La complexité des algorithmes

Amélie Lambert

Cnam

MPRO - Mise à niveau

Définition d'un algorithme

Procédure de calcul bien définie qui prend en entrée une valeur ou un ensemble de valeurs et qui donne en sortie une valeur ou un ensemble de valeurs, c'est donc une séquence d'étapes de calculs qui transforme l'entrée en sortie. (Cormen, Leiserson, Rivert)

Etapes de conception d'un algorithme

Phase 1 Énoncé du problème (Spécification)

Phase 2 Élaboration de l'algorithme

Phase 3 Vérification

Phase 4 Mesure de l'efficacité (Complexité)

Phase 5 Mise en oeuvre (Programmation)

Exemple : le tri d'un tableau

Énoncé du problème

- **Entrée :** Un tableau de n entiers $T = T[0], T[1], \ldots T[n-1]$, la taille du problème est donc n.
- **Sortie :** Permutation du tableau $T' = T'[0], T'[1], \ldots T'[n-1]$, tel que $T'[0] \leq T'[1] \leq \ldots \leq T'[n-1]$ (trié par ordre croissant).

Algorithme 1 : le tri par sélection (1/3)

Principe

TANT QUE il reste plus d'un élément non trié **faire**

1. Chercher le plus petit parmi les non triés.
2. Échanger le premier élément non trié avec le plus petit trouvé.

FAIT

Algorithme 1 : le tri par sélection (2/3)

minimum du sous-tableau restant à trier
éléments triés
éléments non triés

Étape 1 :

7	8	15	5	10

Algorithme 1 : le tri par sélection (2/3)

minimum du sous-tableau restant à trier
éléments triés
éléments non triés

Étape 1 :	7	8	15	5	10
Étape 2 :	5	8	15	7	10

Algorithme 1 : le tri par sélection (2/3)

minimum du sous-tableau restant à trier
éléments triés
éléments non triés

Étape 1 :	7	8	15	5	10
Étape 2 :	5	8	15	7	10
Étape 3 :	5	7	15	8	10

Algorithme 1 : le tri par sélection (2/3)

minimum du sous-tableau restant à trier
éléments triés
éléments non triés

Étape 1 :	7	8	15	5	10
Étape 2 :	5	8	15	7	10
Étape 3 :	5	7	15	8	10
Étape 4 :	5	7	8	15	10

Algorithme 1 : le tri par sélection (2/3)

minimum du sous-tableau restant à trier
éléments triés
éléments non triés

Étape 1 :	7	8	15	5	10
Étape 2 :	5	8	15	7	10
Étape 3 :	5	7	15	8	10
Étape 4 :	5	7	8	15	10
Étape 5 :	5	7	8	10	15

Algorithme 1 : le tri par sélection (2/3)

minimum du sous-tableau restant à trier
éléments triés
éléments non triés

Étape 1 :	7	8	15	5	10
Étape 2 :	5	8	15	7	10
Étape 3 :	5	7	15	8	10
Étape 4 :	5	7	8	15	10
Étape 5 :	5	7	8	10	15
Étape 6 :	5	7	8	10	15

Algorithme 1 : le tri par sélection (3/3)

```
static void tri_selection(int [] T) {
   int i,ind_min,k;
   int temp;
    for (i=0;i<n−2;i++) {
          ind_min = i;
          for (k= i+1; k < n ; k++) {
                si (T[k] < T[ind_min])
                    ind_min = k;
          }
          temp = T[i];
          T[i] = T[ind_min];
          T[ind_min] = temp;
    }
}
```

Évaluation d'un algorithme (1/2)

Mesure d'efficacité par rapport à la taille de l'entrée, 2 critères :

- Temps d'exécution : nombre d'opérations élémentaires
- Taille : mémoire occupée

Ces mesures varient en fonction des données d'entrée de l'algorithme.

Exemple (temps d'exécution) : Déterminer si un entier n est premier ?

- Si $n = 1148$? non, divisible par 2
- Si $n = 1147$? ? ? ? $(31 * 37)$

Évaluation d'un algorithme (2/2)

Efficacité =
nombre d'opérations en fonction de la taille des données

On peut l'évaluer dans 3 cas :

- Dans le meilleur des cas (peu d'intérêt),
- Dans le pire des cas,
- En moyenne.

