Complexité algorithmique

Florent Bouchez Tichadou

5 septembre 2019

L'algorithmique est la science qui s'intéresse non seulement à l'écriture des algorithmes, mais également à leur étude et analyse. Dans ce document, nous abordons la notion de *complexité algorithmique*, qui est une mesure de l'« efficacité » d'un algorithme. Nous nous intéressons donc non seulement à l'écriture d'algorithmes qui produisent des résultats corrects, mais également à la vitesse à laquelle ils résolvent le problème.

Note importante : contrairement à ce que le nom suggère, la complexité *n'est pas* une mesure de si un algorithme est « simple » ou « complexe » d'un point de vue humain. C'est en fait bien souvent l'inverse : un algorithme simple aura généralement une complexité plus élevée (il « prend plus de temps ») qu'un algorithme ingénieux, qui aura une faible complexité (plus « rapide »).

1 Introduction à la complexité algorithmique

La *complexité d'un algorithmique* est une prédiction ou une garantie que l'algorithme ne prendra jamais plus qu'un certain nombre d'étapes ou opérations, qui dépend souvent de la *taille* des données qu'il manipule. On note en général n cette taille et on cherche « *formule(n)* » qui représente le nombre maximum d'opérations et dépend de l'algorithme.

Voyons un algorithme simple qui renvoie la somme de tous éléments d'un ensemble représenté dans un tableau avec longueur explicite.

```
somme(E)
    s ← 0
    pour i de 0 à E.longueur −1 faire
        s ← s + E.tab[i]
    retourner s
```

Analysons les différentes parties de cet algorithme :

- l'initialisation de s coûte 1 opération ;
- pour la boucle, le nombre d'opérations est la somme des opérations de chacune des itérations :
  - corps de boucle : 2 opérations : une addition puis le stockage du résultat dans s ;
  - maintenance de boucle : à chaque fois un incrément et un test sont cachés dans le « pour » : $i \leftarrow i + 1$ et $i \leqslant$ longueur -1 : 2 opérations ;
  - entrée dans la boucle : le premier test (1 opération, pas d'incrément au début) ;
- retour de la fonction : 1 opération.

Au final, chaque itération de la boucle exécute le même nombre d'instructions, le nombre total pour la boucle est donc « coût d'une itération » × « nombre d'itérations » ; la boucle est répétée « longueur » fois. La formule pour cet algorithme est $1 + 1 + n \times (2 + 2) + 1 = 3 + 4n$ où $n =$ longueur de l'ensemble.

Nous voyons ici que le temps d'exécution de cet algorithme croît linéairement en n. Quand nous étudions la complexité algorithmique, ce qui nous intéresse le plus est le comportement *asymptotique* de l'algorithme, c'est-à-dire la « forme » de la formule pour n grand. Nous gardons donc seulement le « plus gros » terme de la formule sans sa constante : nous utilisons la notation « grand O » des mathématiques, et disons que l'algorithme est en $O(n)$.

2 La notation grand O

Étant donnés deux fonctions f et g, $f = O(g)$ ssi pour tout n suffisamment grand, il existe une constante C telle que $f(n) \leqslant C \times g(n)$. Plus formellement :

$$\exists n_0 \geqslant 0, \exists C > 0 \; t.q. \; \forall n \geqslant n_0, f(n) \leqslant C \times g(n)$$

Cette notation nous permet de nous concentrer sur les « grandes » valeurs de n, sans se préoccuper des petites valeurs de n, pour lesquelles de toutes façons l'algorithme se terminera très rapidement sur un ordinateur moderne.

La constante C de la notation nous permet d'« oublier » la constante du plus grand facteur. L'analyse devient non seulement plus simple mais surtout possible car il est en pratique presque impossible de savoir exactement combien d'opérations sont nécessaires : cela dépend notamment du processeur où le code est exécuté. Par exemple dans le corps d'une boucle, ce qui nous importe est alors juste de savoir qu'il y a un nombre constant d'opérations, et si la boucle est répétée n fois on a alors $O(n)$ opérations.

Ces deux propriétés combinées nous permettent de nous concentrer sur le comportement pour des « grandes » instances, où il y a une énorme différence par exemple entre une complexité en $O(n)$ et une en $O(n^2)$, mais où des complexités qui seraient $\approx 4 \times n$ et $\approx 5 \times n$ sont considérées comme équivalentes.

