

INTRODUCTION AUX ALGORITHMES ET PROGRAMMATION EN C++

V. Chabannes, P. Helluy

OUTLINE

1. Les fonctions en C++
2. Complexité des algorithmes
3. Algorithmes récursifs

1. LES FONCTIONS EN C++

Pour structurer un programme, un des moyens les plus courants est de diviser le code en briques appelées fonctions. On appelle fonction un sous-programme qui permet d'effectuer un ensemble d'instructions par simple appel de la fonction.

Une fonction peut :

- passer plusieurs paramètres, appelés arguments de la fonction.
- renvoyer des objets.
- ne rien renvoyer. On parle de procédure.
- appeler d'autres fonctions, et ainsi de suite.
- s'appeler elle-même : on parle alors de fonctions récursives.

```
#include <iostream>

double my_pow(double a, unsigned int n)
{
    double res = 1;
    for (int i = 0; i < n; ++i)
        res *= a;
    return res;
}

int main()
{
    std::cout << "2^6 = " << my_pow(2.0, 6) << "\n";
}
```

Syntaxe

```
type_de_donnee Nom_De_La_Fonction(type1 argument1, type2 argument2, ...) {  
    liste d'instructions  
}
```

- **type de donnée** représente le type de valeur que la fonction est sensée retourner (char, int, float, ...)
- Si la fonction ne renvoie rien, on utilise le mot-clé `void`
- Le nom de la fonction suit les mêmes règles que les noms de variables :
 - le nom doit commencer par une lettre
 - un nom de fonction peut comporter des lettres, des chiffres et les caractères `_` et `$` (les espaces ne sont pas autorisés !)
 - le nom de la fonction, comme celui des variables, est sensible à la casse (différenciation entre les minuscules et majuscules)
- Les arguments sont facultatifs, mais s'il n'y a pas d'arguments, les parenthèses doivent rester présentes

- La **déclaration** est une instruction fournie au compilateur qui donne les informations nécessaires à son utilisation : `type identificateur(paramètres)`
- La **définition** correspond à l'écriture du corps de la fonction : `type identificateur(paramètres) { instructions }`
- Toute fonction doit être déclarée avant d'être appelée.
- La définition d'une fonction peut faire office de déclaration.

```
int min(int, int); // déclaration

int main()
{
    int i = min(2, 3); // appel
    return 0;
}

// définition
int min(int i, int j)
{
    if (i < j) return i;
    else return j;
}
```

toto.hpp

```
// déclaration
int min(int, int);
```

toto.cpp

```
// définition
int min(int i, int j)
{
    if (i < j) return i;
    else return j;
}
```

Si l'on considère la syntaxe d'un paramètre type `argument`, la variable passée en paramètre est copiée dans une variable locale. C'est cette variable qui est utilisée pour faire les calculs dans la fonction appelée.

```
#include <iostream>
void myfunc(int n)
{
    n = 3;
}

int main()
{
    int k = 6;
    myfunc(k);
    std::cout << "k = " << k << "\n";
}
```

```
user $ ./myprog
k = 6
```

En C++, il est possible de passer en paramètre une variable qui peut ensuite être modifiée dans le corps de la fonction. Pour cela, on utilise les références, dont la syntaxe d'un paramètre sera type `&argument`.

```
#include <iostream>
void myfunc(int &n)
{
    n = 3;
}

int main()
{
    int k = 6;
    myfunc(k);
    std::cout << "k = " << k << "\n";
}
```

```
user $ ./myprog
k = 3
```

Les paramètres d'une fonction peuvent être des pointeurs. Dans ce cas, l'objet qui est pointé peut être initialisé, modifié ou détruit dans la fonction.

```
void init(double *n) { *n = 3.14; }  
void modif(double *n) { *n = 10.3; }
```

Le passage d'un tableau comme paramètre d'une fonction s'effectue par le biais de l'adresse de son premier élément, c'est-à-dire par un pointeur. Plusieurs syntaxes sont possibles comme ci-dessous, mais elles sont toutes équivalentes :

```
void f(char *c);  
void f(char c[]);  
void f(char c[16]);
```

À l'intérieur des crochets, le nombre ne sert à rien : ce n'est qu'une indication pour l'utilisateur. La taille du tableau n'est pas accessible dans la fonction ; elle est généralement passée aussi en paramètre.

L'autre façon de définir la fonction main est la suivante :

```
int main(int argc, char *argv[])
```

Le paramètre argv est en réalité un tableau de pointeurs, où chaque case du tableau pointe sur une chaîne de caractères. Le paramètre argc, quant à lui, indique simplement le nombre de chaînes de caractères.

Le premier élément du tableau argv représente le nom de l'exécutable. Les suivants sont les options données à l'exécutable (séparées par des espaces).

```
int main(int argc, char* argv[])
{
    std::cout << "argc=" << argc << "\n";
    for (int k = 0; k < argc; ++k)
        std::cout << "argv[" << k << "] : "
                    << argv[k] << "\n";
    return 0;
}
```

```
user $ ./myprog abc toto
argc=3
argv[0] : ./myprog
argv[1] : abc
argv[2] : toto
```

On peut donner des valeurs par défaut à certains paramètres de nos fonctions. On n'est ainsi plus obligé d'indiquer tous les paramètres à chaque appel.

```
int myfunc(int a, int b = 1) { return a*b; }
int main()
{
    int res1 = myfunc(2);    // res == 2
    int res2 = myfunc(3, 5); // res == 15
    return 0;
}
```

Les paramètres facultatifs doivent obligatoirement se trouver à la fin (à droite).

```
int myfunc(int a, int b = 1, int c = 2) { return a*b*c; } // OK
int myfunc(int a = 1, int b, int c = 2) { return a*b*c; } // PAS OK
```

Les valeurs par défaut sont spécifiées uniquement dans le prototype, pas dans la définition de la fonction.

```
// Prototype avec les valeurs par défaut
int nombreDeSecondes(int heures, int minutes = 0, int secondes = 0);
// Main
int main()
{
    std::cout << nombreDeSecondes(1, 10, 25) << std::endl;
    return 0;
}
// Définition de la fonction, SANS les valeurs par défaut
int nombreDeSecondes(int heures, int minutes, int secondes)
{
    int total = 0;
    total = heures * 60 * 60;
    total += minutes * 60;
    total += secondes;
    return total;
}
```