Exemples

- **Exemple 2 : recherche des deux villes les plus proches (N villes)**
  - **Problème :** Étant donné un ensemble de villes séparées par des distances données, trouver les deux villes les plus proches.
  - **Données** distances entre chaque paire de villes : N^2 données.
  - **Énumération et recherche du minimum** N^2 distances possibles.

- **Exemple 3 : le voyageur de commerce (N villes)**
  - **Problème :** Étant donné un ensemble de villes séparées par des distances données, trouver le plus court chemin qui relie toutes les villes.
  - **Données :** distances entre chaque paire de villes : N^2 données.
  - **Énumération et évaluation des** $(N-1)!$ tournées possibles et sélection de la plus économique.

Comparaison

Temps de calcul si 10^{-9} secondes par comparaison

Nombre de villes N	10	20	30	40	50
Nb données N^2	100	400	900	1600	2500
Min de N^2 nb	0.1 μ s	0.4μ s	0.9μ s	1.6 μ s	2.5 μ s
Min de $(N-1)!$ nb	363 μ s	$>$ 3 ans	10^{16} ans	10^{30} ans	10^{45} ans

Complexité des algorithmes

- La notion de complexité formalise la notion d'efficacité d'un programme : capacité à fournir le résultat attendu dans un temps minimal
- Elle consite en la détermination des ressources nécéssaires en temps de calcul, mémoire.
- On se donne un modèle de machine avec une mémoire infinie et munie d'opérations dont le temps d'exécution (coût élémentaire) est constant (ou majoré par une constante).
- On cherche à exprimer le coût de l'algorithme en fonction de la taille des données.

Notations

- L'ensemble $D_n = \{\text{données de taille } n\}$

  L'exécution de l'algorithme sur une donnée d est modélisée par une séquence finie d'opérations élémentaires de la machine : addition, multiplication, test, ...

- le coût de l'exécution $\text{cout}_A(d)$ = nombre d'exécution de chaque opération de l'algorithme A sur la donnée d

  En général, on restreint le nombre d'opérations élémentaires étudiées

Complexité dans le pire des cas

$$\max_A(n) = \max_{d \in D_n} \text{cout}_A(d)$$

- Borne supérieure du coût,
- Assez facile à calculer,
- Souvent réaliste.

Complexité en moyenne

- p (d) = probabilité de la donnée d

- $\mathrm{Moy}_A(n) = \sum_{d \in D_n} p(d) * \mathrm{cout}_A(d)$
  - Connaître la distribution des données,
  - Souvent difficile à calculer,
  - Intéressant si le comportement usuel de l'algorithme éloigné du pire des cas.

Evaluation de la complexité d'un algorithme

- Coût d'une instruction conditionnelle $\leq$ maximum des coûts de chacune des branches de l'instruction conditionnelle ;
- Coût d'un appel de procédure $=$ somme du coût de l'appel (i.e. coût du corps de la procédure) ;
- Algorithme itératif : coût d'une itération $=$ somme des coûts de chaque terme de l'itération ;
- Algorithme récursif : on exprime le coût de l'algorithme en fonction du coût des appels récursifs. On obtient ainsi une équation de récurrence sur la fonction de coût.

Exemple : Le tri par sélection

```
static void tri_selection(int [] T) {
   int i,ind_min,k;
   int temp;
    for (i=0;i<n−2;i++) {
          ind_min = i;
          for (k= i+1; k < n ; k++) {
                si (T[k] < T[ind_min])
                    ind_min=k;
          }
          temp = T[i];
          T[i] = T[ind_min];
          T[ind_min] = temp;
    }
}
```

Pire des cas :

$$4(n-2) + 2\sum_{i=0}^{n-2}(n-i-1)$$
$$= 4(n-2) + 2\sum_{i=1}^{n-1} i$$
$$= 4(n-2) + n(n-1)$$
$$= n^2 + 3n - 8$$

Rappel

$$\sum_{i=1}^{n-1} i = n(n-1)/2$$

Exemple 2 : produit de 2 matrices

```
static float[][] prod_mat(float [][] A, float [][] B) {
   // A[m,n] , B[n,p], résultat C[m,p]
   float [][] C = new float[m][p];
   int i,j,k;
   float s;
   for (i=0; i< m; i++) {
        for (j=0; j< p; j++) {
             s = 0;
             for (k=0; j< n; j++) {
                  s = s + A[i][k] * B[k][j]; }
             C[i][j] = s ;
        }
   }
}
```

pire cas = cas moyen = mp(2n+2) opérations

Comportement asymptotique

- Ce qui nous intéresse est le comportement asymptotique en fonction de n
- Pour cela on compare la fonction de coût à des fonctions de référence Typiquement les fonctions polynomiales : n^a, puissance x^n , exponentielles 2^n ou logarithmiques $log_2(n)$
- Pour exprimer la comparaison asymptotique on utilise la notation de Landau

