# Analyse d'un algorithme

En informatique, l'analyse des algorithmes est une partie très cruciale. Il est important de trouver l'algorithme le plus efficace pour résoudre un problème. Il est possible d'avoir de nombreux algorithmes pour résoudre un problème, mais le défi ici est de choisir le plus efficace car les algorithmes efficaces (rapides) sont bien plus importants que le matérielle plus rapide.

La complexité de l'espace et du temps d'exécution agit comme une échelle de mesure des algorithmes. Nous analysons que les algorithmes corrects (sorties corrects) et sur la base de leur besoins en mémoire et le temps d'exécution c'est à dire comment le temps d'exécution augmente à mesure que la taille du problème augmente.

La quantité totale de mémoire de l'ordinateur utilisée par un algorithme lors de son exécution correspond à la complexité de l'espace de cet algorithme. Nous n'aborderons pas la complexité de l'espace dans ce rapport.

## Complexité du temps d'exécution d'un algorithme

La complexité du temps d'exécution est le nombre d'opérations qu'un algorithme effectue pour accomplir sa tâche (étant donné que chaque opération prend le même temps). L'algorithme qui effectue la tâche dans le plus petit nombre d'opérations est considéré comme le plus efficace en termes de complexité du temps d'exécution.

### Types d'analyses

#### Le Pire des Cas

La complexité du pire des cas de l'algorithme est la fonction définie par le nombre maximum de pas effectués sur toute instance de taille *n*. Il fournit une limite supérieure sur le temps de fonctionnement

#### Le Meilleur des Cas

La meilleure complexité des cas de l'algorithme est la fonction définie par le nombre minimum de pas effectués sur toute instance de taille *n*. Il fournit une limite inférieure sur le temps de fonctionnement.

#### Le Cas Moyen

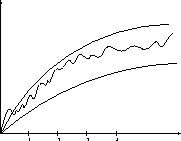
La complexité moyenne des cas de l'algorithme est la fonction définie par un nombre moyen de pas effectués sur toute instance de taille *n*. Il fournit une prédiction sur le temps de fonctionnement.

Dans la pratique, la plus utile de ces trois mesures s'avère être la complexité du pire des cas.

Chacune de ces complexités temporelles définit une fonction numérique, représentant le temps par rapport à la taille du problème. Ces fonctions sont aussi bien définies que toute autre fonction numérique. Les complexités temporelles sont cependant des fonctions compliquées. Afin de simplifier le travail avec de telles fonctions, nous aurons besoin de la notation asymptotique.

### La Notation Asymptotique

Nous savons que les complexités des cas meilleures, pires et moyennes d'un algorithme donné sont des fonctions numériques de la taille des instances. Cependant, il est difficile de travailler précisément avec ces fonctions car elles sont souvent très compliquées, avec de nombreuses petites bosses de haut en bas, comme le montre la figure.



Ainsi, il est généralement plus facile de parler des limites supérieures et inférieures de ces fonctions. C'est là que la notation asymptotique entre en scène.

On utilise les notations suivantes:

#### O Notation (Big O)

​ signifie que ​ est une borne supérieure sur ​. Il existe donc une constante telle que ​ soit toujours inferieure à ​, pour ​ assez grand.

En d'autres termes, ​ est asymptotiquement "inférieure à"​.

#### Ω notation (Big Omega)

​ signifie que ​ est une borne inferieure sur ​. Il existe donc une constante ​telle que ​​ soit toujours supérieure à​, pour ​ assez grand.

En d'autres termes, ​ est asymptotiquement "supérieure à"​.

#### Θ notation (Big Thêta)

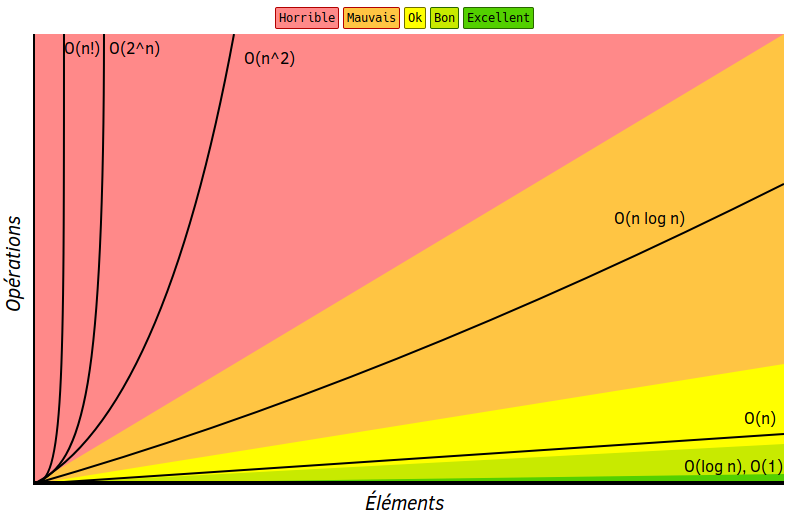
​ signifie que ​​ est une borne supérieure sur ​​ et que ​​ est une borne inferieure sur ​ , pour ​ assez grand. Il existe donc des constantes ​ et ​ telles que ​ ​ et ​. Cela signifie que ​ est une borne étroite sur ​ ​.