Il est possible de créer plusieurs fonctions ayant le même nom. Il faut alors que la liste des arguments des deux fonctions soit différente : c'est ce qu'on appelle la **surcharge** d'une fonction. C'est possible car, en C++, la **signature** d'une fonction est déterminée par son nom et ses paramètres.

```
void showMe(int a) {  
    std::cout << "Je suis un int : " << a << "\n";  
}  
void showMe(double a) {  
    std::cout << "Je suis un double : " << a << "\n";  
}  
int main()  
{  
    showMe(3);  
    showMe(5.3);  
    return 0;  
}
```

```
user $ ./myprog  
Je suis un int : 3  
Je suis un double : 5.3
```

Définition

Une fonction ou procédure récursive est une fonction qui s'appelle elle-même. Si, dans le corps (le contenu) de la fonction, la fonction s'appelle elle-même, elle est dite récursive.

```
int fibo(int n)
{
    if (n <= 2)
        return 1;
    else
        return fibo(n-1) + fibo(n-2);
}
int main()
{
    std::cout << "fibonacci(10)= " << fibo(10) << "\n";
    return 0;
}
```

```
user $ ./myprog
fibonacci(10)=55
```

Une fonction récursive a souvent l'avantage (comparée à son équivalent itératif s'il existe) d'être :

- Plus simple et courte à écrire.
- Plus facilement compréhensible (proche des maths).
- Adaptée aux stratégies “diviser pour régner”.

Le piège de la récursivité

Le principal inconvénient est le coût en espace mémoire des algorithmes récursifs. En effet, chaque appel récursif empile une adresse de retour et les paramètres en cours (la pile étant de taille limitée).

Dans certains cas, l'analyse de la complexité algorithmique peut aussi montrer que la version récursive d'un algorithme sera bien moins efficace que sa version itérative (exemple Fibonacci).

Dans le cas de récursivité dite terminale, le compilateur est généralement capable d'optimiser la fonction en la transformant en une version itérative plus efficace.

Un pointeur de fonction est un pointeur comme un pointeur de variable, mais il pointe sur l'adresse d'une fonction.

Syntaxe

```
type_de_retour (*monPointeur)(type_argument_1, type_argument_2, ...)
```

Une fois le pointeur défini, on peut appeler la fonction à partir de ce pointeur.

```
int f(int x, int y) { return x+y; }

int main()
{
    int (*pf)(int, int) = f; // pf est le pointeur de fonction
    std::cout << "pf(2,3) = " << pf(2,3) << "\n"; // va afficher 5
    return 0;
}
```

Le pointeur de fonction peut être un argument d'une autre fonction.

```
#include <iostream>
#include <cmath>

double carre(double x) { return x*x; }
double inverse(double x) { return 1/x; }
double racine(double x) { return sqrt(x); }

double minimum(double a, double b, double (*f)(double)) // on passe le pointeur en argument
{
    double min = f(a);
    for (double x = a; x < b; x += 0.01)
        min = min < f(x) ? min : f(x);
    return min;
}

int main()
{
    int type;
    std::cout << " Entrer le type de fonction : 1= carre ,2= inverse ,3= racine \n";
    std::cin >> type;
    double (*monPointeur)(double);
    switch (type) { // on définit le pointeur sur la fonction choisie
        case 1: monPointeur = carre; break;
        case 2: monPointeur = inverse; break;
        case 3: monPointeur = racine; break;
    }
    std::cout << "Le minimum de la fonction entre 2 et 3 est: " << minimum(2,3,monPointeur) << "\n";
    return 0;
}
```

Le C++ dispose du mot clé `inline`, qui permet de modifier la méthode d'implémentation des fonctions. Placé devant la déclaration d'une fonction, il propose au compilateur de ne pas instancier cette fonction : le compilateur remplace alors l'appel de la fonction par le code correspondant.

```
inline int max(int i, int j)
{
    if (i > j) return i;
    else return j;
}
```

C'est un mécanisme d'optimisation : l'appel de fonction, le passage des paramètres et le retour de la valeur sont ainsi évités. Le compilateur peut aussi effectuer des optimisations supplémentaires.

Toutefois, il y a des restrictions à l'utilisation des fonctions inline :

- les appels récursifs ne sont pas inlinés
- prendre un pointeur de fonction force en général une version hors ligne

La bibliothèque standard du C++ implémente l'ensemble des fonctions mathématiques de base, disponibles via l'en-tête `cmath`. Voici quelques exemples ; une liste plus exhaustive se trouve [ici](#).

```
#include <cmath>

int main()
{
    double a = sin(3.14);
    double b = cos(2);
    double c = exp(3);
    double d = pow(c, 3);
    double e = sqrt(5.17);
    return 0;
}
```

2. COMPLEXITÉ DES ALGORITHMES

Quand on tente de résoudre un problème, il faut souvent choisir un algorithme. L'exécution d'un algorithme mobilise en effet les ressources de l'ordinateur et entraîner un coût :

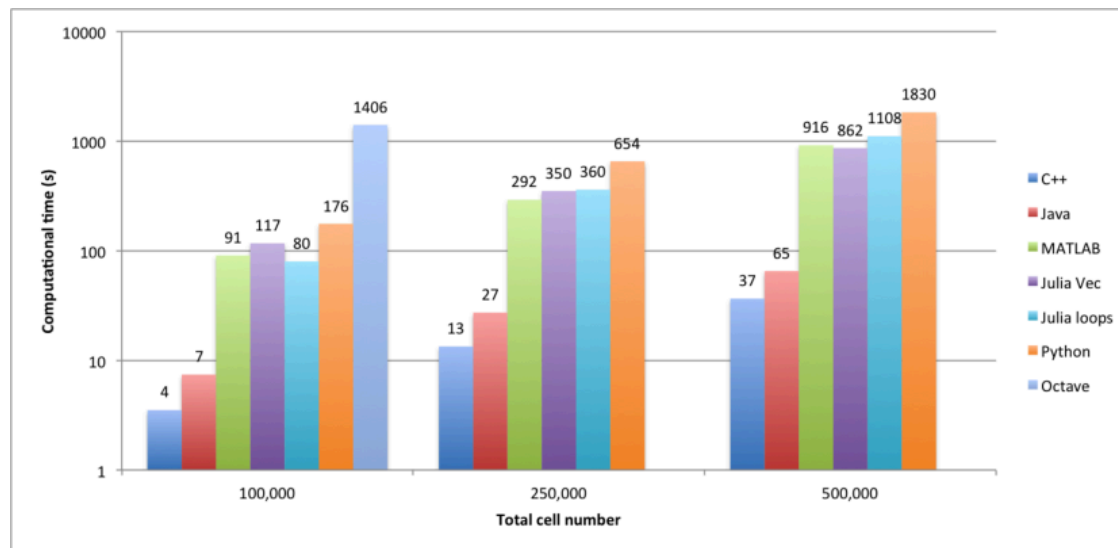
- Temps d'exécution : complexité en temps
- Espace mémoire requis : complexité en espace

L'étude de la complexité des algorithmes a pour objectif l'estimation de ce coût. Elle permet de comparer plusieurs algorithmes sans les programmer, de choisir le mieux adapté (le plus rapide, le moins gourmand en mémoire, ...) et d'exploiter au mieux les ressources de l'ordinateur.



