Rapport
Projet : programmation fonctionnelle

Quentin Januel & Clément Defrétiére

6 avril 2018

1 Analyse naïve des algorithmes

2 Analyse de complexité

Nous allons à présent analyser le temps que prend chaque fonction de tri en fonction de la longueur de la liste qu'on lui passe en argument, en nous basant directement sur le code source de chacune des trois fonctions de tri.

Pour ce faire, nous allons d'abord considérer que ce temps est proportionnel au nombre de calculs que doit faire tel algorithme en fonction de la longueur de la liste.

Ainsi, pour une liste de taille N, nous recherchons l'ordre de grandeur en fonction de N du nombre d'opérations que va effectuer l'algorithme.

Par soucis de simplicité, nous allons considérer que les fonctions comme `List.length` ou `@` sont instantannées ou au moins proportionnelles à la longeur des listes avec lesquelles on les utilise.

Afin de représenter la complexité d'un algorithme, nous allons utiliser la notation O.
Ainsi, un algorithme ayant une complexité $O(n^2)$ sera un algorithme qui, pour une liste de taille n, effectue $n \times n$ opérations (et prend donc un temps relativement proportionnel).

Pour finir, nous n'allons pas prendre en compte les opérations ne dépendant pas de n, ni les facteurs multiplicatifs de n puisque ces derniers n'auront aucun impact quand n sera suffisamment grand.
Une complexité $O(100 + 5 \times n)$ est donc la même que $O(n)$.

2.1 Tri par création du maximum

Voici le code source de la fonction de tri par création du maximum (ou tri par selection) :

```
let rec tri l =
    if l = [] then [] else
    let max = selectionne_max comp l in
        let subL = supprime max l in
            let sorted = tri comp subL in
                sorted@[max]
```

Afin de selectionner l'élément maximum de la liste, `selectionne_max` doit faire $n-1$ comparaisons.
Ensuite la fonction fait une récurrence sur elle même avec une sous liste de longueur $n-1$, jusqu'à se retrouver avec une liste vide.

Le nombre total de calculs sera donc de

$$(n-1)+(n-2)+...+1=\sum_{k=1}^{n-1}k=\frac{n\times(n-1)}{2}$$

La complexité de l'algorithme est $O(n^2)$.

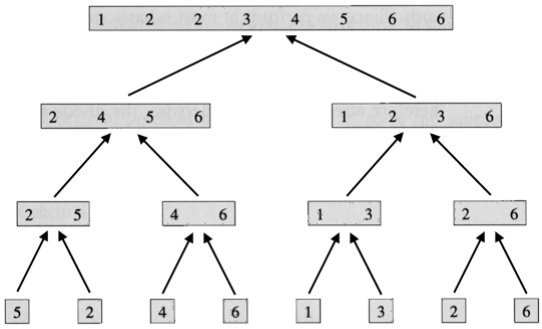
2.2 Tri par fusion

Voici le code source de la fonction de tri par fusion :

```
let rec tri comp l =
    let l1, l2 = partitionne l in
        if (List.length l1)+(List.length l2) < 2 then
            l1@l2
        else
            let sorted1 = tri comp l1
            and sorted2 = tri l2 in
                fusionne comp sorted1 sorted2
```

Nous pouvons constater que cette fonction fait elle même appel à deux fonctions ; `partitionne` et `fusionne`. Ces dernières ont toutes deux une complexité $O(n)$.

A chaque étape, cet algorithme coupe la liste en deux, pour ensuite les fusionner.
Nous pouvons le représenter par le diagramme ci dessous :



Appelons niveau i de l'algorithme la i^{eme} ligne du schéma.

A chaque niveau, la liste est coupé en p parties et la longueur de chaque sous liste est de n/p. Il y aura donc exactement n calculs par niveau (puisque, rappelons le, `partitionne` et `fusionne` sont de complexité $O(n)$).

La complexité de cet algorithme est donc $O(n \times \#niveaux)$.
Le nombre de niveaux est le nombre de fois qu'il faut couper une liste en deux jusqu'à n'obtenir qu'un seul élément.

