$\frac{n}{2} + \left\lfloor \frac{n+1}{2} \right\rfloor = \left\lfloor \frac{2k}{2} \right\rfloor + \left\lfloor \frac{2k+1}{2} \right\rfloor = k + k = n$$

- Si n est impair : soit $k \in \mathbb{N}$ tel que $n = 2k+1$, alors :

$$\left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n+1}{2} \right\rfloor = \left\lfloor \frac{2k+1}{2} \right\rfloor + \left\lfloor \frac{2k+2}{2} \right\rfloor = k + k + 1 = n$$

□

Étudions la complexité des fonctions intermédiaires. On compte le nombre de comparaisons entre éléments de la liste à trier.

- Aucune comparaison dans `partition`
- On constate que étant données deux listes `l1` et `l2`, le nombre de comparaisons dans `fusion l1 l2` est au plus la longueur de $l1$ + longueur de $l2$.

2.3.2 Cas où la longueur de la liste est une puissance de deux

Notons, pour tout $n \in \mathbb{N}$, C_n le nombre *maximal* de comparaisons dans l'appel de `tri` sur une liste de longueur *au plus* l. On constate alors que pour tout $n \in \mathbb{N}$:

$$C_n \leqslant n + C_{\lfloor \frac{n}{2} \rfloor} + C_{n-\lfloor \frac{n}{2} \rfloor}$$

N.B. Relation de récurrence typique lors de l'étude de la complexité d'un algorithme de type « diviser pour régner ».

La relation est plus simple si n est une puissance de 2. En effet pour tout $k \in \mathbb{N}$:

$$C_{2^k} \leqslant 2^k + 2C_{2^{k-1}}$$

Et en divisant par 2^k :

$$\frac{C_{2^k}}{2^k} \leqslant 1 + \frac{C_{2^{k-1}}}{2^{k-1}}$$

Et la suite $\left(\frac{C_{2^k}}{2^k}\right)_{n\in\mathbb{N}}$ est arithmétique de raison 1. Son premier terme est $\frac{C_1}{1} = 0$ (le cas d'une liste de longueur 1 est un des cas de base), on obtient $\forall k \in \mathbb{N}$, $\frac{C_{2^k}}{2^k} \leqslant k$, donc :

$$C_{2^k} \leqslant k2^k.$$

Remarque : En gros, si on pouvait remplacer ci-dessus k par $\log_2(n)$, on obtiendrait :

$$C_n \leqslant n \log_2(n)$$

Mais bien sûr, $\log_2(n)$ n'est en général pas entier !

2.3.3 Cas général

Pour faire le calcul propre, nous allons encadrer n entre deux puissances de 2 : Soit $k \in \mathbb{N}$ tel que :

$$2^k \leq n < 2^{k+1} \tag{1}$$

En fait, ceci revient à prendre $k = \lfloor \log_2(n) \rfloor$.

Lemme 2.2. *La suite C est croissante.*

Démonstration : C'est du au fait qu'on a défini C_n comme étant la complexité maximale pour des listes de longueur **au plus** n. □

En utilisant le lemme, il vient :

$$C_{2^k} \leq C_n \leq C_{2^{k+1}}$$
$$\text{donc } C_n \leq (k+1)2^{k+1}$$

Or, d'après 1, on déduit $k \leq \log_2(n)$ donc :

$$C_n \leq (\log_2(n)+1)2^{\log_2(n)+1}$$
$$\leq C_n \leq 2n(\log_2(n)+1)$$

D'où on tire $C_n = O_{n\to\infty}(n\log(n))$.

Ainsi, le tri fusion a une complexité de l'ordre de $n\log(n)$. En particulier, cette complexité est négligeable devant les méthodes naïves du tri bulle, par insertion, ou par extraction.

En outre, le tri fusion est finalement très simple à programmer. Il s'agit donc d'un excellent exemple d'utilité de la récursivité, et du principe « diviser pour régner » en particulier.

Remarque : On peut démontrer que $O(n\log(n))$ est la complexité optimale pour le tri d'une liste de longueur n en utilisant des comparaisons deux à deux d'éléments de la liste. Voir le chapitre sur les arbres.