En d'autres termes, ​​ et ​​ sont asymptotiquement "égales".

Voici une liste de fonctions en notation asymptotique que nous rencontrons souvent lors de l'analyse d'algorithmes, classées par croissance la plus lente à la plus rapide:

* ​​
* ​
* ​
* ​
* ​
* ​
* ​
* ​
* ​

La figure suivant montre les courbes des temps d'exécution par rapport à la taille du problème pour certaines fonctions:



### Ordre de grandeur

En travaillant avec la notation asymptotique, nous rejetons les constantes multiplicatives et les termes d'ordre inférieur. Les fonctions ​ et ​​ sont traitées de la même manière, même si ​ est un million de fois plus grand que ​ pour toutes les valeurs de​​.

1. **Analyse de l’algorithme de Tri par insertion**

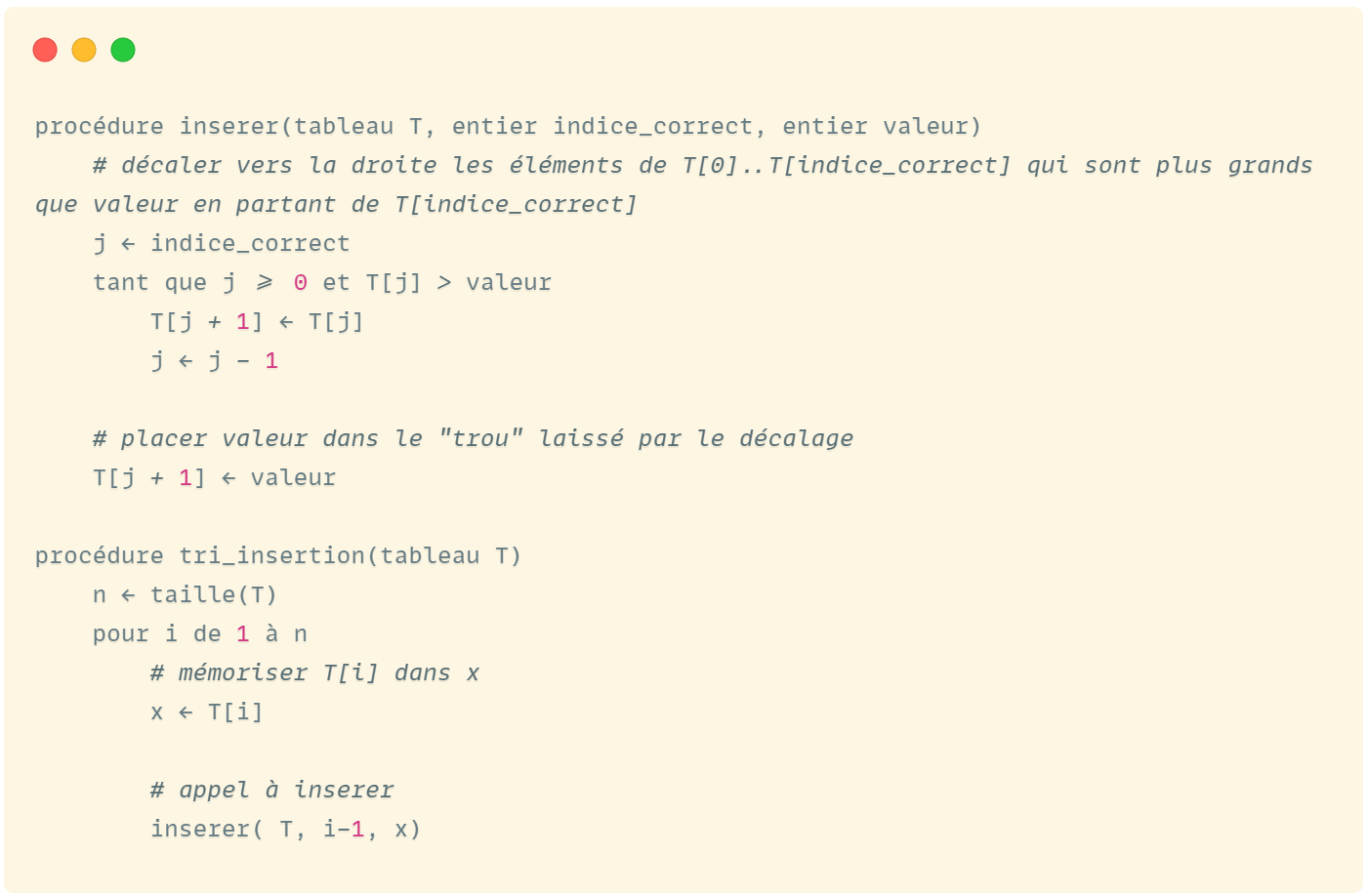
En général, le tri par insertion est beaucoup plus lent que d'autres algorithmes comme le tri rapide et le tri fusion pour traiter de grandes séquences, car sa complexité asymptotique est quadratique.