Si l'on prend en compte pour l'estimation de la complexité les ressources de la machine (fréquence d'horloge, nombre de processeurs, le temps d'accès au disque, etc.), on se rend vite compte de la complication, voire de l'impossibilité, d'une telle tâche.

Aussi, on se contente souvent d'estimer la relation entre les données à traiter et le temps d'exécution, indépendamment de l'architecture, du langage et du compilateur (ou de l'interpréteur).



On étudie généralement la complexité en temps ou en espace en fonction de la taille de la donnée en entrée de l'algorithme.

Exemples

- Recherche d'un élément E dans une liste $L \rightarrow$ taille de la liste L
- Tri d'une liste $\rightarrow$ taille de la liste à trier
- Calculer le produit d'une matrice par elle-même $\rightarrow$ dimension de la matrice
- Recherche d'un motif de taille m dans un texte de n caractères $\rightarrow$ 2 tailles m et n à prendre en compte

Pour certains problèmes, la complexité algorithmique ne dépend pas seulement de la taille de la donnée, mais aussi de la donnée elle-même. On verra des exemples plus tard avec les algorithmes récurifs.

Dans ce cours, on s'intéressera principalement à la complexité en temps, la complexité en espace étant en général moins critique.

Mesurer le temps de calcul

Consiste à sommer le temps d'exécution associé à chaque instruction du pseudo-code.

On fait l'hypothèse que chacune des opérations de base apparaissant dans la description d'un algorithme :

- arithmétique
- affectation de variable
- comparaisons
- etc...

consomme un temps élémentaire, que l'on borne par une constante C .

Notations

On note n la taille des données utilisées en entrée de l'algorithme et $T(n)$ le temps d'exécution qui est donc une fonction de $\mathcal{N}$ dans $\mathcal{R}_+^*$.

La complexité d'un algorithme n'est pas forcément la même pour deux jeux de données différents (mais de même taille).

Soit D_n l'ensemble des données d'entrée possibles de taille n d'un problème et $\tilde{T}(i_n)$ le temps de calcul pour une donnée d'entrée $i_n \in D_n$.

- Dans le pire des cas : $T_{\max}(n) = \max_{i_n \in D_n} \tilde{T}(i_n)$
- Dans le meilleur des cas : $T_{\min}(n) = \min_{i_n \in D_n} \tilde{T}(i_n)$
- Pour le cas moyen : $T_{\text{moy}}(n) = \sum_{i_n \in D_n} p(i_n) \tilde{T}(i_n)$
avec p une distribution de probabilité sur D_n

On se focalise généralement sur le cas le plus défavorable ($T(n) \equiv T_{\max}(n)$) :

- Donne une borne supérieure sur le temps d'exécution.
- Le meilleur cas n'est généralement pas représentatif.
- Le cas moyen est souvent difficile à évaluer.

Entrées

- Tableau Tab de n éléments
- un élément x à rechercher

Sortie : la position de x dans Tab

Meilleur des cas

- 1 affectation de i
- 1 comparaison entre i et $n - 1$
- 1 accès à Tab[i]
- 1 comparaison entre Tab[i] et x

$$\Rightarrow T_{\min}(n) = 4$$

```
1: for i = 0 to n - 1 do
2:   if Tab[i] = x then
3:     return i
4:   end if
5: end for
```

Pire des cas

- n affectations de i
- n comparaisons entre i et $n - 1$
- n accès à Tab[i]
- n comparaisons entre Tab[i] et x

$$\Rightarrow T_{\max}(n) = 4n$$

Entrées

- Tableau Tab de n éléments
- un élément x à rechercher

Sortie : la position de x dans Tab

Complexité en moyenne

- q la probabilité que x soit présent
- si x est présent, on suppose qu'il peut se trouver à chaque indice avec la même probabilité
- D_n^i , $0 \leq i \leq n - 1$, ensemble de tableaux où x est à la i^{e} place
- D_n^n , ensemble de tableaux où x n'est pas présent
- $p(D_n^i) = q/n$ et $T(D_n^i) = 4(i + 1)$
- $p(D_n^n) = 1 - q$ et $T(D_n^n) = 4n$

```
1: for i = 0 to n - 1 do
2:   if Tab[i] = x then
3:     return i
4:   end if
5: end for
```

Entrées

- Tableau Tab de n éléments
- un élément x à rechercher

Sortie : la position de x dans Tab

Complexité en moyenne

$$\begin{aligned} T_{\text{moy}}(n) &= \sum_{i=0}^n p(D_n^i) T(D_n^i) = (1-q)4n + \sum_{i=0}^{n-1} \frac{q}{n} 4(i+1) \\ &= (1-q)4n + 4q \frac{n(n+1)}{n2} = (1-q)4n + 2q(n+1) \end{aligned}$$

- si $q = 1$ alors $T_{\text{moy}}(n) = 2(n+1)$
- si $q = 1/2$ alors $T_{\text{moy}}(n) = 3n + 1$

```
1: for i = 0 to n - 1 do
2:   if Tab[i] = x then
3:     return i
4:   end if
5: end for
```

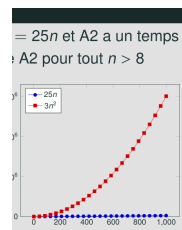
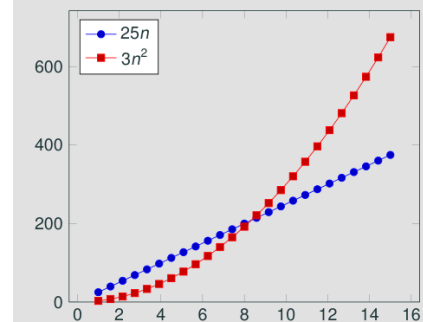
Ce qui nous intéresse dans la complexité d'un algorithme, c'est la rapidité de croissance de la fonction $T(n)$ lorsque n devient très grand : **ordre de grandeur asymptotique**

⇒ Le décompte exact du nombre d'instructions n'est pas le plus important. De même, les constantes multiplicatives comptent peu.