Deuxième partie

Exercices

Exercices : diviser pour régner et tris

1 Diviser pour régner

Exercice 1. ** Factorielle

On propose une méthode de type « diviser pour régner » pour calculer une factorielle : il s'agit d'écrire une fonction auxiliaire `produit_intervalle` prenant deux entier a et b et renvoyant $\prod_{k=a}^{b-1} k$.

1. Programmer cette méthode.
2. Démontrer que pour tout $(a, b) \in \mathbb{N}^2$ tel que $a \leqslant b$, `produit_intervalle a b` nécessite $b-a-1$ multiplications.
3. Quel est l'intérêt de cette méthode en comparaison à la méthode naïve ?
4. Utiliser la fonction `produit_intervalle` pour calculer des coefficients binomiaux.

Exercice 2. ** Méthode des rectangles à pas variable

1. *Méthode des rectangles normale, c'est-à-dire à pas fixé :* Écrire une fonction récursive `rectangle f a b h` calculant une valeur approchée de $\int_a^b f$.

   On utilisera le principe suivant : si $b-a \leq h$ on approche f par une fonction constante. Sinon on découpe $[a, b]$ en deux, et on recommence sur chacun des deux intervalles ainsi créés.

   Comprenez-vous pourquoi cette méthode est appelé « méthode des rectangles » ?
2. *Méthode des rectangles à pas variable :* À présent, on suppose que f est monotone. On désire écrire une fonction prenant un flottant $eps > 0$ en argument, et calculant une valeur approchée de $\int_a^b f$ précise à $\varepsilon \times (b-a)$ près. Autrement dit, on s'autorise une erreur proportionnelle à la longueur de l'intervalle.

   On reprend le même principe que pour la méthode des rectangles à pas fixé, sauf que l'on s'arrêtera lorsque la précision voulue est obtenue et non lorsque le pas voulu est atteint.

   (a) Montrer que $\left|\int_a^b f - f(a)\cdot(b-a)\right| \leq |f(b)-f(a)| \times (b-a)$. En déduire le cas d'arrêt de la fonction.

   N.B. Le parenthésage est ainsi : $\left|\left(\int_a^b f\right) - f(a)\cdot(b-a)\right|$.

   (b) Programmer une fonction `rectanglePasVariable f a b eps` correspondante.

   (c) On suppose que f est de classe $\mathcal{C}^1$. Justifier que la fonction termine.

   (d) *Amélioration :* on remarque que certaines valeurs de f sont calculées deux fois. Proposer une amélioration pour éviter cette perte de temps. On pourra créer une fonction auxiliaire `aux f a b eps fa` prenant des arguments supplémentaires : fa qui contiendra $f(a)$ ou fb qui contiendra $f(b)$.

2 Tri

Exercice 3. * Liste triée

Écrire une fonction qui vérifie si une liste est triée dans l'ordre croissant.

Exercice 4. * Utilisation de listes triées

1. Écrire les fonctions suivantes pour qu'elles utilisent des listes triées. Calculer leur complexité.

   (a) Tester si un élément est dans une liste triée. *Remarque :* Ici l'utilisation d'un tableau serait plus efficace. Pourquoi ?

   (b) Enlever les doublons d'une liste triée.

   (c) Étant donnés une liste triée l et un élément x, renvoyer la liste des éléments de l inférieurs à x.

   (d) Étant donnés une liste triée l et un élément x, renvoyer deux listes, l'une contenant les éléments de l inférieurs à x, l'autre les éléments strictement supérieurs à x.

   On ne parcourra l qu'une seule fois.

   (e) Rechercher la médiane dans une liste triée.
2. On veut maintenant réaliser les fonctions précédentes pour des listes non triées. Dans quels cas est-il préférable de commencer par les trier pour pouvoir appliquer les fonctions précédentes ?

Exercice 5. **! Manipulation de listes strictement croissantes

Dans cet exercice, les fonctions prendront et renverront des listes strictement croissantes, donc en particulier sans doublon.