Revenons à notre exemple pour en faire directement l'analyse avec la notation O. Pour mémoire, rappelons qu'un nombre constant d'opération est en $O(1)$. Nous avons maintenant les propriétés suivantes pour une analyse de la complexité en temps. Le détail est à gauche mais on général on annote directement sur l'algorithme comme montré ci-dessous à droite.

— initialisation : $O(1)$;
— boucle : n fois le corps de boucle ;
— corps de boucle : $O(1)$;
— finalisation : $O(1)$.

```
somme(E)
    s ← 0                                   O(1)
    pour i de 0 à E.longueur −1 faire       n ×
        s ← s + E.tab[i]                    O(1)
    retourner s                             O(1)
```

Au global, la complexité est $O(1) + n \times O(1) + O(1) = O(1) + O(n) = O(n)$.

3 Comparer les complexités

Le but de l'analyse de complexité est de pouvoir comparer plus facilement différents algorithmes qui effectuent la même tâche. On préférera par exemple l'algorithme qui s'exécute en $O(n)$ par rapport à celui en $O(n^2)$. Il devient également possible de prédire le temps que prendra un algorithme à trouver la solution sur une instance très grande en extrapolant une mesure de temps d'exécution faite sur une plus petite instance.

Dans cette section, nous nous intéressons à observer combien de fois plus de temps est nécessaire à un algorithme quand on double la taille des données d'entrée. Par exemple, un algorithme linéaire ($O(n)$) mettra deux fois plus de temps. On dénote $\log n$ le logarithme en base 2, qui fait partie des complexités courantes pour un algorithme (voir sections suivantes). Nous donnons ici les complexités dans l'ordre du plus courant au moins courant (sans être exhaustif, il existe bien sûr beaucoup d'autres complexités possibles). Nous donnons également une idée de la taille maximale que peuvent avoir les données (colonne « Max n ») avant que l'algorithme devienne impraticable (cela prendrait trop de temps pour résoudre le problème).

Complexité	Nom courant	Temps quand on double la taille de l'entrée	Max n
$O(n)$	linéaire	prend 2 fois plus de temps	10^{12}
$O(1)$	constant	prend le même temps	pas de limite
$O(n^2)$	quadratique	prend 4 fois plus de temps	10^6
$O(n^3)$	cubique	prend 8 fois plus de temps	$10\,000$
$O(\log n)$	logarithmique	prend seulement une étape de plus	$10^{10^{12}}$
$O(n \log n)$	linearithmique	prend deux fois plus de temps $+ \log n$	10^{11}
$O(2^n)$	exponentiel	prend tellement de temps que c'est inconcevable	30

On observe une séparation nette entre la complexité exponentielle et les autres, qu'on appelle *polynomiales*. Les algorithmes exponentiels sont si catastrophiques qu'il ne sert à rien de se demander « que se passe-t-il si je double la taille de mon entrée » car, à l'inverse, il suffit d'augmenter la taille du 1 pour *doubler* le temps de calcul ! Ces algorithmes doivent être évités à tout prix, sauf si l'on sait que les instances sont très petites.

4 Complexité des structures de boucles et conditions

4.1 Complexité des boucles imbriqués

Comme établi précédemment, pour une boucle on fait la somme des complexités de chaque itération. Dans le cas de boucles imbriquées, on calculera d'abord la complexité de la boucle interne car on en a besoin pour connaître le coût d'une itération de la boucle externe. De fait, souvent on va multiplier entre elles le nombre d'itérations de chacune des boucles.

Considérons l'algorithme ci-dessous qui affiche toutes les additions possibles de deux nombres entre 1 et n.

Chaque itération de la boucle sur j coûte $O(1)$ (une addition et un affichage). Cette boucle est répétée n fois soit $n \times O(1) = O(n)$. Donc chaque itération de la boucle sur i coûte $O(n)$. Il y a également n itérations de cette boucle donc au total $n \times O(n) = O(n^2)$.

```
toutes_additions(n)
    pour i de 1 à n faire                    n ×
        pour j de 1 à n faire                      n ×
            afficher(i + j)                              O(1)
```

On peut remarquer que cet algorithme affiche plusieurs fois chaque addition. En effet on aura par exemple $2 + 7$ (quand $i = 2$ et $j = 7$) puis plus tard $7 + 2$. Analysons l'algorithme ci-dessous qui ne répète pas les doublons en faisant s'arrêter j à la valeur i au lieu de n.