Borne supérieure asymptotique : fonction $\mathcal{O}$

- f, g fonction $\mathbb{N} \to \mathbb{N}$

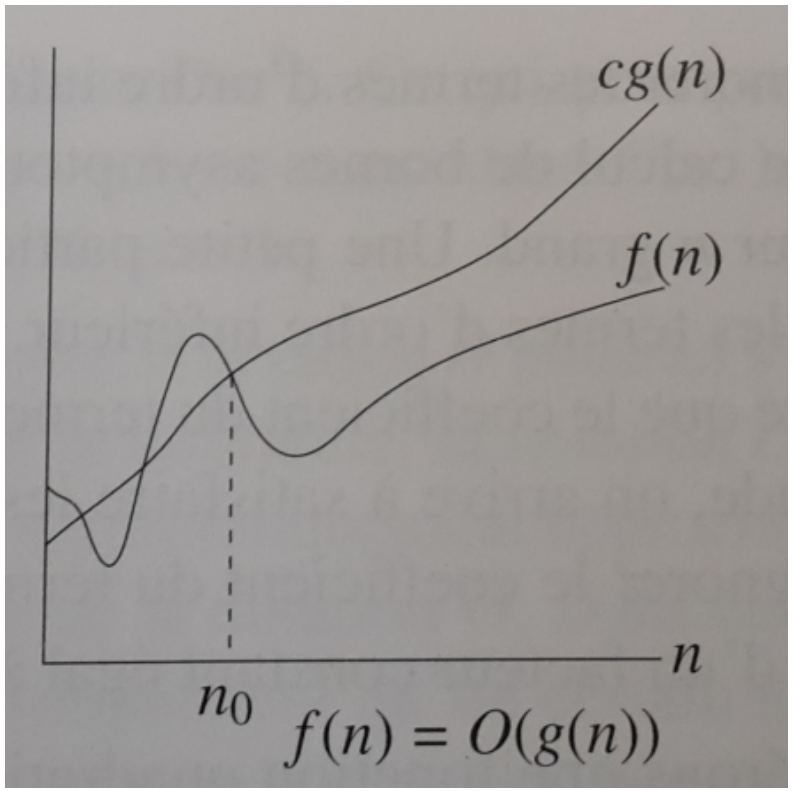
- **Définition :** Pour une fonction donnée g on note $\mathcal{O}(g)$ l'ensemble des fonctions :

$$\mathcal{O}(g) = \{f \text{ telles que } \exists c \geq 0, \exists n_0 \geq 0, \forall n \geq n_0, f(n) \leq cg(n)\}$$

on écrit $f = \mathcal{O}(g)$ pour $f \in \mathcal{O}(g)$ et on dit f est en "grand O" de g. (intuitivement : à partir d'un certain rang, f ne croît pas plus vite que g).

Exemple :
On a $f(n) = 4n^3 + 2n + 1$ opérations et $\forall n \geq 1$, $4n^3 + 2n + 1 \leq 7n^3$
Si on prend $n_0 = 1$, $c = 7$, et $g(n) = n^3$, on montre que
$f(n) = \mathcal{O}(g(n)) = \mathcal{O}(n^3)$