1. Description de l'algorithme

Dans l'algorithme, on parcourt le tableau à trier du début à la fin. Au moment où on considère le i-ème élément, les éléments qui le précèdent sont déjà triés. Pour faire l'analogie avec l'exemple du jeu de cartes, lorsqu'on est à la i-ème étape du parcours, le i-ème élément est la carte saisie, les éléments précédents sont la main triée et les éléments suivants correspondent aux cartes encore en désordre sur la table.

L'objectif d'une étape est d'insérer le i-ème élément à sa place parmi ceux qui précèdent. Il faut pour cela trouver où l'élément doit être inséré en le comparant aux autres, puis décaler les éléments afin de pouvoir effectuer l'insertion. En pratique, ces deux actions sont fréquemment effectuées en une passe, qui consiste à faire « remonter » l'élément au fur et à mesure jusqu'à rencontrer un élément plus petit.

Voici une description en pseudo-code de l'algorithme présenté. Les éléments du tableau T (de taille n) sont numérotés de 0 à n-1.



1. Analyse de l’algorithme

Le tri par insertion effectue une boucle sur les indices du tableau. Il appelle inserer sur les éléments aux indices 1, 2, 3, ..., n-1. Chaque appel à inserer prend un laps de temps qui dépend de la taille du sous-tableau trié.

Prenons une situation dans laquelle nous appelons inserer et la valeur insérée dans un sous-tableau est inférieure à chaque élément du sous-tableau, alors chaque élément du sous-tableau doit glisser sur une position vers la droite. Donc, en général, si nous insérons dans un sous-tableau avec *k* éléments, tous les *k* devront peut-être glisser d'une position. Plutôt que de compter exactement le nombre de lignes de code dont nous avons besoin pour tester un élément par rapport à une clé et faire glisser l'élément, admettons qu'il s'agit d'un nombre constant de lignes; appelons cette constante *c*. Par conséquent, cela peut prendre jusqu'à *c.k* lignes pour s'insérer dans un sous-tableau de *k* éléments.

Supposons qu'à chaque appel à inserer, la valeur insérée soit inférieure à chaque élément du sous-tableau à sa gauche. Lorsque nous appelons insert la première fois, *k = 1*. La deuxième fois, *k = 2*. La troisième fois, *k = 3*. Et ainsi de suite, jusqu'à la dernière fois, quand *k = n-1*. Par conséquent, le temps total consacré à l'insertion dans des sous-tableaux triés est de :

Cette somme est une série arithmétique, sauf qu'elle va jusqu'à *n-1* plutôt que *n*. En utilisant notre formule pour les séries arithmétiques, nous obtenons que le temps total passé à insérer dans des sous-tableaux triés soit :

En utilisant la notation asymptotique, nous rejetons le terme d'ordre inférieur *cn/2* et les facteurs constants *c* et 1/2, obtenant le résultat que le temps d'exécution du tri par insertion, dans ce cas, est .

Le tri par insertion peut prendre moins de temps à s’exécuter. Supposons que nous ayons le tableau [2, 3, 5, 7, 11], où le sous-tableau trié est les quatre premiers éléments, et que nous insérons la valeur 11. Lors du premier test, nous trouvons que 11 est supérieur à 7, et donc aucun élément du sous-tableau n'a besoin de glisser vers la droite. Ensuite, cet appel d’inserer prend juste un temps constant. Supposons que chaque appel d’inserer prend un temps constant. Comme il y a *n-1* appels à inserer, si chaque appel prend du temps qui est une constante *c*, alors le temps total pour le tri par insertion est , qui est , pas .

Un appel à inserer fait glisser chaque élément si la clé insérée est inférieure à chaque élément à sa gauche. Ainsi, si chaque élément est inférieur à chaque élément à sa gauche, le temps d'exécution du tri par insertion est . Cela signifie que le tableau devrait démarrer dans l'ordre trié inverse, tel que [11, 7, 5, 3, 2]. Un tableau trié inversé est donc le pire des cas pour le tri par insertion.

Un appel à insérer ne fait glisser aucun élément si la clé insérée est supérieure ou égale à chaque élément à sa gauche. Ainsi, si chaque élément est supérieur ou égal à chaque élément à sa gauche, le temps d'exécution du tri par insertion est . Cette situation se produit si le tableau commence étant déjà trié et donc un tableau déjà trié est le meilleur cas pour le tri par insertion.

Supposons que le tableau démarre dans un ordre aléatoire. En moyenne, nous nous attendons à ce que chaque élément soit inférieur à la moitié des éléments à sa gauche. Dans ce cas, en moyenne, un appel à insérer sur un sous-tableau de *k* éléments en ferait glisser *k/2*. Le temps de fonctionnement serait la moitié du temps de fonctionnement le plus défavorable. Mais en notation asymptotique, où les coefficients constants n'ont pas d'importance, le temps d'exécution dans le cas moyen serait toujours , tout comme le pire des cas.

Calculons la complexité de l’algorithme du tri par insertion avec une autre méthode. On remarque qu’il y a deux boucles dans l’algorithme on aura donc :

Pour le pire des cas :

Donc .

Pour le meilleur des cas :

Donc .