Ces fonctions seront utiles l'an prochain : les enregistrer dans un fichier à part, et ranger celui-ci dans un répertoire facile à retrouver.

1. Écrire une fonction `fusion_stricte` qui prend deux listes strictement croissantes `l1` et `l2` et renvoie la liste strictement croissante contenant les éléments de $l_1 \cup l_2$.
2. En déduire une fonction `tri_strict` qui prend une liste `l` et renvoie une liste triée et sans doublon contenant les éléments de `l`.
3. Écrire également une fonction `intersection` pour calculer l'intersection de deux listes strictement croissantes.
4. Écrire un prédicat `intersection_non_vide` pour indiquer si deux listes strictement croissances ont un élément en commun.
5. Enfin, écrire une fonction `a_plat` de type `'a list list ->'a list` qui prend une liste de listes strictement croissantes et qui renvoie la fusion stricte de toutes ces listes.
6. Donner la complexité de ces fonctions, et comparer avec leur analogue pour des listes quelconques.
7. On souhaite calculer l'intersection de deux listes pas forcément croissantes de même longueur. Est-il judicieux de commencer par les trier ?

Exercice 6. **! Tri par insertion

Le tri par insertion est le tri du joueur de cartes. On prend une liste vide et on y insère les éléments de la liste à trier l'un après l'autre, en prenant garde de toujours maintenir une liste triée.

Programmer cette méthode, et calculer sa complexité au pire. Calculer aussi sa complexité dans le meilleur des cas.

Exercice 7. ** Tri par extraction

1. Écrire une fonction `extraitMin` prenant en entrée une liste `l` et renvoyant un couple `(m,reste)` où `m` est le minimum de `l` et `reste` la liste obtenue en retirant `m` de `l`. Si `m` apparaît plusieurs fois dans `l`, on ne le retirera qu'une seule fois.
2. En déduire une fonction de tri dont le principe est le suivant : on récupère le minimum de la liste, on le place en tête, puis on réitère récursivement sur la queue.
3. Étudier la complexité de ce tri, en terme de nombre de comparaisons effectuées.

Quelques indications

1 Attention au cas de base.

3 Si votre fonction renvoie `true` pour la liste `[1;2;0;1]` c'est que vous êtes tombé dans le piège.

4 Pour la médiane, commencer par écrire une fonction `element_d_indice` prenant un entier i et une liste l et renvoyant l'élément d'indice i dans l.

5
1. Presque la fonction `fusion` du cours ; il faut juste mettre un cas particulier lorsque les deux éléments extraits des deux listes sont égaux.
2. La même qu'en cours !
3. Partir de la structure de `fusion_stricte`.
4. On peut la faire à l'aide d'un `List.fold_left`.

6 Écrire une première fonction `insertion` dont le but est d'insérer un élément dans une liste triée.

7 1) Comme la fonction renvoie un couple, il faudra passer par un `let (m,reste) = extraitMin q in` pour l'appel récursif.

Quelques solutions

1 1.

2. Récurrence forte

3. Il y a autant de multiplications avec les deux méthodes, mais les multiplications concernent des entiers plus petits avec la méthode diviser pour régner.

2 1.

```
let rec rectangles f a b h=
    (* Valeur approchée de \int_a^b f(t)dt obtenue par la méthode des rectangles à
       ↪ gauche en découpant en intervalles de longueur < h *)

    if b-.a < h then f a*. (b-.a) (* Remplacer f a par f b pour rectangles droite, ou
       ↪ (f a +. f b)/.2. pour trapèzes) *)
    else
        let m=(a+. b)/.2. in
        rectangles f a m h +. rectangles f m b h
    ;;
    (* Précision en O(h) *)

4.*.rectangles (fun x -> 1./.(1.+.x*.x)) 0. 1. 0.0001;;
```

2. Chaque appel qui n'est pas dans le cas d'arrêt conduit à deux appels récursifs. Cependant, tous les appels se font sur des intervalles différents, il n'y a dont pas d'appel redondant.

