Diviser pour Régner : Complexité et Tri Fusion

1 Notion de Complexité

Nous allons étudier la *complexité* des algorithmes étudiés. Il s'agit, en général, d'estimer le nombre d'opérations « élémentaires » (dont la définition peut varier selon le problème) en fonction de la taille de l'entrée de l'algorithme. Nous nous restreignons à la complexité *au pire cas*, c'est-à-dire que nous voulons une borne supérieure du nombre d'opérations. D'autres analyses de complexité existent, par exemple la complexité *en moyenne* sur des entrées aléatoires.
Par exemple, pour étudier la complexité d'un algorithme de tri sur des listes, nous nous intéresserons au nombre d'appels récursifs de la fonction de tri, ou au nombre de comparaisons effectuées (qui peuvent être coûteuses selon ce que contient la liste).

1.1 Notations

Étant donné deux fonctions f et g dans $\mathbb{N}$ ou $\mathbb{R}$, nous dirons que $f(x) = O(g(x))$ $(x \to \infty)$ s'il existe M, N tels que $\forall x > M, |f(x)| \leq N|g(x)|$ (notation de Landau [1]). Intuitivement, cela signifie que f ne croît pas plus vite que g, on dira que g *domine* f. On écrira $f(x) = \Theta(g(x))$ s'il existe M, N_1 et N_2 tels que $\forall x > M, N_1|g(x)| \leq |f(x)| \leq N_2|g(x)|$. Nous écrirons le plus souvent $O(n)$ avec n la taille de l'entrée car le plus souvent il suffit de se placer dans $\mathbb{N}$.
En général, si une fonction f est la somme de $f_1, \ldots, f_n$ alors la complexité de f est le maximum des complexités de $f_1, \ldots, f_n$.
La notion de taille d'entrée sera le plus souvent le nombre d'éléments pour une liste, un tableau ou une matrice, ou le nombre de bits pour un entier (i.e., $log_2(n)$). Un certain nombre de fonctions apparaissent souvent lors des études de complexité, et forment une hiérarchie (il va de soi qu'une complexité plus faible est préférable) :

1. Le O signifie "ordre" de la fonction.

Notation	Complexité...
$O(1)$	constante (pour toute entrée)
$O(ln(n))$	logarithmique
$O(n)$	linéaire
$O(n \cdot ln(n))$	quasi-linéaire
$O(n^2)$	quadratique
$O(n^c)(c > 1)$	polynômiale
$O(e^n)$	exponentielle

1.2 Exemples Simples

Étudions la complexité au pire cas de quelques algorithmes que nous avons déjà rencontrés.

Recherche dans une liste

Pour chercher un élément dans une liste (ou un tableau), le plus simple est de la traverser récursivement. Étudions la fonction `mem : int -> int list -> bool` qui renvoie `true` ssi l'élément concerné est dans la liste :

```
let rec mem x l = match l with
    | [] -> false
    | y :: _ when x=y -> true
    | _ :: l' -> mem x l';;
```

Cette fonction est de complexité $O(n)$ au pire cas (avec n la longueur de la liste, et en considérant le nombre d'appels récursifs ou le nombre de tests d'égalité). Cela se démontre facilement, car si on cherche un élément dans une liste à laquelle il n'appartient pas, il faut la traverser en entier (n appels récursifs donc).

Tri par Insertion

Le tri par insertion est un algorithme de tri sur les listes très simple. Il s'agit de parcourir une liste non triée, en ajoutant chaque élément dans une seconde liste que l'on maintient triée.

```
let rec insert x l = match l with
    | [] -> [x]
    | y :: _ when x <= y -> x :: l
    | y :: l' -> y :: insert x l' ;;

let tri_insertion l =
    let rec aux acc l = match l with
        | [] -> acc
        | x :: l' -> aux (insert x acc) l'
    in aux [] l;;
```

```
(* ou, de maniere equivalente : *)
let tri_insertion2 l =
    fold_left (fun acc x -> insert x acc) [] l;;
```

L'analyse se fait en deux temps. D'abord, remarquons que la fonction `insert` peut prendre n opérations pour insérer un élément x dans une liste l de longueur n (si $\forall y \in l,\ x > y$). Le pire cas pour notre implémentation est lorsque la liste à trier est déjà triée ; en effet, la complexité est $O(n^2)$ dans ce cas.
Montrons par récurrence sur n qu'une liste $l_n = [1; 2; \ldots; n]$ requiert $\sum_{k=0}^{n-1} k$ appels récursifs. Le cas de base est trivial ; supposons que ce soit le cas pour l_n et montrons-le pour l_{n+1}. Par définition le tri des n premiers éléments est similaire, on se retrouve avec un accumulateur qui contient n éléments triés, dans lequel on va insérer l'élément $n+1$. Cela demande $n+1$ appels récursifs par la remarque précédente sur la fonction `insert`. Par conséquent le nombre total d'appels est $\sum_{k=0}^{n-1} k + n = \sum_{k=0}^{n} k$. Puisque $\sum_{k=0}^{n-1} k = \dfrac{n(n-1)}{2} = \Theta(n^2)$, la complexité est bien quadratique.