fonction $\mathcal{O}$: majorant du nombre d'opérations d'un algorithme



$$f(n) = \mathcal{O}(g(n))$$

Borne inférieure asymptotique : fonction Ω

- f, g fonction $\mathbb{N} \to \mathbb{N}$

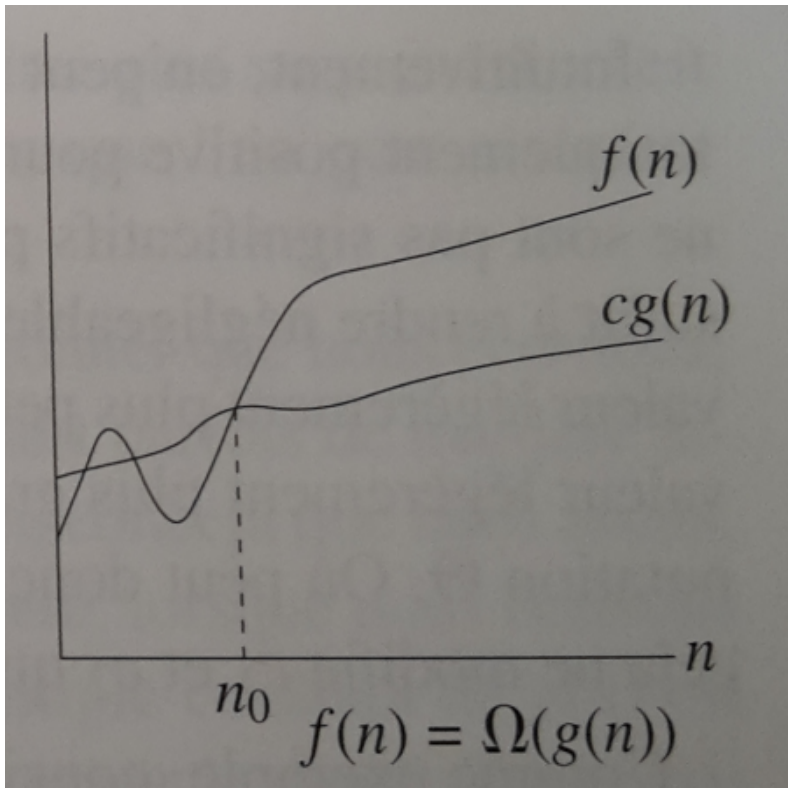
- **Définition :** Pour une fonction donnée g on note $\Omega(g)$ l'ensemble des fonctions :

$$\Omega(g) = \{f \text{ telles que } \exists c \geq 0, \exists n_0 \geq 0, \forall n \geq n_0, cg(n) \leq f(n)\}$$

on écrit $f = \Omega(g)$ pour $f \in \Omega(g)$ et on dit f est en Omega de g.
(intuitivement : à partir d'un certain rang, f croît plus vite que g).

Exemple :
On a $f(n) = 4n^3 + 2n + 1$ opérations et $\forall n \geq 1,\ n \leq f(n)$
Si on prend $n_0 = 1$, $c = 1$, et $g(n) = n$, on montre que
$f(n) = \Omega(g(n)) = \Omega(n)$

fonction Ω : minorant du nombre d'opérations d'un algorithme



$$f(n) = \Omega(g(n))$$

Borne asymptotique approchée : fonction Θ

- f, g fonction $\mathbb{N} \to \mathbb{N}$

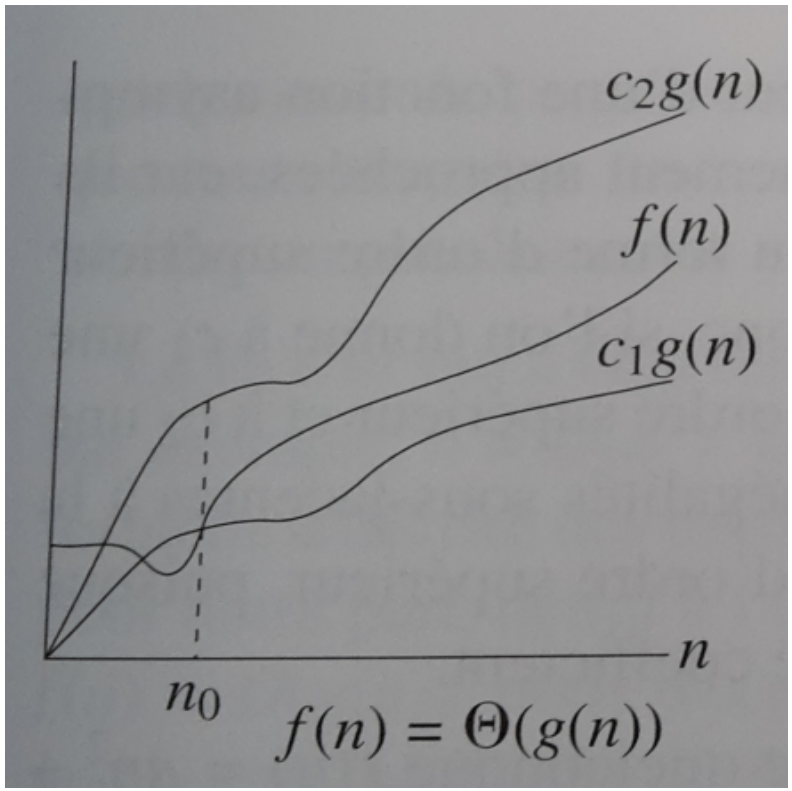
- **Définition :** Pour une fonction donnée g on note $\Theta(g)$ l'ensemble des fonctions :