Pour résumer les durées d'exécution du tri par insertion:

* Pire cas: .
* Meilleur cas: .
* Cas moyen pour un tableau aléatoire: .

1. Analyse de l’algorithme de tri par fusion

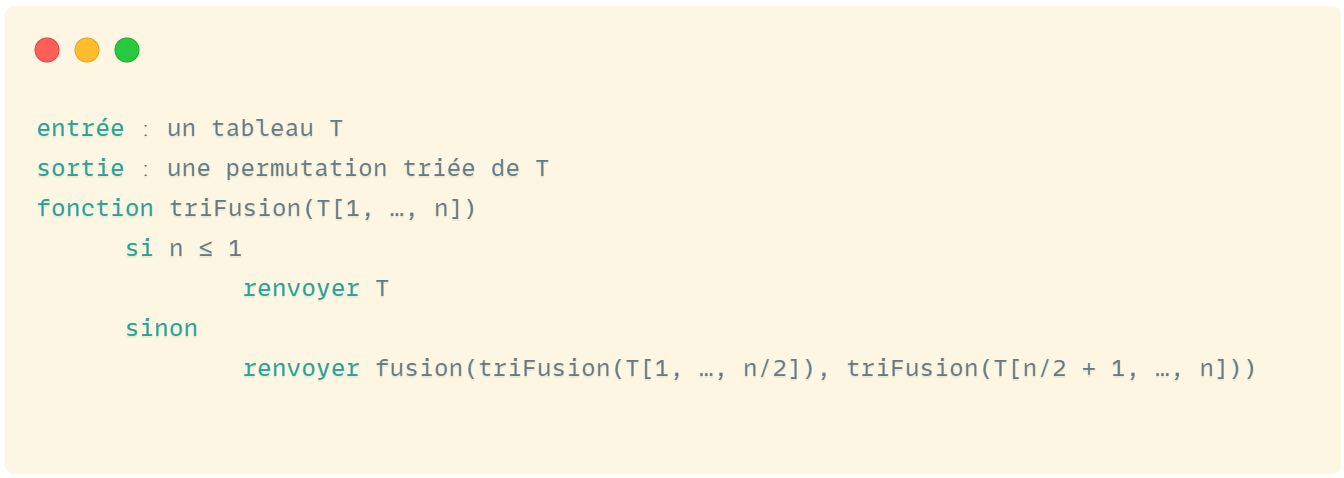
Le tri fusion est un algorithme de tri basé sur la technique algorithmique "diviser pour régner ". L'opération principale de l'algorithme est la fusion, qui consiste à réunir deux listes triées en une seule. L'efficacité de l'algorithme vient du fait que deux listes triées peuvent être fusionnées en temps linéaire. Sa complexité temporelle est asymptotiquement optimale.

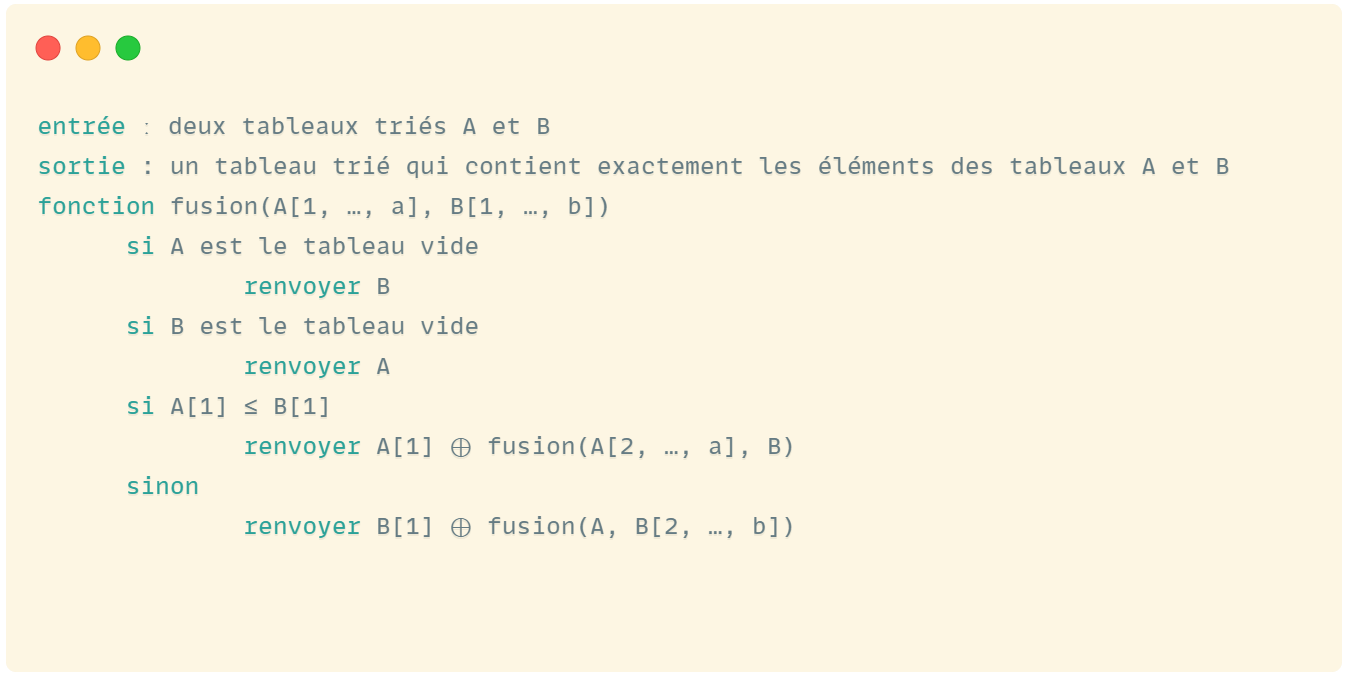
* 1. Description de l’algorithme

L'algorithme est naturellement décrit de façon récursive :