Exemple

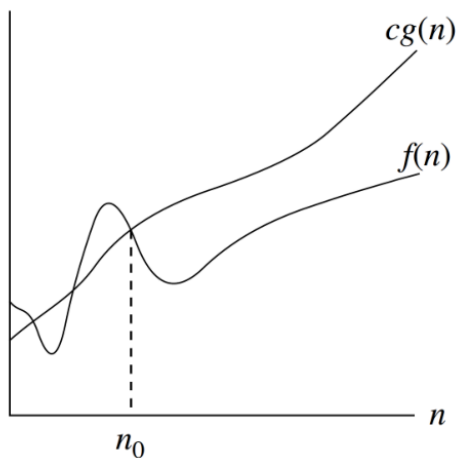
Si un algorithme A_1 a un temps d'exécution $T_1(n) = 25n$ et A_2 a un temps d'exécution $T_2(n) = 3n^2$, alors A_1 est meilleur que A_2 pour tout $n > 8$

l'algorithme A_1 a un temps d'exécution $T_1(n) = 25n$ et A_2 a un temps d'exécution $T_2(n) = 3n^2$, alors A_1



$O(g)$ est l'ensemble des fonctions positives f pour lesquelles il existe une constante strictement positive c et un rang n_0 tels que :

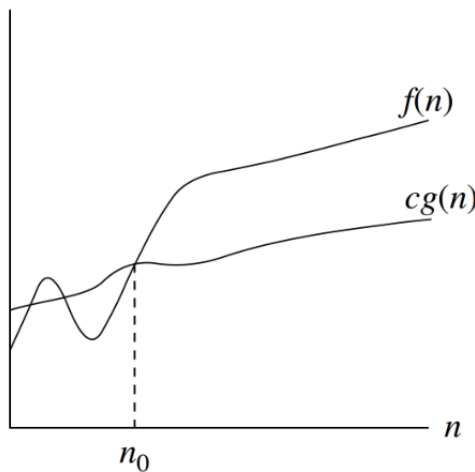
$$f(n) \leq cg(n), \quad \forall n \geq n_0$$



$f(n) \in O(g(n)) \Rightarrow g(n)$ est une borne supérieure asymptotique pour $f(n)$.

$\Omega(g)$ est l'ensemble des fonctions positives f pour lesquelles il existe une constante strictement positive c et un rang n_0 tels que :

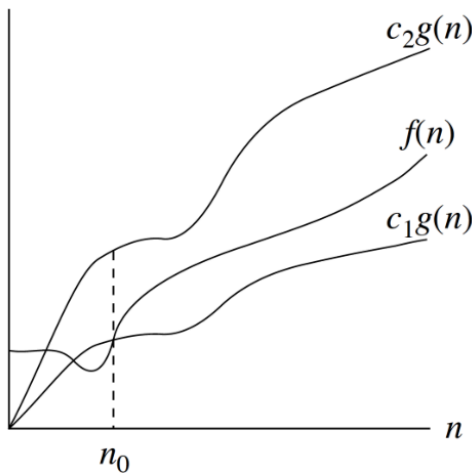
$$f(n) \geq cg(n), \quad \forall n \geq n_0$$



$f(n) \in \Omega(g(n)) \Rightarrow g(n)$ est une borne inférieure asymptotique pour $f(n)$.

$\Theta(g)$ est l'ensemble des fonctions positives f pour lesquelles il existe deux constantes strictement positives c_1 et c_2 et un rang n_0 tels que :

$$c_1 g(n) \leq f(n) \leq c_2 g(n), \quad \forall n \geq n_0$$



$f(n) \in \Theta(g(n)) \Rightarrow g(n)$ est une borne serrée asymptotique pour $f(n)$.

- On utilise les notations asymptotiques pour caractériser la complexité d'un algorithme.
- Il faut préciser de quelle complexité on parle : pire des cas, meilleur des cas, en moyenne
- La notation O est la plus utilisée dans la pratique :
 - On va chercher un majorant du temps d'exécution
 - $f(n) \in O(g(n))$ sous-entend généralement que $O(g(n))$ est le plus petit sous-ensemble qui contient $f(n)$ et que $g(n)$ est la plus concise possible
 - Exemple : $n^3 + 100n^2 - n \in O(n^3) = O(n^3 + n^2) \subset O(n^4) \subset O(2^n)$
- Idéalement, les notations O et Ω devraient être limitées au cas où on n'a pas de borne serrée Θ .

Soient f, g, h, l des fonctions positives et $a, b > 0$.

X désigne l'une quelconque des notations O , Ω ou Θ .

Relation entre notations

- $f(n) \in \Omega(g(n)) \Leftrightarrow g(n) \in O(f(n))$
- $f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n))$ et $f(n) \in \Omega(g(n))$

Symétrie

- $f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$

Transitivité

- $f(n) \in X(g(n))$ et $g(n) \in X(h(n)) \Rightarrow f(n) \in X(h(n))$

Somme et produit

- $f(n) \in X(h(n))$ et $g(n) \in X(h(n)) \Rightarrow af(n) + bg(n) \in X(h(n))$
- $f(n) \in X(h(n))$ et $g(n) \in X(l(n)) \Rightarrow f(n)g(n) \in X(h(n)l(n))$

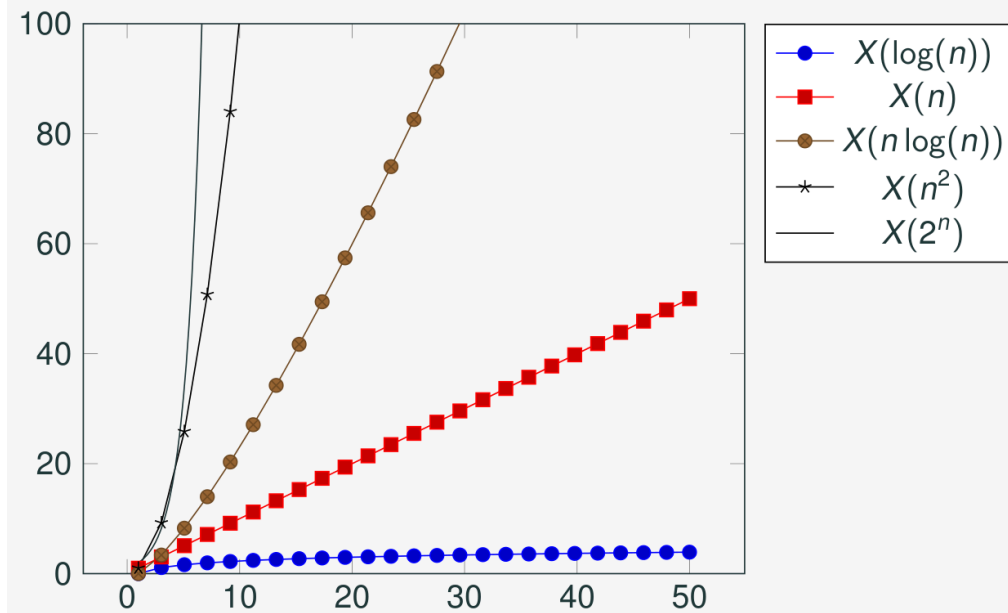
X désigne l'une quelconque des notations O , Ω ou Θ .