Cette fois, la boucle interne a un coût de $O(i)$. Le problème est que i change à chaque itération, donc on ne peut pas simplement multiplier le tout par n : la dernière itération coûte bien $O(n)$ mais la première seulement $O(1)$!

```
toutes_additions(n)
    pour i de 1 à n faire                    n ×
        pour j de 1 à i faire                      i ×
            afficher(i + j)                              O(1)
```

Dans ce cas, on peut cependant être conservateur. Comme $i \leqslant n$, on a $i = O(n)$ d'après la définition et on se retrouve comme dans le premier cas avec une complexité en $O(n^2)$.

Si l'on veut savoir plus précisément combien de fois la boucle interne est exécutée, il est possible de le calculer directement : à la première itération de la boucle externe, 1 fois, puis 2 fois, puis 3, etc. jusque n. Au total, cela fait $\sum_{k=1}^{n} k$ fois, soit, $n \times (n-1)/2$. La complexité est donc $n \times (n-1)/2 \times O(1) = O(n^2/2 - n/2) = O(n^2/2) = O(n^2)$. Cette analyse est plus précise mais ne change pourtant pas la complexité qui reste en $O(n^2)$.

4.2 Complexité des conditions

Toutes les parties d'un algorithme ne sont pas forcément exécutées, c'est notamment le cas lors qu'il y a des conditions « si... alors... sinon ». L'une des deux branches sera exécutée mais pas l'autre. Il faut alors analyser séparément les deux branches possibles et les comparer.

Plusieurs cas de figure sont possibles :

- complexités égales dans le *alors* et le *sinon* : la complexité globale est alors la même. Exemple : $O(1)$ et $O(1)$ donne $O(1)$:
- complexité plus importante dans une des deux branches : pour être conservateur, on garde alors la plus grande. Exemple $O(1)$ et $O(n)$ donne $O(n)$;
- complexité plus importante dans une des deux branches *mais celle-ci est moins souvent exécutée*. On peut aussi rester conservateur et garder le maximum, mais on aura une analyse moins précise. Nous allons étudier ci-dessous un exemple.

```
exemple (E)                                                    n ×
    pour i de 0 à E.longueur −1 faire                          |
        si i = 0 alors                                         |
            x ← E.tab[i]                                       |  O(1)
            pour j de 0 à E.longueur −1 faire                  |  n ×
                E.tab[j] ← x+E.tab[j]                          |      O(1)
        sinon                                                  |
            afficher (E.tab[i])                                |  O(1)
```

Dans cet exemple, on pourrait en première approximation dire que la complexité est en $O(n^2)$ à cause des boucles imbriquées. Ce n'est pas faux mais cela manque de précision car ne capture pas le fait que la boucle interne n'est exécutée qu'une seule fois, lorsque $i = 0$. Si l'on regarde le coût des itérations de la boucle sur i, la première itération coûte bien $O(n)$, cependant les suivantes ne coûtent que $O(1)$. La complexité totale est donc de $O(n) + (n-1) \times O(1) = O(n)$.

5 Exemples d'analyses de complexité

Nous savons déjà que de faire la somme des éléments d'un ensemble (ou une séquence) peut être fait en temps linéaire. De manière générale, le parcours d'un ensemble ou d'une séquence (i.e., itérer sur tous les éléments) peut être effectué en $O(n)$.

5.1 Ajout en début de séquence

Cette analyse dépend de la structure de donnée utilisée pour stocker la séquence.

liste chaînée : on ajoute une nouvelle cellule au début et on la lie à l'ancienne tête : $O(1)$;
tableau + longueur : on décale tous les éléments existant d'une case vers la droite : $O(n)$.

5.2 Ajout en fin de séquence

Cela dépend encore une fois de la structure de donnée utilisée.

liste chaînée : on parcourt toute la liste pour arriver en queue : $O(n)$;
tableau + longueur : on incrémente la longueur et écrit au dernier indice : $O(1)$.

5.3 Tri par sélection

Dans un tri par sélection, on cherche le maximum (ou minimum) de la séquence, puis on le place à la fin (resp. au début). On répète le même processus avec le reste de la sous-séquence qui est non-triée. Il est possible de faire un tri par sélection dans un tableau ou dans une liste chaînée.