* Si le tableau n'a qu'un élément, il est déjà trié.
* Sinon, séparer le tableau en deux parties à peu près égales.
* Trier récursivement les deux parties avec l'algorithme du tri fusion.
* Fusionner les deux tableaux triés en un seul tableau trié.

En pseudo-code :





L'algorithme se termine car la taille du tableau à trier diminue strictement au fil des appels. Le symbole ⊕ désigne ici la concaténation de tableaux.

2. Analyse de l’algorithme

La fusion de n éléments prend O (n) temps, et par conséquent, le temps d'exécution de la fusion est linéaire dans la taille du sous-tableau. La fusion comporte trois parties:

1. On copie chaque élément du tableau[p..r] dans moitierDown ou moitierUp.
2. Tant que certains éléments ne sont pas pris à la fois dans moitierDown et moitierUp, on compare les deux premiers éléments non pris et on copie le plus petit dans le tableau.
3. Une fois que moitierDown ou moitierUp a eu tous ses éléments copiés dans le tableau, on copie chaque élément non pris restant de l'autre tableau temporaire dans le tableau.

Un nombre constant de lignes de code doit être exécuté par élément pour chacune de ces étapes. Chaque élément est copié du tableau dans moitierDown ou moitierUp exactement une fois à l'étape 1. Chaque comparaison à l'étape 2 prend un temps constant, car elle ne compare que deux éléments, et chaque élément "gagne" une comparaison au plus une fois. Chaque élément est recopié dans le tableau exactement une fois dans les étapes 2 et 3 combinées. Puisque nous exécutons chaque ligne de code un nombre constant de fois par élément et que nous supposons que le sous-tableaux tableau[p..q] contient *n* éléments, le temps d'exécution de la fusion est en effet .

Étant donné que la fonction de fusion s'exécute in time lors de la fusion de n éléments, nous verrons comment tri\_fusion s'exécute en temps . Nous commençons par examiner les trois parties de "diviser-pour-régner" et comment rendre compte de leurs temps de fonctionnement. Nous supposons que nous trions un total de *n* éléments dans le tableau entier.

1. L'étape de division prend un temps constant, quelle que soit la taille du sous-tableau. Après tout, l'étape de division calcule simplement le point médian *q* des indices *p* et *r*. Rappelons que dans la notation asymptotique, on indique le temps constant par .
2. L'étape de conquête, où nous trions de manière récursive deux sous-tableaux d'environ *n/2* éléments chacun, prend un certain temps, mais nous en tiendrons compte lorsque nous considérerons les sous-problèmes.
3. L'étape de combinaison fusionne un total de *n* éléments, en prenant un temps .

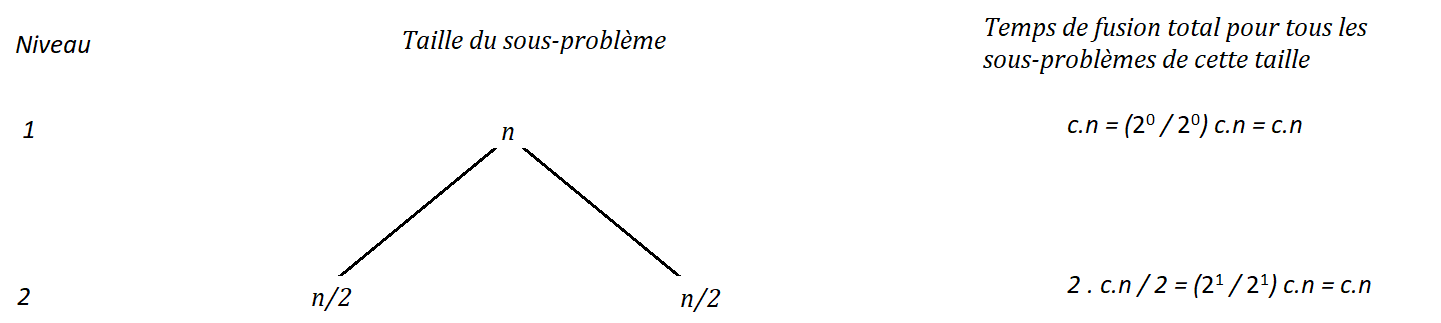
Si nous pensons aux étapes de division et de combinaison, le temps d'exécution pour l'étape de division est un terme d'ordre inférieur par rapport au temps d'exécution de l'étape de combinaison. Considérons donc l’ensemble des étapes de division et de combinaison prenant un temps . Pour rendre les choses plus concrètes, disons que les étapes de division et de combinaison prennent du temps pour une constante *c*.