Ordre de grandeur	Classe	Exemple de problème
$X(1)$	constante	accès à une cellule de tableau $\Rightarrow \Theta(1)$
$X(\log(n))$	logarithmique	recherche dichotomique dans un tableau trié de taille $n \Rightarrow \Theta(\log(n))$
$X(n)$	linéaire	calcul du produit scalaire de deux vecteurs de taille $n \Rightarrow \Theta(n)$
$X(n \log(n))$	quasi-linéaire	tri par fusion d'une séquence de taille $n \Rightarrow \Theta(n \log(n))$
$X(n^k)$	polynomiale	multiplication de deux matrices carrées d'ordre $n \Rightarrow \Theta(n^3)$
$X(a^n)$	exponentielle	tours de Hanoï : nombre de mouvements minimum $\Rightarrow \Theta(2^n)$

Les quatre premières classes sont considérées rapides alors que $X(n^k)$ est considérée lente et $X(a^n)$ est considérée impraticable.

X désigne l'une quelconque des notations O , Ω ou Θ .

X désigne n'importe lequel des notations O , Ω ou Θ .



Aucun progrès technologique (modèle de machine standard) ne permet à un algorithme de changer de classe de complexité.

Exemple : temps de calcul fixé

Effets de la multiplication de la puissance d'une machine par 10, 100 et 1000 sur la taille maximale N des problèmes que peuvent traiter des algorithmes en un temps donné.

Complexité	×1	×10	×100	×1000
$\Theta(\log(n))$	N	$10N$	$100N$	$1000N$
$\Theta(n)$	N	$10N$	$100N$	$1000N$
$\Theta(n \log(n))$	N	$< 10N$	$< 100N$	$< 1000N$
$\Theta(n^2)$	N	$\sqrt{10}N \simeq 3N$	$10N$	$\sqrt{1000}N \simeq 32N$
$\Theta(2^n)$	N	$N + \log_2(10) \simeq N + 3$	$N + \log_2(100) \simeq N + 7$	$N + \log_2(1000) \simeq N + 10$

Aucun progrès technologique (modèle de machine standard) ne permet à un algorithme de changer de classe de complexité.

Exemple : donnée de taille fixée

Effets du temps de calcul en fonction de la puissance de calcul de 10^6 , 10^9 et 10^{12} flops pour des données de taille 1 million.

Complexité	10^6	10^9	10^{12}
$\Theta(\log(n))$	0.013 ms	0.013 μ s	0.013 ns
$\Theta(n)$	1 s	1 ms	1 μ s
$\Theta(n \log(n))$	14 s	14 ms	14 μ s
$\Theta(n^2)$	278 h	0.28 h	1 s
$\Theta(2^n)$	10000 ans	10 ans	1 semaine

X désigne l'une quelconque des notations O , Ω ou Θ .

- Les instructions élémentaires (affectations, comparaisons, ...) sont en temps constant, soit en $\Theta(1)$.
- La complexité d'une séquence d'instructions est la plus forte complexité de toutes les instructions :
 $a \in X(A), b \in X(B)$ et $c \in X(C)$

$$\{a \ b \ c\} \in X(\max(A, B, C))$$

- Structure conditionnelle

- Notation O : $a \in O(A), b \in O(B)$ et $c \in O(C)$

$$(\text{if } a \text{ then } b \text{ else } c) \in O(A + \max(B, C))$$

- Notation Ω : $a \in \Omega(A), b \in \Omega(B)$ et $c \in \Omega(C)$

$$(\text{if } a \text{ then } b \text{ else } c) \in \Omega(A + \min(B, C))$$

- Structures itératives : $a_i \in X(A_i)$

$$\text{for } i \text{ from } 1 \text{ to } n \ a_i \} \in X\left(\sum_{i=1}^n A_i\right)$$

- Structures itératives : $a_i \in X(A) \ (A_i == A \forall i)$

$$\text{for } i \text{ from } 1 \text{ to } n \text{ do } a_i \} \in X(nA)$$

- Double boucle : $a_{ij} \in X(A)$

$$\text{for } i \text{ from } 1 \text{ to } n \text{ for } j \text{ from } 1 \text{ to } n \ a_{ij} \} \in X(n^2 A) \quad \text{for } i \text{ from } 1 \text{ to } n \text{ for } j \text{ from } 1 \text{ to } i \ a_{ij} \} \in X(n^2 A)$$

- Boucles avec un incrément exponentiel : $k \in \mathcal{N}^*$ et $a_i \in X(1)$

$$i = 1 \text{ while } i \leq n \ a_i \ i = k \ i \} \in X(\log_k(n))$$

Problème du tri

Consiste à organiser une collection d'éléments selon une relation d'ordre déterminée.

- **Applications innombrables :**
 - Tri des mails selon leur ancienneté
 - Tri des résultats de requête dans un moteur de recherche
 - Tri des facettes des objets pour l'affichage dans les jeux 3D
 - Gestion des opérations bancaires
 - ...
- Le tri sert aussi de brique de base / pré-traitement pour d'autres algos
 - Recherche par dichotomie
 - Recherche des doublons dans un tableau
 - Recherche du k^{e} élément le plus grand dans une liste
 - ...
- **Différents algorithmes :** tri par insertion, tri par sélection, tri par fusion, tri rapide, tri à bulles, ...