  $$\Theta(g) = \{f \text{ tq } \exists c_1, c_2 \geq 0, \exists n_0 \geq 0, \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

  on écrit $f = \Theta(g)$ pour $f \in \Theta(g)$ et on dit f est en theta de g. (intuitivement : à partir d'un certain rang, f croît de la même façon que g).

Exemple : Prenons $f(n) = \frac{1}{2}n^2 - 3n$, on a :
$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2 \Rightarrow c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$ et $\forall n \geq 7 : \frac{1}{14} \leq \frac{1}{2} - \frac{3}{n} \leq \frac{1}{2}$
Si on prend $n_0 = 7$, $c_1 = \frac{1}{14}$, $c_2 = \frac{1}{2}$, et $g(n) = n^2$, on montre que $f(n) = \Theta(g(n)) = \Theta(n^2)$

fonction Θ : fonction approchée du nombre d'opérations d'un algorithme



$$f(n) = \Theta(g(n))$$

Caractérisation de la complexité

- Comportement des algorithmes quand le nombre de données augmente,
- Comparaison entre algorithmes.

Algorithme polynômial :

$$f(n) = a_0 n^0 + a_1 n^1 + \ldots + a_{p-1} n^{p-1} + a_p n^p$$
$$f(n) = \mathcal{O}(n^p)$$

Exemple :

Pour le `tri-sélection` on a : $f(n) = n^2 + 3n - 4$

C'est un algorithme polynômial en $\mathcal{O}(n^2)$.

Classification de la complexité des algorithmes

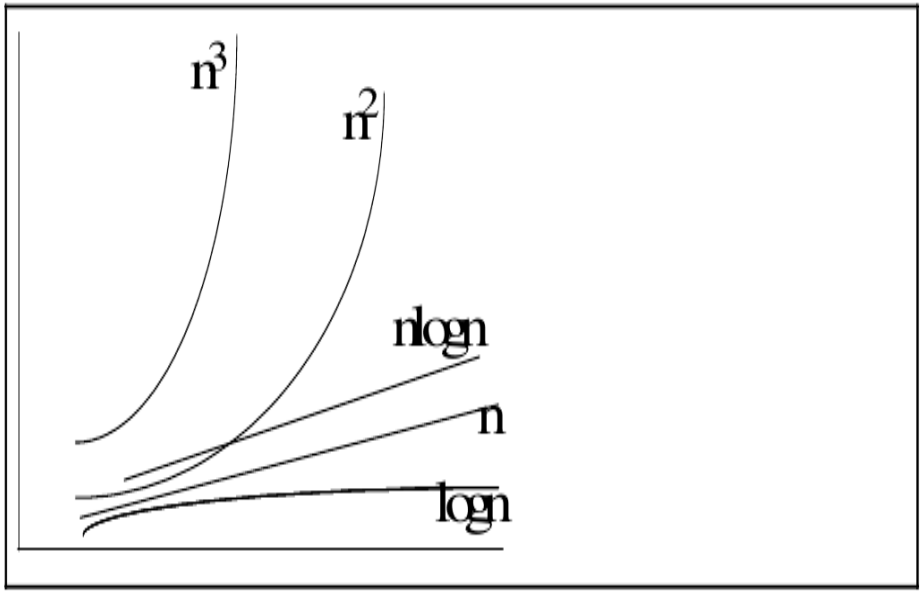
- **Algorithme "efficace" = Algorithme polynômial**

  Exemple : $n^2 + 3n - 4$

- **Problème "intraitable" : théorie de la complexité**

  Exemple : Voyageur de commerce

Une "bonne complexité"



$n = 10^6$ et 1μs par opération :

$log_2(n)$	n	$nlog_2(n)$	n^2	n^3
20 μs	1 s	20	12 j	32 Ka

Algorithme 2 : le tri par insertion (tri d'un jeu de carte)

Principe :

POUR chaque élément i du tableau **faire**

1. Trouver la place j de l'élément i parmi les éléments qui le précède.
2. Décaler les éléments de j à i d'une case.
3. Placer l'élément i à sa place j.