3. **Rectangles à pas variable :**

   (a) Il faut juste remarquer que $f(a)\cdot(b-a) = \int_a^b f(a)\,\mathrm{d}t$. Ainsi :

$$\begin{aligned}\left|\int_a^b f - f(a)\cdot(b-a)\right| &= \left|\int_a^b \big(f(t)-f(a)\big)\,\mathrm{d}t\right| \\ &\leqslant \int_a^b \big|f(t)-f(a)\big|\,\mathrm{d}t\end{aligned}$$

   Ensuite, pour tout $t\in[a,b]$, $|f(t)-f(a)| \leqslant |f(b)-f(a)|$ (traiter les deux cas f croissante et f décroissante). D'où :

$$\begin{aligned}\left|\int_a^b f - f(a)\cdot(b-a)\right| &\leqslant \int_a^b \big|f(b)-f(a)\big|\,\mathrm{d}t \\ &= \big|f(b)-f(a)\big| \times (b-a)\end{aligned}$$

   Ainsi, on peut prendre comme cas d'arrêt le cas où $|f(b)-f(a)| \leqslant h$.

   (b)

```
let rec rectanglesPasVariable f a b eps =
    if abs_float (f b -. f a) < eps then f a*. (b-.a) (* Remplacer f(a) par
       ↪ f(b) pour rectangles droite, ou (f(a)+.f(b))/.2. pour trapèzes) *)
    else
        let m=(a+. b)/.2. in
        rectanglesPasVariable f a m eps +. rectanglesPasVariable f m b eps
;;

4.*.rectanglesPasVariable (fun x -> 1./.(1.+.x*.x)) 0. 1. 0.0001;;
```

   (c) La fonction f est de classe $\mathcal{C}^1$ sur le *segment* $[a,b]$, elle est donc lipschitzienne. Soit k tel que f est k-lipschitzienne. Alors si $(b-a) \leqslant \frac{\varepsilon}{k}$, on a $|f(b)-f(a)| \leqslant k|b-a| \leqslant \varepsilon$, et donc l'appel `rectanglesPasVariable f a b eps` termine.

   Dans le cas général, comme à chaque appel récursif, la valeur $b-a$ est divisée par deux, elle finit toujours par devenir inférieure à $\frac{\varepsilon}{k}$, et donc la fonction termine.

   (d)

```
let rec rectanglesAux f a b eps fa fb =
    (* Arguments supplémentaires : f(a) et f(b) *)
    if abs_float (fb -. fa) < eps then fa*. (b-.a)
    else
        let m=(a+. b)/.2. in
    let fm=f m in
    rectanglesAux f a m eps fa fm +. rectanglesAux f m b eps fm fb
;;
```

```
9
10      let rectanglesFinale f a b eps =
11          rectanglesAux f a b eps (f a) (f b)
12      ;;
13
14      4.*.rectanglesFinale (fun x -> 1./.(1.+.x*.x)) 0. 1. 0.0001;;
```

3

4

6

```
1       let rec insertion x l=
2       (* Précondition : l est triée *)
3       (* Renvoie la liste obtenue en insérant x à sa place dans l *)
4       match l with
5       |[] -> [x]
6       |t::q when x < t -> x::l
7       |t::q -> t:: insertion x q
8       ;;
9       (* Cxté en O(|l|) *)
10
11      insertion 2 [1;2;8];;
12
13      let rec triInsertion l=
14      match l with
15      |[] -> []
16      |t::q -> insertion t (triInsertion q)
17      ;;
18      (* Cxté en O(|l|^2) *)
19      triInsertion [5;6;8;1;0];;
```

- Pour tout x et toute liste l, la complexité de `insertion x l` est en $O(|l|)$.
- Pour tout $n \in \mathbb{N}$, notons C_n le nombre maximal de comparaisons entre éléments de liste lors du tri d'une liste de longueus au plus n. On constate que $\begin{cases} C_0 = 0 \\ \forall n \in \mathbb{N},\ C_n \leqslant (n-1) + C_{n-1} \end{cases}$.
  On en déduit classiquement que $C_n = O_{n\to\infty}\left(()\, n^2\right)$.

7