Require: A séquence à trier de taille n

```
1: for j = 2 to n do      ▷ c1
2:   key = A[j]           ▷ c2
3:   i = j - 1            ▷ c3
4:   while i > 0 and A[i] > key do ▷ c4
5:     A[i + 1] = A[i]    ▷ c5
6:     i = i - 1          ▷ c6
7:   end while
8:   A[i + 1] = key       ▷ c7
9: end for
```

Tableau trié

- $j = 2$: 13 14 15 16 17
(0 iter)
- $j = 3$: 13 14 15 16 17
(0 iter)
- $j = 4$: 13 14 15 16 17
(0 iter)
- $j = 5$: 13 14 15 16 17
(0 iter)

Meilleur des cas

- Le tableau est déjà trié
- La condition $A[i] > \text{key}$ sera toujours fausse
- $T(n) = (n - 1)(c_1 + c_2 + c_3 + c_4 + c_7) + c_1$
- $T(n) \in \Theta(n)$

Require: A séquence à trier de taille n

```
1: for j = 2 to n do      ▷ c1
2:   key = A[j]           ▷ c2
3:   i = j - 1           ▷ c3
4:   while i > 0 and A[i] > key do ▷ c4
5:     A[i + 1] = A[i]    ▷ c5
6:     i = i - 1         ▷ c6
7:   end while
8:   A[i + 1] = key       ▷ c7
9: end for
```

Tableau trié en sens inverse

- $j = 2$: 17 16 15 14 13
(1 iter)
- $j = 3$: 16 17 15 14 13
(2 iter)
- $j = 4$: 15 16 17 14 13
(3 iter)
- $j = 5$: 14 15 16 17 13
(4 iter)

Pire des cas

- La boucle while s'arrêtera lorsque $i == 0$ ($A[i] > \text{key}$ sera toujours vrai)
- $$T(n) = (n-1)(c_1 + c_2 + c_3 + c_7) + (c_4 + c_5 + c_6) \sum_{k=1}^{n-1} k + c_1$$
$$= (n-1)(c_1 + c_2 + c_3 + c_7) + (c_4 + c_5 + c_6)(n(n-1)/2) + c_1$$
- $T(n) \in \Theta(n^2)$

Require: A séquence à trier de taille n

```
1: for j = 2 to n do      ▷ c1
2:   key = A[j]           ▷ c2
3:   i = j - 1            ▷ c3
4:   while i > 0 and A[i] > key do ▷ c4
5:     A[i + 1] = A[i]    ▷ c5
6:     i = i - 1          ▷ c6
7:   end while
8:   A[i + 1] = key        ▷ c7
9: end for
```

Exemple de tableau du cas moyen

- $j = 2$: 14 13 15 17 16
(1 iter)
- $j = 3$: 13 14 15 17 16
(0 iter)
- $j = 4$: 13 14 15 17 16
(0 iter)
- $j = 5$: 13 14 15 17 16
(1 iter)

Cas moyen (avec équiprobabilité)

- La moitié du tableau $A[1..j-1]$ est $< A[j]$ et l'autre est $> A[j]$
- En moyenne on teste la moitié du sous-tableau $A[1..j-1]$
- $T(n) = (n-1)(c_1 + c_2 + c_3 + c_7) + (c_4 + c_5 + c_6) \sum_{k=1}^{n-1} k/2 + c_1$
- $T(n) \in \Theta(n^2)$

- Pareil que pour la complexité en temps, sauf qu'on compte les utilisations élémentaires de l'espace mémoire plutôt que les temps de calcul élémentaires
- On fait souvent des hypothèses simplificatrices, comme le fait que le stockage de n'importe quel entier utilise un espace mémoire constant
- On ne tient pas compte de l'espace nécessaire à représenter l'entrée
- On utilise aussi O , Ω , Θ pour résumer la complexité asymptotique
- Un algorithme est dit **en place** s'il n'utilise qu'un nombre très limité de variables, indépendamment de la taille n i.e. $\Theta(1)$

Exemple : recherche dans un tableau

La complexité en espace est en $O(1)$: il suffit de stocker la variable i en mémoire (en plus des entrées Tab et x), ce qui occupe un espace élémentaire indépendant de la taille de l'entrée

3. ALGORITHMES RÉCURSIFS

Définition

On appelle récursive toute fonction ou procédure qui s'appelle elle-même.

La programmation récursive est un moyen simple et élégant pour les problèmes définis par récurrence (cf. $n!$, x^n , $f(n)$, ...) et à la stratégie “diviser pour régner”.

```
Require: n : un réel  
Require: p : un entier  $\geq 0$   
1: procédure Puissance(n,p)  
2:   if p = 0 then  
3:     return 1  
4:   else  
5:     return n × Puissance(n,p - 1)  
6:   end if  
7: end procédure
```

$$n^p = ?$$

$$\text{Puissance}(5,3) = 5 \times \text{Puissance}(5,2) = 5 \times 5 \times \text{Puissance}(5,1) = 5 \times 5 \times 5 \times \text{Puissance}(5,0) = 5 \times 5 \times 5 \times 1 = 125$$

Tout algorithme récursif doit être soigneusement justifié :

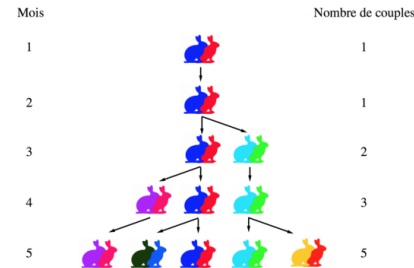
- **preuve de la terminaison** : les appels récursifs sont soumis à une condition dont il faut montrer qu'elle devient fausse après un nombre fini d'appels. Typiquement, on associe aux paramètres un entier naturel qui décroît strictement à chaque appel, la récursion s'arrêtant dès que cet entier passe sous un certain seuil
- **preuve de la validité du résultat** : en général par récurrence
 - Cas de base de la récursion
 - Cas inductif : on suppose que les appels récursifs sont corrects et on en déduit que l'appel courant est correct
- **calcul de complexité** : souvent à l'aide d'une relation de récurrence (la résolution de cette équation n'est généralement pas triviale)

Les lapins de Fibonacci



Leonardo Pisano dit Fibonacci rédige le **Liber abaci** vers 1202.

Possédant initialement un couple de lapins, combien de couples obtient-on en douze mois si chaque couple engendre tous les mois un nouveau couple à compter du second mois de son existence ?