Supposons également que si *n>1*, alors *n* est toujours pair, de sorte que lorsque nous devons penser à *n/2*, c'est un entier. (Prendre en compte le cas où *n* est impair ne change pas le résultat en termes de notation asymptotique.) Nous pouvons donc maintenant penser au temps d'exécution de tri\_fusion sur un sous-tableau à n éléments comme étant la somme de deux fois le temps tri\_fusion sur un sous-tableau de *(n/2)* éléments (pour l'étape de conquête) plus *cn* (pour les étapes de division et de combinaison).

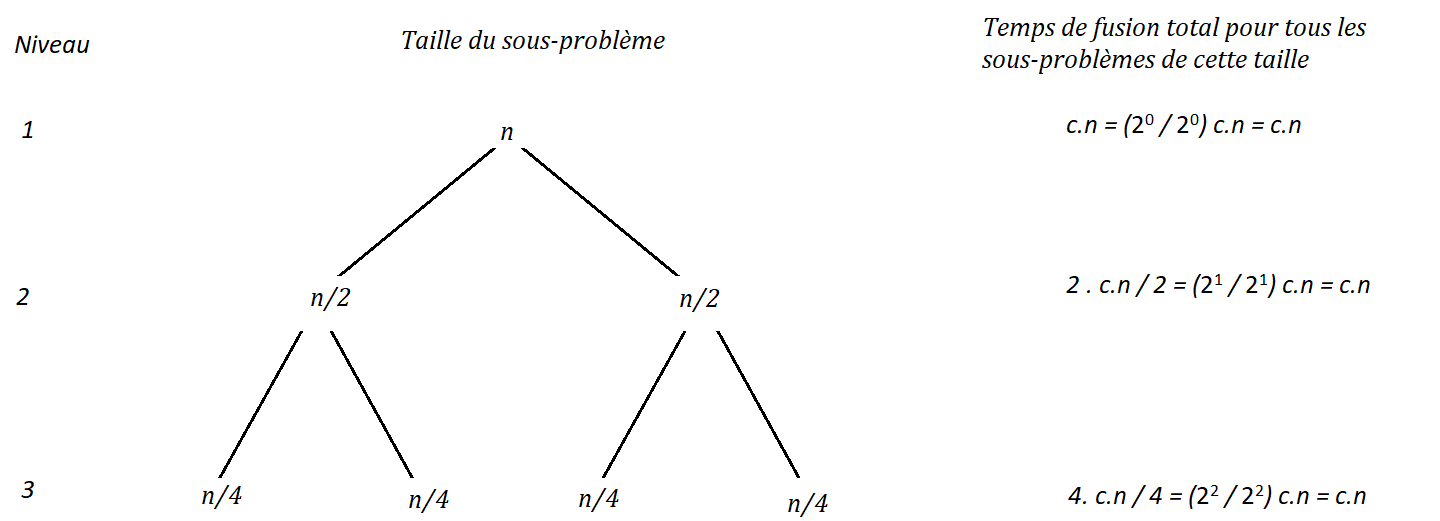
Nous devons maintenant déterminer le temps d'exécution de deux appels récursifs sur *n/2* éléments. Chacun de ces deux appels récursifs prend deux fois le temps d'exécution de tri\_fusion sur un sous-tableau (*n/4*) éléments (car nous devons diviser par deux *n/2*) plus *cn/2* pour fusionner. Nous avons deux sous-problèmes de taille *n/2* et chacun prend un temps de *cn/2* pour fusionner, et donc le temps total que nous passons à fusionner pour les sous-problèmes de taille *n/2* est de .

Nous allons utiliser une des méthodes de résolutions des récurrences à savoir la méthode de l’arbre de récursivité