5.3.1 Dans un tableau

```
tri_selection(S)                                               n ×
    pour dernier de S.longueur −1 à 1 faire                    |
        imax ← 0                                               |  O(1)
        max ← S.tab[0]                                         |  O(1)
        pour i de 1 à dernier faire                            |  O(n) ×
            si S.tab[i] > max alors                            |      O(1)
                imax ← i                                       |      O(1)
                max ← S.tab[i]                                 |      O(1)
        S.tab[imax] ← S.tab[dernier]                           |  O(1)
        S.tab[dernier] ← max                                   |  O(1)
```

L'algorithme utilise deux boucles imbriquées. La boucle extérieure a n itération, et celle intérieure v itération, avec v qui varie : $n-1$ au début mais diminue ensuite au cours du temps jusque 1. Nous savons que $v \leqslant n$ est toujours vrai, donc nous savons que le corps de la boucle interne est exécuté à $O(n)$ fois à chaque itération. Le corp boucle est en $O(1)$ (ainsi que le reste des calculs), donc la complexité globale est $O(n^2)$.

De même que vu précédemment, on pourrait faire une analyse plus précise qui nous conduirait à trouver qu'on exécute le corps de la boucle interne $n \times (n-1)/2$ fois, mais cela ne changerait pas la complexité.

5.3.2 Dans une liste chaînée

```
tri_selection(S)
    dernier ← Nil                                          O(1)
    tant que dernier ≠ S.tête faire                        n ×
        max ← S.tête.valeur                                    O(1)
        maxcel ← S.tête                                        O(1)
        maxpred ← Nil                                          O(1)
        cel ← maxcel.suivant                                   O(1)
        queue ← maxcel                                         O(1)
        tant que cel ≠ dernier faire                           O(n) ×
            si cel.valeur > max alors                                  O(1)
                max ← cel.valeur                                       O(1)
                maxcel ← cel                                           O(1)
                maxpred ← queue                                        O(1)
            queue ← cel                                                O(1)
            cel ← cel.suivant                                          O(1)
        si queue ≠ maxcel alors                                O(1)
            si maxpred = Nil alors
                S.tête ← maxcel.suivant                        O(1)
            sinon
                maxpred.suivant ← maxcel.suivant               O(1)
            maxcel.suivant ← dernier                           O(1)
            queue.suivant ← maxcel                             O(1)
```

Analyse de complexité : chaque boucle « tant que » sur la liste chaînée agit de manière similaire à la boucle « pour » correspondante dans la représentation par tableau. Comme précédemment, on ne connait pas exactement le nombre d'itérations de la boucle interne, mais celui-ci est borné par n donc $O(n)$ itérations, et au final la complexité est $O(n^2)$.

5.4 Recherche d'un élément dans une séquence

Considérons deux cas : selon que la séquence est triée ou non.

Séquence non triée Il nous faut vérifier chaque élément de la séquence : c'est un parcours classique, donc $O(n)$ pour un tableau ou une liste chaînée.

Séquence triée Pour une liste chaînée, cela nous permet de nous arrêter dès que l'on trouve un élément strictement plus grand que celui que l'on cherche. Cela ne change tout de même pas la complexité en général, car dans le pire des cas, il nous faut toujours vérifier jusqu'au dernier élément de la séquence. En moyenne on vérifiera la moitié de la séquence, ce qui est toujours du $O(n)$.

Pour un tableau, nous pouvons utiliser la *dichotomie* puisque nous avons directement accès au « milieu » de la séquence : en comparant avec l'élément du milieu, on n'a plus ensuite qu'à chercher dans une moitié de la séquence. Supposons que l'on cherche x, et que S.tab[longueur/2] contient v ; si $x < v$, alors x ne peut pas être à un indice plus grand que longueur/2. En utilisant cette propriété, on peut alors faire une *recherche binaire*, algorithme présenté à droite.

```
recherche (S, x)                                        /* S doit être trié */
    i ← 0                                               O(1)
    j ← S.longueur −1                                   O(1)
    tant que i⩽ j faire                                 k ×
        m ← (i + j) div 2                                       O(1)
        si x = S.tab[m] alors retourner m                       O(1)
        sinon si x < S.tab[m] alors j ← m − 1                   O(1)
        sinon i ← m + 1                                         O(1)
    retourner −1                                        /* non trouvé */
```

Quelle est la complexité de cet algorithme ? Nous voyons que les parties avant et après la boucle sont en $O(1)$, de même que le corps de boucle. La complexité va donc être directement liée au nombre d'itérations de cette boucle que l'on note k.