Solution : la suite de Fibonacci $(F_n)_{n \geq 1}$

- $F_1 = 1$
- $F_2 = 1$
- $F_n = F_{n-1} + F_{n-2}, \quad \forall n \geq 3$

$$(F_n)_{n \geq 1} = \{1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, \dots\}$$

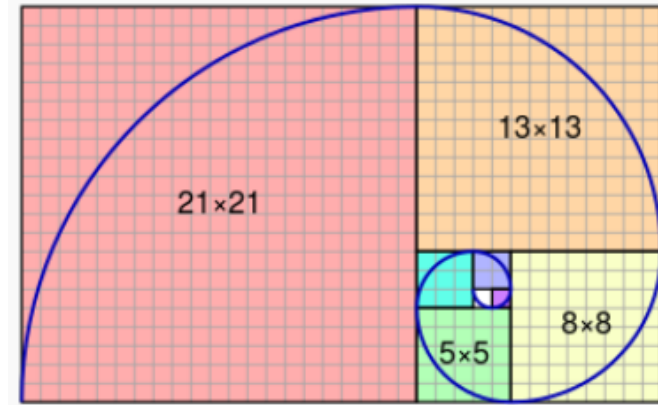
```
Require:  $n \geq 1$  : position dans la suite de Fibonacci
1: procedure Fibonacci( $n$ )
2:   if  $n \leq 2$  then
3:     return 1
4:   else
5:     return Fibonacci( $n - 1$ ) + Fibonacci( $n - 2$ )
6:   end if
7: end procedure
```

Proposition

$\forall n \geq 1$, Fibonacci(n) renvoie F_n , le n^{e} terme de la suite de Fibonacci

Démonstration :

- **Cas de base** : Fibonacci(1) = 1 et Fibonacci(2) = 1
- **Cas inductif** : $\forall n \geq 3$, Fibonacci(n) renvoie Fibonacci($n-1$) + Fibonacci($n-2$) = $F_{n-1} + F_{n-2}$ (par hypothèse inductive) = F_n



$$(F_n)_{n \geq 1} = \{1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \dots\}$$

Lorsqu'un algorithme contient un appel récursif à lui-même, son temps d'exécution peut souvent être décrit par une **équation de récurrence**. Elle exprime le temps d'exécution pour un problème de taille n en fonction du temps d'exécution pour des entrées plus petites.

Exemples

- $T(n) = a \times T(n - 1)$ (équation de récurrence linéaire d'ordre 1)
- $T(n) = a \times T(n - 1) - b \times T(n - 2)$ (équation de récurrence linéaire d'ordre 2)
- $T(n) = \sum_{i=1}^k a_k \times T(n - k)$ (équation de récurrence linéaire d'ordre k)
- $T(n) = a \times T(n - 1) + b$ (récurrence linéaire d'ordre 1 non homogène)
- $T(n) = a \times T(n/2) + g(n)$ (récurrence non linéaire et non homogène)

1. Conditions de base $T(0) = c, \dots$

Require: n : l'entier naturel dont on calcule la factorielle

```
1: procedure Factorielle(n)
2:   if  $n = 0$  then
3:     return 1
4:   else
5:     return  $n \times \text{Factorielle}(n - 1)$ 
6:   end if
7: end procedure
```

$$n! = ?$$

Équation réursive : $T(n) = 2 + T(n - 1)$ et $T(0) = 1$

Calcul de la complexité par une méthode de substitution

- **Intuition** : $T(n) \in \Theta(n)$
- **Hypothèse** : $T(n) = an + b$
 $T(n) = 2 + T(n - 1) = 2 + a(n - 1) + b = an + b \Rightarrow a = 2$
En utilisant $T(0) = a \times 0 + b = 1 \Rightarrow b = 1$
- **Conclusion** : $T(n) = 2n + 1 \in \Theta(n)$

Require: $n \geq 1$: position dans la suite de Fibonacci

```
1: procedure Fibonacci(n)
2:   if  $n \leq 2$  then
3:     return 1
4:   else
5:     return Fibonacci( $n - 1$ ) + Fibonacci( $n - 2$ )
6:   end if
7: end procedure
```

$$T(n) = 2 + T(n - 1) + T(n - 2), \quad \forall n \geq 3$$

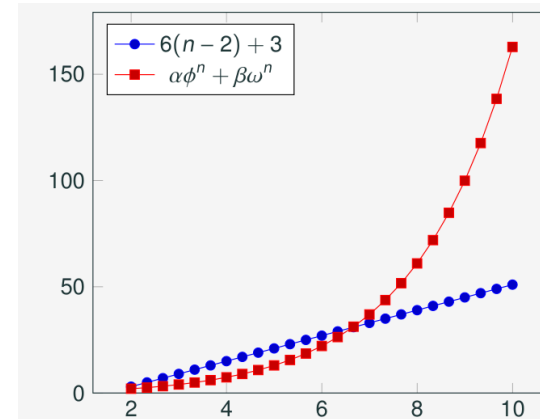
$$T(1) = T(2) = 1$$



Calcul de la complexité

- On pose $U(n) = T(n) + 2$ car -2 est le point fixe de la récurrence.
- $\forall n \geq 3, U(n) = U(n - 1) + U(n - 2)$ (suite récurrente linéaire d'ordre 2)
- Résolue par l'équation caractéristique : $r^2 = r + 1 \Rightarrow$ solutions : φ et ω
- $U(n) = \alpha\varphi^n + \beta\omega^n$ avec $\varphi = (1 + \sqrt{5})/2 \simeq 1.61803$ et $\omega = (1 - \sqrt{5})/2 \simeq -0.61803$
- On remarque que ω^n est décroissante et tend vers 0
- $\Rightarrow T(n) \in \Theta(\varphi^n)$
(φ est le nombre d'or)

```
Require:  $n \geq 1$  : position dans la suite
1: procedure Fibonacci( $n$ )
2:   if  $n \leq 2$  then           ▷ 1 instruction
3:     return 1
4:   else
5:      $a = b = 1$                ▷ 2 instructions
6:     for  $i = 3$  to  $n$  do       ▷ 2 instructions
7:        $c = b$                  ▷ 1 instruction
8:        $b = a + b$              ▷ 2 instructions
9:        $a = c$                  ▷ 1 instruction
10:    end for
11:    return  $b$ 
12:  end if
13: end procedure
```



donc la complexité peut s'évaluer par complexité en $\Theta(n)$.

1 (naturelle) de la version récursive.

Pour $n \geq 3$, il y a $(n - 2)$ tours de boucle, donc la complexité peut s'évaluer par $T(n) = 6(n - 2) + 3$. On obtient alors une complexité en $\Theta(n)$.

Cet exemple illustre un piège de l'utilisation (naturelle) de la version récursive.

Définition

On dit qu'une procédure est récursive terminale si la dernière instruction appelée est l'appel récursif.

Autrement dit, la valeur retournée est directement la valeur obtenue par un appel récursif, sans aucune opération sur cette valeur. Cela permet d'économiser l'espace de la pile d'exécution $\Rightarrow$ gain mémoire.

Récursivité terminale

```
1: procedure Algo1(n)
2:   ...
3:   return Algo1(n - 1)
4: end procedure
```

Récursivité non terminale

```
1: procedure Algo2(n)
2:   ...
3:   return 5 + Algo2(n - 1)
4: end procedure
```

Propriété

Quand une procédure est récursive terminale, on peut transformer l'appel récursif en une boucle itérative, sans utilisation de mémoire supplémentaire.