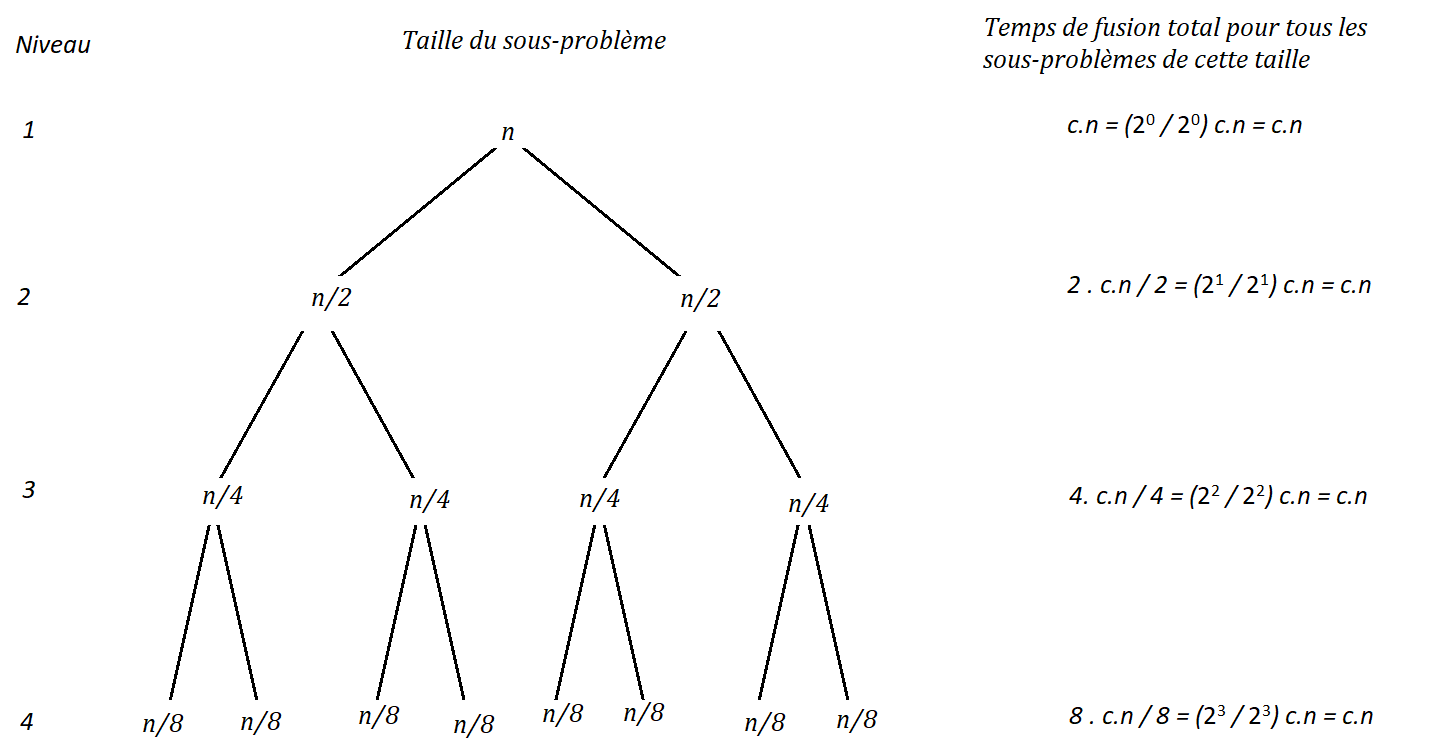
Illustrons les temps de fusion à travers un "arbre":



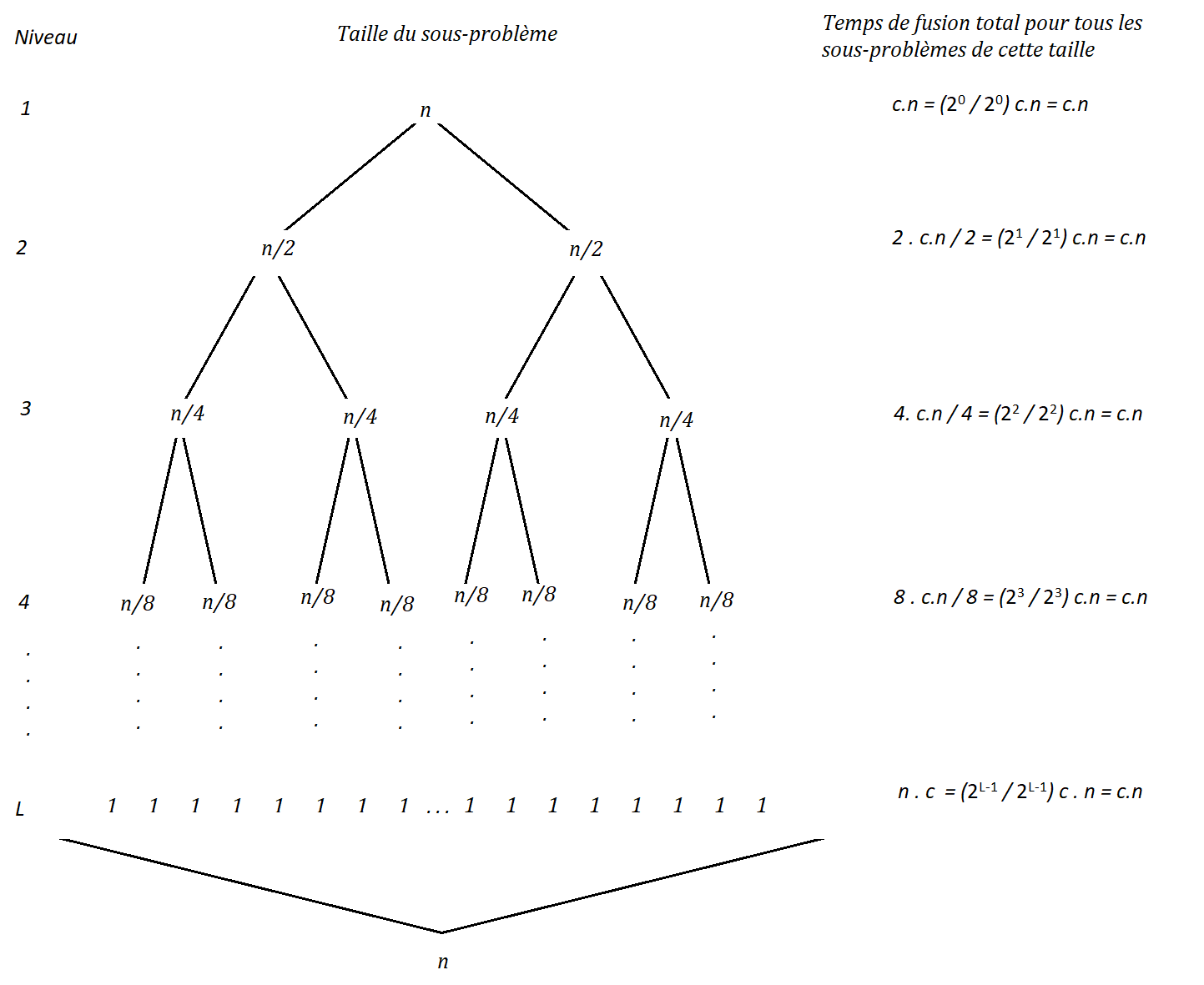
Chacun des sous-problèmes de taille n / 2 trie de manière récursive deux sous-tableaux de taille *(n/2)/2* ou *n/4*. Comme il existe deux sous-problèmes de taille *n/2*, il existe quatre sous-problèmes de taille *n/4*. Chacun de ces quatre sous-problèmes fusionne un total de *n/4* éléments, de sorte que le temps de fusion pour chacun des quatre sous-problèmes est *cn/4*. En résumé sur les quatre sous-problèmes, nous voyons que le temps de fusion total pour tous les sous-problèmes de taille *n/4* est de :