Considérons la quantité $j - i$: au début, elle est égale à S.longueur, et à chaque itération cette quantité va être divisée approximativement par 2. Dans le pire des cas (quand x n'est pas dans la séquence), ce processus est répété jusqu'à atteindre un, puis zéro, puis i devient plus grand que j (à cause du -1 ou du $+1$).

Nous devons donc répondre à la question suivante : combien de fois faut-il diviser n par 2 pour attendre 1 (on peut ne pas considérer les deux dernières étapes car on cherche une notation O) ? La réponse est $O(\log n)$ (le logarithme en *base 2*). En effet, on cherche le plus petit k tel que $n/2^k \leqslant 1$, c'est-à-dire, $n \leqslant 2^k$, et nous savons que $2^{\log n} = n$.

Une autre façon de voir (ou prédire) la complexité logarithmique est de remarquer que même si l'on double la taille de l'entrée (i.e., une séquence de taille $2n$ au lieu de n), on n'a besoin que d'une étape de plus pour effectuer la recherche...

En général, on retrouve une complexité logarithmique dans tous les algorithmes qui contiennent une boucle divisant une quantité de donnée par une constante à chaque itération.

5.5 Calcul des nombres de Fibonacci

Les nombres de Fibonacci sont définis ci-dessous à gauche. Un algorithme très naïf pour calculer le $n^{\text{ème}}$ nombre est proposé à droite.

$$\begin{aligned} F_0 &= 1 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

```
Fibo(n)
    si n < 2 alors
        retourner 1
    sinon
        retourner Fibo(n − 1) + Fibo(n − 2)
```

Il est plus difficile de calculer la complexité de cet algorithme car il est récursif : la complexité de Fibo dépend... de la complexité de Fibo !

Définissons alors C_n comme étant le nombre d'opérations nécessaire pour calculer Fibo(n). On peut écrire la formule suivante : $C_n = O(1) + C_{n-1} + C_{n-2}$. Remarquez que cette formule ressemble très fortement à l'équation de récurrence de la suite de Fibonacci elle-même !

Faisons maintenant une approximation. Puisque Fibo($n-1$) va elle-même appeler Fibo($n-2$), on sait que $C_{n-1} \geqslant O(1) + C_{n-2}$, donc on écrit $C_n \leqslant 2 \times C_{n-1}$. Avec $C_0 = C_1 = 1$, on a directement que $C_n \leqslant 2^n$, donc la complexité de Fibo est $O(2^n)$.

Bien sûr, cette borne n'est pas exacte puisque nous avons fait une approximation grossière. La complexité réelle pourrait être bien plus petite ! Pour prouver que cet algorithme n'est *pas* polynomial, nous pouvons également dire que $C_n \geqslant 2 \times C_{n-2}$ et donc $C_n \geqslant 2^{n/2}$. Cet algorithme a donc une complexité exponentielle, comprise entre $O(2^{n/2})$ et $O(2^n)$.

5.6 Tri fusion

Nous allons voir un algorithme différent pour trier une séquence, basé sur l'idée suivante : si la séquence est de longueur supérieure à 2, on divise la séquence en deux moitié de taille identiques, on trie (récursivement) les deux sous-séquences, puis on les fusionne pour obtenir la séquence triée.

L'implémentation présentée ici n'est pas la plus optimisée, il serait par exemple possible d'effectuer moins de copies entre tableaux. Mais cela rend difficile la lecture de l'algorithme sans pour autant en diminuer la complexité. Nous donnons une implantation à base de tableaux mais il est possible d'utiliser des listes chaînées.