En plus du gain mémoire, la transformation en boucle itérative permet d'accélérer l'algorithme (coût de l'appel de fonction).

Appel récursif

```
1: procedure Algo1(n,a)
2:   if n = 0 then
3:     return a
4:   else
5:     return Algo1(n - 1, n × a)
6:   end if
7: end procedure
```

Boucle itérative

```
1: procedure Algo2(n,a)
2:   while n ≠ 0 do
3:     a = n × a
4:     n = n - 1
5:   end while
6:   return a
7: end procedure
```

Un certain nombre de langages de programmation peuvent détecter la récursivité terminale et l'optimiser en transformant automatiquement la récursion en itération.

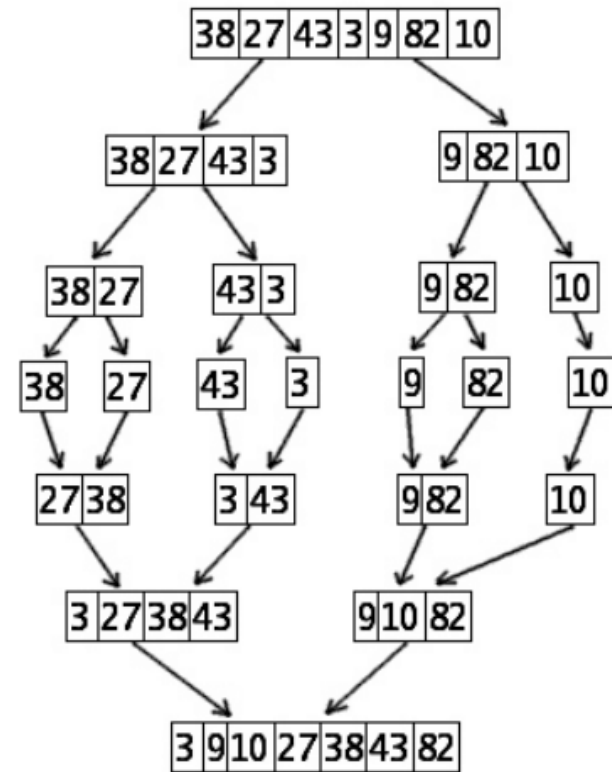
La stratégie de **diviser pour régner** est une méthode qui permet parfois de trouver des solutions efficaces à des problèmes algorithmiques. L'idée est de découper le problème initial, de taille n , en plusieurs sous-problèmes de taille sensiblement inférieure, puis de recombinaison les solutions partielles. Ce type d'algorithme se prête naturellement à une écriture récursive.

Ceci donne lieu à trois étapes à chaque niveau de récursivité :

- **Diviser** : le problème en un certain nombre de sous-problèmes
- **Régner** : résoudre les sous-problèmes récursivement — ou directement si leur taille est assez réduite
- **Combiner** : les solutions des sous-problèmes en une solution complète du problème initial.

L'exemple typique est l'algorithme **tri fusion** : pour trier un tableau de taille n , on le découpe en deux tableaux de taille $n/2$ et l'étape de fusion permet de recombinaison des deux solutions en $n - 1$ opérations.

```
Require: Tab : le tableau de taille n à trier
1: procedure TriFusion(Tab)
2:   if |Tab| ≤ 1 then
3:     return Tab
4:   else
5:     n = |Tab|
6:     Tab1 = TriFusion(Tab[0,...,[n/2]])
7:     Tab2 = TriFusion(Tab[[n/2]+1,...,n-1])
8:     return Fusion(Tab1,Tab2)
9:   end if
10: end procedure
```



Master Theorem (simplifié)

$T(n) = aT(n/b) + f(n)$, avec $a \geq 1, b > 1$

- n la taille du problème.
- a est le nombre de sous-problèmes.
- n/b est la taille de chaque sous-problème.
- $f(n)$ est le coût de la subdivision/combinaison des solutions des sous-problèmes. On suppose que $f(n) \in O(n^d)$.

$$T(n) = \begin{cases} \Theta(n^d) & \text{si } a < b^d \\ \Theta(n \log(n)) & \text{si } a = b^d \\ \Theta(n^{\log_b a}) & \text{si } a > b^d \end{cases}$$

Remarque : Si n/b n'est pas un entier, on le remplace par sa partie entière inférieure ou supérieure. On peut prouver que cela ne change pas l'expression asymptotique, et les sous-problèmes ont tous pratiquement la même taille.

```
Require: Tab1, Tab2 : deux tableaux triés
1: procedure Fusion(Tab1, Tab2)
2:   i = j = k = 0
3:   n1 = |Tab1|, n2 = |Tab2|
4:   TabFusion = createTab(n1+n2)
5:   while i < n1 and j < n2 do    ▷ au pire n1+n2-1
itérations
6:     if Tab1[i] < Tab2[j] then  ▷ 1 instruction
7:       TabFusion[k] = Tab1[i]  ▷ 1 instruction
8:       i = i+1; k = k+1        ▷ 2 instructions
9:     else
10:      TabFusion[k] = Tab2[j]   ▷ 1 instruction
11:      j = j+1; k = k+1        ▷ 2 instructions
12:    end if
13:  end while
14:  if i == n1 and j < n2 then
15:    TabFusion[k:n1+n2] = Tab2[j:n2-1]  ▷ au pire
n2 instructions
16:  else if j == n2 and i < n1 then
17:    TabFusion[k:n1+n2] = Tab1[i:n1-1]  ▷ au pire
n1 instructions
18:  end if
19:  return TabFusion
20: end procedure
```

La fonction $f(n)$ sera de la forme $f(n) = a \times (n_1 + n_2) + b$.
La complexité de l'algorithme Fusion est donc $O(n_1 + n_2) = O(n)$, c'est-à-dire linéaire.

```
Require: Tab : le tableau de taille n à trier
1: procedure TriFusion(Tab)
2:   if |Tab| ≤ 1 then                                ▷ 1
3:     return Tab
4:   else
5:     n = |Tab|                                       ▷ 1
6:     Tab1 = TriFusion(Tab[0,...,[n/2]]) ▷ 1
7:     Tab2 = TriFusion(Tab[[n/2]+1,...,n-1]) ▷ 1
8:     return Fusion(Tab1,Tab2)
9:   end if
10: end procedure
```

$$T(n) = 2T(n/2) + f(n) + 4$$

$$T(0) = T(1) = 1$$

On vient de voir que $f(n) \in O(n)$: on peut donc appliquer le Master théorème avec $a = 2, b = 2, d = 1$. On se retrouve dans le cas $a = b^d$ et on en déduit que la complexité du tri par fusion est $\Theta(n \log(n))$