FAIT

Algorithme 2 : Le tri par insertion

T[i] courant
éléments triés
éléments non triés

Étape 1 :

7	8	15	5	10

place de T[1] = 1

Algorithme 2 : Le tri par insertion

T[i] courant
éléments triés
éléments non triés

Étape 1 :

7	8	15	5	10

place de T[1] = 1

Étape 2 :

7	8	15	5	10

place de T[2] = 2

Algorithme 2 : Le tri par insertion

T[i] courant
éléments triés
éléments non triés

Étape 1 :	7	8	15	5	10	place de T[1] = 1
Étape 2 :	7	8	15	5	10	place de T[2] = 2
Étape 3 :	7	8	15	5	10	place de T[3] = 0

Algorithme 2 : Le tri par insertion

T[i] courant
éléments triés
éléments non triés

Étape 1 :	7	8	15	5	10	place de T[1] = 1
Étape 2 :	7	8	15	5	10	place de T[2] = 2
Étape 3 :	7	8	15	5	10	place de T[3] = 0
Étape 4 :	5	7	8	15	10	place de T[4] = 3

Algorithme 2 : Le tri par insertion

T[i] courant
éléments triés
éléments non triés

Étape 1 :	7	8	15	5	10	place de T[1] = 1
Étape 2 :	7	8	15	5	10	place de T[2] = 2
Étape 3 :	7	8	15	5	10	place de T[3] = 0
Étape 4 :	5	7	8	15	10	place de T[4] = 3
Étape 5 :	5	7	8	10	15	

Algorithme 2 : Le tri par insertion

```
static void tri_insertion (int [] T) {
 int cle,i,j;
 for (i=1;i<n;j++) {
      cle = T[i];
      j= i−1;
      while (j>0) && (T[j] > cle) {
            T[j+1] = T[j];
            j= j−1;
      }
      T[j+1] = cle;
  }
}
```

Algorithme 2 : La complexité du tri par insertion

Complexité au pire et en moyenne

Si l'insertion du p^{ieme} élément $\Longrightarrow$ le décalage des $p-1$ éléments qui le précèdent :

1. La boucle `while` effectue au plus $n-1$ itérations.
2. La complexité de cette boucle est donc en $\mathcal{O}(n)$.
3. La boucle `for` effectue $n-1$ itérations.
4. Complexité au pire égale à $(n-1)*(n-1) = n^2 - 2n + 1 = \mathcal{O}(n^2)$.

Complexité au mieux

Cas où la boucle `while` a une complexité constante (ex : $T = 2, 1, 3, 4$)
Complexité au mieux égale à $(n-1) = \mathcal{O}(n)$.

Algorithme 3 : Le tri par fusion

Principe :

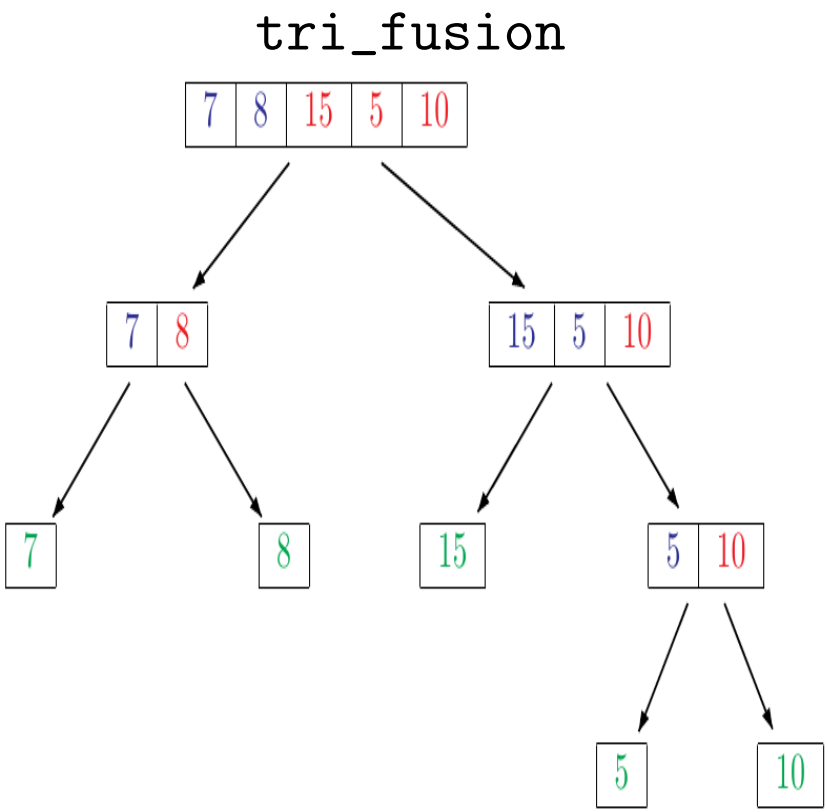
- Algorithme récursif.
- **TANT** c'est possible **faire**
  1. Diviser le tableau en deux parties équilibrées.
  2. Trier indépendemment chaque partie.
  3. Fusionner les sous parties triées.