Supposons que n soit de la forme 2^a, il y aura donc a niveaux.
Plus généralement, pour une liste de longueur n, il y aura approximativement $log_2(n) = \log(n)/\log(2)$ niveaux.

Cet algorithme a donc une complexité $O(n \times \log(n))$.

2.3 Tri par arbre binaire de recherche

Voici le code source de la fonction de tri par arbre :

```
let tri comp l = parcours_arbre (insere_liste_noeuds comp l
    ArbreVide)
```

Afin de déterminer sa complexité, il nous faut donc celle de `parcours_arbre` et de `insere_liste_noeuds`.
Il est suffisamment évident que la première est de complexité $O(n)$.

Quant à la seconde, cela dépend de la liste à placer dans l'arbre.
Si cette liste est déjà triée (ce qui est le pire scénario), chaque élément va être placé à droite et l'arbre n'aura donc qu'une seule branche et une longueur de n.

En d'autres thermes, placer le premier élément prendra 1 calcul, le second 2, et ainsi de suite. Le nombre total de calculs sera donc

$$1 + 2 + 3 + ... + n = \sum_{k=1}^{n} = \frac{n \times (n+1)}{2}$$

Ce qui signifie que cet algorithme a une complexité de $O(n^2)$.

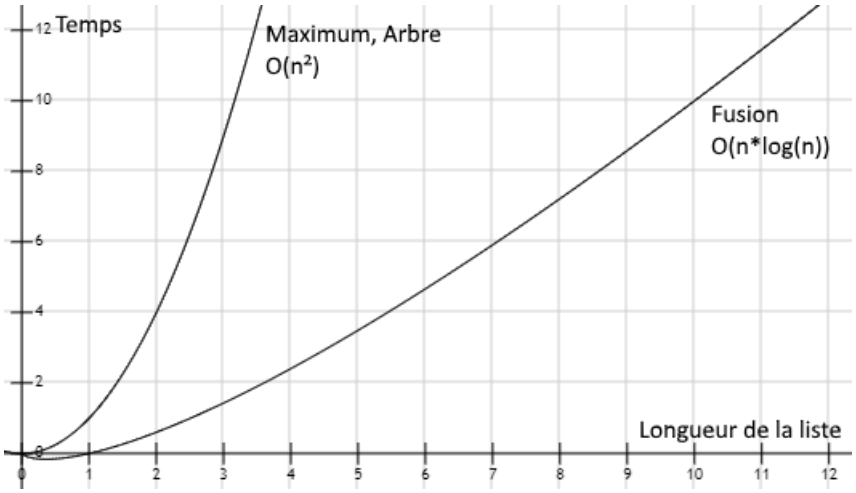
Il est toutefois important de noter qu'en moyenne, l'arbre s'équilibre et l'algorithme a donc une complexité de seulement $O(n \times \log(n))$ (en procédant avec le même raisonnement que pour l'algorithme précédent).

Ouverture :

L'arbre utilisé n'est pas équilibré, mais il aurait été possible de faire l'algorithme avec un arbre équilibré.
Ainsi, nous n'aurions jamais eu une seule branche et la complexité aurait été de $O(n \times \log(n))$.

3 Comparaison des efficacités

Nous pouvons grapher la complexité des différents algorithmes afin de comparer leurs croissances.



En prenant pour critère le temps, le meilleur algorithme est donc le tri par fusion.

Il est important de noter que le tri par arbre n'est pas aussi mauvais que le tri par maximum en général, ce graphique prend en considération le pire scénario.

Ouverture :

Nous pourrions comparer les algorithmes en fonction de l'espace mémoire qui leur est nécessaire.
Par exemple, le tri par maximum n'a besoin d'aucune mémoire supplémentaire, mais le tri par arbre crée un arbre temporaire de n éléments.
L'espace mémoire requis pour le tri par maximum en donc en $O(1)$, et $O(n)$ pour le tri par arbre.