2 Diviser pour Régner

La complexité des algorithmes précédents peut être améliorée dans certains cas en utilisant le principe dit *diviser pour régner*[2]. Le principe de base est de diviser un gros problème en un ou plusieurs plusieurs sous-problème de taille plus réduite qu'on peut traiter indépendamment de la même manière, récursivement. Une seconde phase où les résultats de ces traitements récursifs sont collectés peut être nécessaire. La clé de cette technique repose sur cette subdivision récursive en problèmes plus petits.
Un exemple très simple consiste à deviner le nombre, compris entre 0 et 100, qu'un ami a en tête, par une suite de comparaisons. Il est logique de commencer par comparer à 50, puis à 25 ou 75 selon que le nombre est plus petit ou plus grand que 50, et ainsi de suite.
Nous allons nous pencher sur deux algorithmes très classiques qui relèvent de ce paradigme : la recherche par dichotomie (dans un tableau) et le tri-fusion. D'autres algorithmes, tels que la multiplication de Karatsuba, l'exponentiation rapide ou la recherche de plus proches points dans le plan, relèvent aussi de cette technique.

2.1 Rechercher Dichotomique dans un Tableau

Nous avons vu que la recherche d'un élément dans une Íliste était de complexité linéaire. Il en est de même pour un tableau ; dans le cas où ce tableau est trié on peut toutefois faire mieux. Le problème à résoudre, plus précisément, consiste à rechercher un élément x dans un tableau v de longueur n, trié par ordre croissant, et, s'il est présent, de retourner son indice dans le

2. en anglais, *divide and conquer*. On remarquera l'impérialisme expansionniste de la perfide Albion.

tableau. Voici la version naïve et la version par dichotomie, toutes deux de type `'a vect -> 'a -> int option` (elles retournent `None` si l'élément n'est pas présent dans le tableau) :

```
let search_naive v x =
    let rec aux i =
        if i = vect_length v then None
        else if v.(i) = x then Some i
        else aux (i+1)
    in aux 0
;;

let search_good v x =
    let i = ref 0 and j = ref (vect_length v - 1) in
    let result = ref None in
    while !i <= !j && !result = None do
        let middle = (!i + !j) / 2 in
        if v.(middle) = x
            then result := Some middle
        else if v.(middle) < x
            then i := middle + 1
        else j := middle - 1
    done;
    ! result
;;
```

Analysons maintenant le second algorithme. Il s'agit de rechercher, dans le tableau v, la valeur x. Cependant, nous ne cherchons x que dans l'intervalle des indices $[i, i+1, \ldots, j]$ (avec $i \leq j$ comme l'indique la condition de boucle). Intuitivement, nous procédons de la même manière que pour chercher un nom dans un annuaire. À chaque itération, nous comparons $v.[\lfloor \frac{i+j}{2} \rfloor]$ et x (rappelons que $\lfloor . \rfloor$ désigne la partie entière) :

– si $x = v.[\lceil \frac{i+j}{2} \rceil]$ l'algorithme s'arrête avec succès ;

– si $x < v.[\lceil \frac{i+j}{2} \rceil]$ alors x ne peut être que dans la partie de v entre i et $\lfloor \frac{i+j}{2} \rfloor - 1$;

– sinon x, s'il est présent dans v, doit être entre $\lfloor \frac{i+j}{2} \rfloor + 1$ et j.

.

Cet algorithme termine car à chaque itération qui ne termine pas directement (c'est-à-dire en excluant le cas où on trouve x), l'intervalle $j - i$ décroit strictement (et reste positif car $i \leq j$ est obligatoire pour continuer à boucler). Notons $T(n)$ le nombre d'itérations de l'algorithme sur une entrée de taille n au pire cas. On a :

$$T(n) = \begin{cases} T(\lceil \frac{n}{2} \rceil) + 1 & \text{si } n \geq 2 \\ 1 & \text{sinon} \end{cases}$$

On peut donc dominer la complexité sur une entrée de taille n par celle sur une entrée de taille $2^{\lceil log_2(n) \rceil}$. En ce ramenant à ce dernier cas (tableau dont la taille est une puissance de 2), on prouve par récurrence que la complexité de cet algorithme de $O(ln(n))$ (plus précisément $O(log_2(n))$, ce qui est équivalent). En effet, la suite $(T(2^n))_{n \in \mathbb{N}}$ est trivialement égale à la suite $1, 2, 3, \ldots n$ par récurrence, donc $T(2^n) = O(n)$, et donc $T(n) = O(log_2(n))$ pour n puissance de 2. Il existe des théorèmes plus puissants permettant d'analyser facilement la complexité des algorithmes *diviser pour régner*.