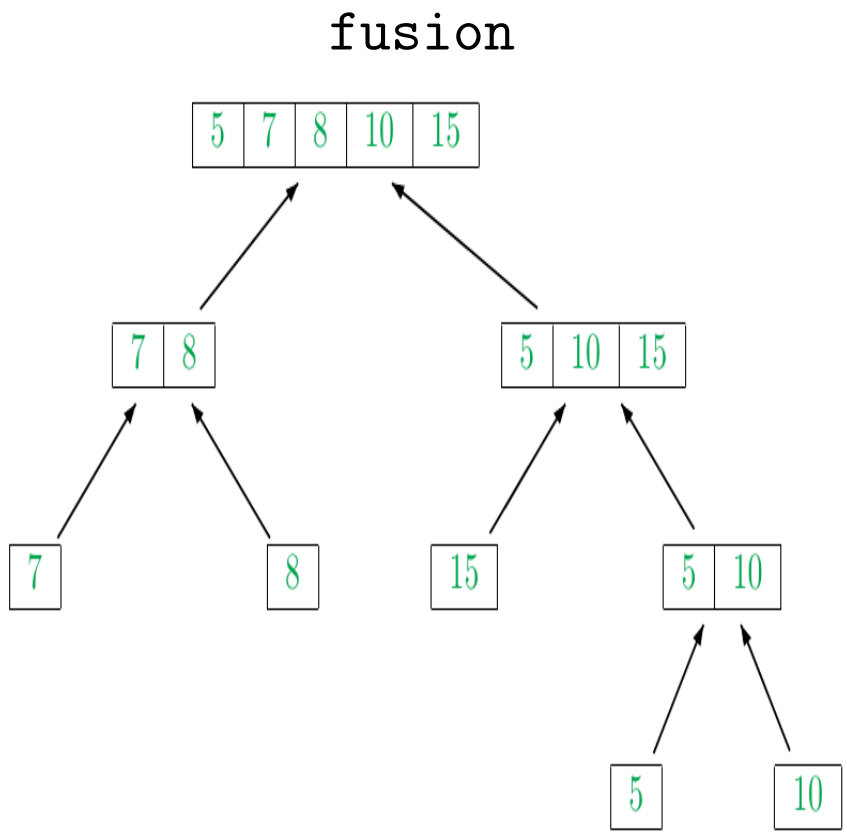
  FAIT

- La difficulté résulte de la fusion, car le tri se fait instantanément sur des parties suffisamment petites (de taille 1 par exemple).
- Il faut donc une procédure de fusion qui ait une bonne complexité.

Algorithme 3 : Le tri par fusion

éléments entre d et m
éléments entre m et f
éléments triés





Algorithme 3 : Le tri par fusion

```
static void tri_fusion (int[] T,int d,int f) {
   int m;
   if (d < f) {
        m = (d + f)/2;
        tri_fusion(T,d,m);
        tri_fusion(T,m+1,f);
        fusion (T,d,m,f); }
}
```

Algorithme 3 : Le tri par fusion

```
public static void fusion (int tab[], int deb1, int fin1, int fin2){
  int [] temp=new int[fin1-deb1+1];
  for (int i=deb1;i<=fin1;i++)
    temp[i-deb1]=T[i];
  int i1 = deb1; //indice pour la 1ere partie
  int i2 = fin1 + 1; //indice pour la 2eme partie
  for(int i=deb1;i<=fin2;i++){
    if (i1==deb2) //on a fini de trier le 1er tableau
      break; //T est trié
      else {
        if (i2==(fin2+1)) { //on a fini de trier le 2e tableau
          T[i]=temp[i1-deb1]; //on copie la fin du 1er tableau
          i1++;}
        else {
          if (temp[i1-deb1]<T[i2]){//le min est dans le 1er tableau
            T[i]=temp[i1-deb1];
            i1++; }
          else {//le min est dans le 2e tableau
            T[i]=T[i2];
            i2++; }
        }
      }
  }
}
```