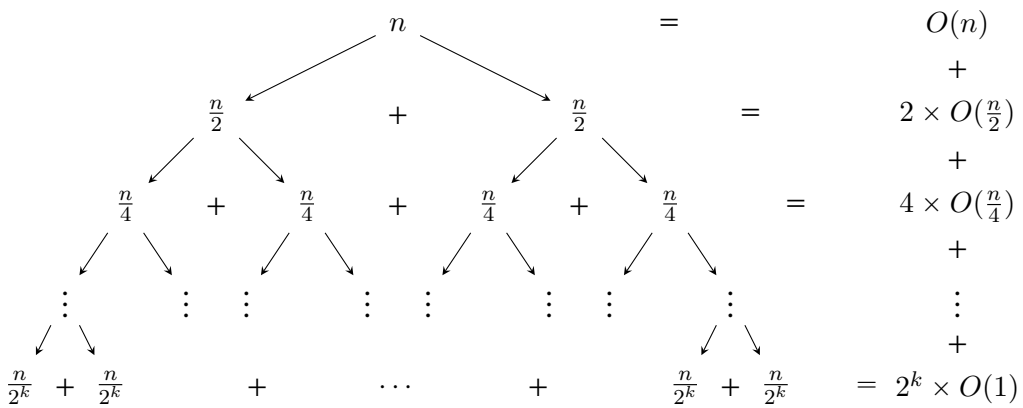
```
Tri(S)
    si S.longueur = 1 alors
        retourner
    m ← S.longueur div 2
    S1 ← m premiers éléments de S
    S2 ← les autres
    Tri(S1)
    Tri(S2)
    Fusion(S, S1, S2)
```

L'algorithme de tri fusion utilise une stratégie classique dite « diviser pour régner ». Elle consiste en découper un gros problème en plus petits problèmes, résoudre récursivement les problèmes, puis combiner les résultats pour obtenir une solution au problème initial. Analysons la complexité de cet algorithme.

```
Fusion(S, S1, S2)
    i ← 0; i1 ← 0; i2 ← 0
tant que i1 < S1.longueur et i2 < S2.longueur faire
    si S1.tab[i1] < S2.tab[i2] alors
        S.tab[i] ← S1.tab[i1]
        i1 ← i1 + 1
    sinon
        S.tab[i] ← S2.tab[i2]
        i2 ← i2 + 1
    i ← i + 1
si i1 = S1.longueur alors
    échanger S1 et S2
    échanger i1 et i2
pour j de i1 à S1.longueur −1 faire
    S.tab[i] ← S1.tab[j]
    i ← i + 1
S.longueur ← S1.longueur + S2.longueur
```

La fonction Tri utilise la fonction Fusion, nous allons donc d'abord analyser cette dernière. Il y a deux boucles : un « tant que » et un « pour », et on ne sait pas à l'avance combien d'opération chacune d'elle va effectuer. En revanche, on sait que le nombre *total* d'itérations combiné des deux boucles est exactement n =S$_1$.longueur + S$_2$.longueur. Puisque les deux boucles font un nombre constant d'opérations, la complexité de la fonction est $O(n)$. En effet, nous copions ici tous les éléments de S$_1$ et S$_2$ vers S exactement une fois.

Voyons maintenant la fonction Tri, qui est récursive. Comme dans la section précédente, écrivons la formule. Si C_n est le nombre d'opérations faites par Tri sur une séquence de taille n, nous avons $C_n = O(n) + 2 \times C_{n/2} + O(n)$. Le premier $O(n)$ correspond à la séparation en S$_1$ et S$_2$ et le deuxième leur fusion. Nous allons maintenant dessiner « l'arbre d'appels » d'une exécution de Tri, où la valeur de chaque nœud représente la taille de la séquence d'un appel (récursif) à Tri.



Le coût du tri est la somme des coûts de tous les nœuds de cet arbre d'appels. Le coût d'un nœud est maintenant simplement le coût de la séparation et de la fusion, puisque le coût des deux appels récursifs est maintenant compté dans les nœuds des fils.

Si l'on regarde les coûts par niveau, on se rend compte que le premier niveau est $O(n)$, le deuxième deux fois $O(n/2)$, le troisième quatre fois $O(n/4)$, etc. Chaque niveau coûte donc $O(n)$, où n est la taille de la séquence initiale, jusqu'au dernier niveau, où il y a 2^k feuilles représentant le coût de « trier » des sous-séquences de taille 1, i.e., $O(1)$, et $2^k = n$.

Au total, la complexité de l'algorithme du tri-fusion est $O(n)\times$ le nombre de niveaux, c'est-à-dire la hauteur de l'arbre. Comme pour la recherche binaire, on trouve que cette hauteur est exactement le nombre de fois qu'il faut diviser n par 2 pour atteindre 1, soit $k = \log n$. La complexité du tri fusion est donc en $O(n \log n)$, ce qui est **bien meilleur** que le tri par sélection analysé plus précédemment.