Pour les sous-problèmes de taille *n/8*, il y en aura huit, et le temps de fusion pour chacun sera *cn/8* pour un temps de fusion total de :



Au fur et à mesure que les sous-problèmes deviennent plus petits, le nombre de sous-problèmes double à chaque "niveau" de la récursivité, mais le temps de fusion diminue de moitié. Le doublement et la réduction de moitié s'annulent, et donc le temps de fusion total est *cn* à chaque niveau de récursivité. Finalement, nous arrivons à des sous-problèmes de taille 1: le cas de base. Il faudra un temps pour trier les sous-tableaux de taille 1, car nous devons tester si *p < r*, et ce test prend du temps. Puisque nous avons commencé avec *n* éléments, il doit y avoir *n* sous-tableaux de taille 1. Puisque chaque cas de base prend temps, disons que dans l'ensemble, les cas de base prennent *cn* temps:



Nous savons maintenant combien de temps la fusion prend pour chaque taille de sous-problème. Le temps total pour tri\_fusion est la somme des temps de fusion pour tous les niveaux. S'il y a *L* niveaux dans l'arborescence, alors le temps total de fusion est *L⋅cn*. Alors qu'est-ce que *L*? Nous commençons avec des sous-problèmes de taille *n* et les divisons à plusieurs reprises jusqu'à ce que nous en arrivions aux sous-problèmes de taille 1. Heureusement, il existe une fonction mathématique qui signifie la même chose que le nombre de fois où nous réduisons de moitié à plusieurs reprises, en commençant par *n*, jusqu'à ce que nous obtenions la valeur 1 : le logarithme en base 2 de *n*. Cela s'écrit souvent . De cette façon, *L* serait . Par exemple, si *n = 8*, alors , et bien sûr, l'arbre a quatre niveaux: *n = 8, 4, 2, 1*. Le temps total de tri\_fusion est alors . Lorsque nous utilisons la notation asymptotique pour décrire ce temps d'exécution, nous pouvons rejeter le terme d'ordre inférieur (*+1*) et le coefficient constant (*c*), ce qui nous donne un temps d'exécution de .

En utilisant la methode iterative, on a :

…………………………………………………………. 1 …………………………………………………………. 2

…………………………………………………………. 3

……

…………………………………………………………. k

L’algorithme continue de diviser jusqu’il ne peut plus diviser c’est à dire que le tableau à diviser est de taille 1. La complexité devient donc *T(1)* . Pour calculer la valeur de *k* on a :

En remplaçant la valeur de *k* dans *T(n)* on aura :

En enlevant les termes d ordre inferieur on aura :

Donc la complexité du tri par fusion:

Une autre chose à propos du tri par fusion mérite d'être notée. Lors de la fusion, il effectue une copie de l'ensemble du tableau en cours de tri, avec une moitié dans moitierDown et l'autre moitié dans moitierUp. Comme il copie plus d'un nombre constant d'éléments à un moment donné, nous disons que le tri par fusion ne fonctionne pas en place. En revanche, le tri par insertion fonctionne en place, car il ne copie jamais plus d'un nombre constant d'éléments de tableau à la fois. Les informaticiens se demandent souvent si un algorithme fonctionne en place, car il existe certains systèmes où l'espace est limité, et donc les algorithmes en place sont préférés.

Pour récapituler, voici un tableau des temps d'exécution de ces deux algorithmes de tri que nous avons vus:

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithme | Pire des cas | Meilleur des cas | Cas moyen |
| Tri par insertion |  |  |  |
| Tri par fusion |  |  |  |