Algorithme 3 : Complexité du tri fusion

Complexité de la fonction `fusion`

1. La première boucle `for` effectue $fin1 - deb1 + 1$ itérations.
2. La deuxième boucle `for` effectue $fin2 - deb1 + 1$ itérations.
3. Au maximum pour un tableau de taille n, on a $fin2 - deb1 + 1 = n + 1$ opérations.
4. Complexité au pire égale à $\mathcal{O}(n)$.

Complexité de la fonction `tri_fusion`

1. Nombre d'appels à la fonction récursive `tri_fusion` : à chaque fois des problèmes de tailles divisées par 2 jusqu'à la taille 1 : $((((n/2)/2)/2)/\ldots/2) = 1$, donc $\frac{n}{2^k} = 1$ et ainsi $k = log_2 n$.
2. Complexité au pire égale à $\mathcal{O}(log_2(n))$.

Complexité du tri fusion : $\mathcal{O}(nlog_2(n))$.

Exemple 2 : Recherche d'un élément dans un tableau

Énoncé du problème

- **Entrée :** Un tableau de n entiers $T = T[0], T[1], \ldots T[n-1]$, tel que $T = T[0] \leq T[1] \leq \ldots \leq T[n-1]$, la taille du problème est donc n.
- **Sortie :** VRAI si l'élément x est présent dans T, FAUX sinon.

Algorithme 1 : Recherche séquentielle

Principe :

- **POUR** chaque élément $T[i]$ du tableau **faire**

  1. Si $T[i] = x$ s'arrêter, répondre VRAI
  2. Si $T[i] > x$ s'arrêter, répondre FAUX

FAIT

Algorithme 1 : Recherche séquentielle

```
static boolean recherche_lin (int x, int [] T){
   boolean rep=false;
   for (int i=0 ; i < T.length && (rep==false); i++){
      if (T[i]==x)
         rep=true;
      if (T[i]>x)
         break;
   }

   return rep;
}
```

Complexité de la recherche séquentielle

Complexité au pire

1. La boucle `for` effectue au plus n itérations.
2. Complexité au pire égale à $\mathcal{O}(n)$

Algorithme 2 : Recherche dichotomique

Principe :

- Algorithme récursif.
- Si x est dans T alors il est dans l'intervalle d'indices $[d \dots f[$; où d initialisé à 0 et f à n (après la fin de T).
- On compare x par rapport à $T[m]$ où m est l'indice du milieu du tableau.
- **TANT QUE** on n'a pas trouvé x et que $f - d > 0$ **faire**
  1. Si $x = T[m]$ on s'arrête car on a trouvé x.
  2. Si $T[m] < x$ on en déduit que x ne peut pas se trouver dans l'intervalle $d \dots m$; on déplace l'indice d.
  3. Sinon ($T[m] > x$) on en déduit que x ne peut pas se trouver dans l'intervalle $m \dots f$; on déplace l'indice f.

FAIT

Algorithme 2 : Recherche dichotomique

```
static boolean recherche_dicho (int x, int [] T) {
  int d=0;
  int f = T.length;//si x est dans le tableau, il le serait dans [d..f[ (f exclu)
  int m;
  boolean trouve = false;

  while ((trouve == false) && (d<f)){
    m = (d+f)/2;
    if (T[m]==x)
      trouve=true;
    else
      if (T[m] < x)
        d=m+1;
      else
        f=m;
  }
  return trouve;
}
```

Complexité de la recherche dichotomique

Complexité au pire

Soit k le nombre de passages dans la boucle `while`. On divise le nombre d'éléments restants par 2 jusqu'à ce qu'il n'en reste qu'un (k divisions) :

1. $((((n/2)/2)/2)/\ldots/2) = 1$.
2. Donc $\frac{n}{2^k} = 1$ et ainsi $k = log_2 n$.
3. Complexité au pire égale à $\mathcal{O}(log_2(n))$.