2.2 Tri Fusion

Le tri fusion est un tri optimal sur les listes, de complexité $O(n \cdot ln(n))$. Il s'agit de décomposer une liste en deux sous-listes chacune deux fois plus petites, de les trier séparément, puis de fusionner les résultats en une liste triée.
Commençons par l'opération de fusion. La fonction `merge` prend en argument deux listes triées et les fusionne en une liste triée, sans enlever les doublons (la longueur du résultat est donc la somme des longueurs des entrées). Cette fonction termine car au moins un de ses arguments décroit strictement, et l'autre décroit ou reste identique [3]. On peut également démontrer par induction que si `l1` et `l2` sont triées, alors `merge l1 l2` est également triée et contient tous les éléments de `l1` et `l2` avec la bonne multiplicité.

```
let rec merge l1 l2 = match l1, l2 with
    | [], _ -> l2
    | _, [] -> l1
    | x::l1', y::l2' ->
        if x < y then x :: merge l1' l2
        else if x > y then y :: merge l1 l2'
        else x :: y :: merge l1' l2'
;;
```

Il nous faut ensuite une fonction `split` qui décompose une liste en deux listes de longueurs à peu près égales [4]. Plusieurs techniques fonctionnent, mais nous allons en choisir une qui choisit alternativement un élément sur deux pour chaque liste. Cette fonction va diviser le problème en sous-problèmes deux fois plus petits.

```
let split l =
    let rec split_aux acc1 acc2 l = match l with
        | [] -> acc1, acc2
        | x::l' -> split_aux acc2 (x::acc1) l'
    in split_aux [] [] l;;
```

Enfin, la fonction principale, `merge_sort`, trie trivialement les listes de taille 0 ou 1, et traite les autres cas par récursion :

3. On peut utiliser l'ordre sur les multi-ensembles de listes, pour obtenir rigoureusement un ordre bien fondé sur les arguments ; le multi-ensemble $\{l_1, l_2\}$ décroit strictement à chaque appel récursif.

4. Si la liste est de longueur impaire, une des sous-listes doit être plus petite que l'autre.

```
let rec merge_sort l = match l with
    | [] -> []
    | [x] -> [x]
    | _ ->
        let l1, l2 = split l in
        let l1' = merge_sort l1 in
        let l2' = merge_sort l2 in
        merge l1' l2'
;;
```

Avec les mêmes notations, après avoir facilement prouvé que la complexité de `merge` et `split` était $O(n)$, on exprime la complexité au pire cas de `merge_sort` par

$$T(n) = \begin{cases} 2 \cdot T(\frac{n}{2}) + 2 \cdot n & \text{si } n \geq 2 \\ 1 & \text{sinon} \end{cases}$$

Pour les puissances de 2, on a $T(2^n) = 2 \cdot T(2^{n-1}) + 2 \cdot 2^n$ et on va montrer par récurrence sur n que $T(2^n) = n \cdot 2^{n+1}$:

$$\begin{aligned} T(2^n) &= 2 \cdot 2^n + 2 \cdot T(2^{n-1}) \\ &= 2 \cdot 2^n + 2 \cdot (n-1) \cdot 2^n \\ &= 2 \cdot 2^n + (n-1) \cdot 2^{n+1} \\ &= 2^{n+1} + (n-1) \cdot 2^{n+1} \\ &= n \cdot 2^{n+1} \end{aligned}$$

Par conséquent, $T(2^n) = log_2(2^n) \cdot 2^{n+1}$, et en majorant $T(n)$ par $T(2^{\lceil log_2(n) \rceil})$ on obtient $T(n) = \Theta(\lceil (log_2(n)) \rceil \cdot 2^{\lceil log_2(n) \rceil + 1}) = O(n \cdot ln(n))$ (tous les logarithmes étant équivalents en complexité, à constante multiplicative près).
Cet algorithme est donc plus efficace asymptotiquement que le tri par insertion qui est quadratique au pire cas.