Algorithmique et structure de données

Récursivité et tableaux

Rohan Fossé

30 Septembre 2021

Avant de commencer

Supports de cours

- Site Web : labri.fr/perso/rfosse
- Enoncé et Correction des EI
- Exercices supplémentaires

Rappel de la dernière séance

Complexité d'un algorithme

- La complexité permet d'évaluer l'efficacité d'un algorithme ;
- Elle permet de comparer des algorithmes indépendamment des machines ;
- On note $\mathcal{O}$ la complexité asymptotique (*dans le pire cas*).

Algorithmes Récursifs

Introduction

- Les **algorithmes récursifs** et les **fonctions récursives** sont fondamentaux en informatique. Un algorithme est dit récursif s'il s'appelle **lui-même** ;
- Les premiers langages de programmation qui ont introduit la récursivité sont `LISP` et `Algol 60` et maintenant tous les langages de programmation modernes proposent une implémentation de la récursivité ;
- On oppose généralement les algorithmes récursifs aux algorithmes dits impératifs ou itératifs qui s'exécutent sans invoquer ou appeler explicitement l'algorithme lui-même.

Lien avec les mathématiques : la factorielle

On souhaite calculer $n!$. On rappelle que pour tout entier $n \geq 0$

$$n! = \prod_{i=1}^{n} i = 1 \times 2 \times \ldots \times n$$

Lien avec les mathématiques : la factorielle

On souhaite calculer $n!$. On rappelle que pour tout entier $n \geq 0$

$$n! = \prod_{i=1}^{n} i = 1 \times 2 \times \ldots \times n$$

On peut déduire de cette définition la propriété importante suivante

$$\forall\, n \geq 1, n! = n \times (n-1)!$$

et donc si on sait calculer (n-1) ! alors on sait calculer n !.

Lien avec les mathématiques : la factorielle

On souhaite calculer $n!$. On rappelle que pour tout entier $n \geq 0$

$$n! = \prod_{i=1}^{n} i = 1 \times 2 \times \ldots \times n$$

On peut déduire de cette définition la propriété importante suivante

$$\forall n \geq 1, n! = n \times (n-1)!$$

et donc si on sait calculer (n-1)! alors on sait calculer n!.

Par ailleurs, on sait que $0! = 1$. On sait donc calculer $1!$, puis $2!$, et par récurrence on peut établir qu'on sait calculer n! pour tout entier $n \geq 0$.

Algorithme de la factorielle

L'algorithme récursif de calcul de la factorielle distingue deux cas. Le premier cas ne nécessite aucun calcul, le second utilise la fonction *fact* pour calculer $(n-1)!$

```
function fact(n)
    if  n == 0  then
        return 1
    else
        return n × fact(n − 1)
```

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$fact(4) \rightarrow 4 \times fact(3)$$

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$fact(4) \rightarrow 4 \times fact(3)$$

$$fact(4) \rightarrow 4 \times 3 \times fact(2)$$

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$fact(4) \rightarrow 4 \times fact(3)$$
$$fact(4) \rightarrow 4 \times 3 \times fact(2)$$
$$fact(4) \rightarrow 4 \times 3 \times 2 \times fact(1)$$

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$fact(4) \rightarrow 4 \times fact(3)$$

$$fact(4) \rightarrow 4 \times 3 \times fact(2)$$

$$fact(4) \rightarrow 4 \times 3 \times 2 \times fact(1)$$

$$fact(4) \rightarrow 4 \times 3 \times 2 \times 1 \times fact(0)$$

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$fact(4) \rightarrow 4 \times fact(3)$$
$$fact(4) \rightarrow 4 \times 3 \times fact(2)$$
$$fact(4) \rightarrow 4 \times 3 \times 2 \times fact(1)$$
$$fact(4) \rightarrow 4 \times 3 \times 2 \times 1 \times fact(0)$$
$$fact(4) \rightarrow 4 \times 3 \times 2 \times 1 \times 1$$

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$fact(4) \rightarrow 4 \times fact(3)$$
$$fact(4) \rightarrow 4 \times 3 \times fact(2)$$
$$fact(4) \rightarrow 4 \times 3 \times 2 \times fact(1)$$
$$fact(4) \rightarrow 4 \times 3 \times 2 \times 1 \times fact(0)$$
$$fact(4) \rightarrow 4 \times 3 \times 2 \times 1 \times 1$$
$$fact(4) \rightarrow 24$$

Autre algorithme de factorielle

Considérons l'algorithme suivant :

function $fact2(n)$
  return $n \times fact(n-1)$

Autre algorithme de factorielle

Considérons l'algorithme suivant :

function $fact2(n)$
    return $n \times fact(n-1)$

Quel est le soucis avec cette fonction ?

Autre algorithme de factorielle

Considérons l'algorithme suivant :

function $fact2(n)$
    return $n \times fact(n-1)$

Quel est le soucis avec cette fonction ?

L'évaluation de $fact2(1)$ conduit à un calcul infini :

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$fact2(1) \rightarrow 1 \times fact2(0)$$

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$fact2(1) \rightarrow 1 \times fact2(0)$$

$$fact2(1) \rightarrow 1 \times 0 \times fact2(-1)$$

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$fact2(1) \rightarrow 1 \times fact2(0)$$

$$fact2(1) \rightarrow 1 \times 0 \times fact2(-1)$$

$$fact2(1) \rightarrow 1 \times 0 \times -1 \times fact2(-2)$$

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$
\begin{aligned}
&fact2(1) \rightarrow 1 \times fact2(0)\\
&fact2(1) \rightarrow 1 \times 0 \times fact2(-1)\\
&fact2(1) \rightarrow 1 \times 0 \times -1 \times fact2(-2)\\
&fact2(1) \rightarrow \ldots
\end{aligned}
$$

Règle implicite de conception d'un algorithme récursif

Règle implicite

Tout algorithme récursif doit distinguer plusieurs cas dont l'un au moins ne doit pas contenir d'appels récursifs.

Les cas non récursifs d'un algorithme récursif sont appelés *cas de bases*. Les conditions que doivent satisfaire les données dans ces cas de bases sont appelées *conditions de terminaison*.

Règle implicite de conception d'un algorithme récursif

Règle implicite

Tout algorithme récursif doit distinguer plusieurs cas dont l'un au moins ne doit pas contenir d'appels récursifs.

Les cas non récursifs d'un algorithme récursif sont appelés *cas de bases*. Les conditions que doivent satisfaire les données dans ces cas de bases sont appelées *conditions de terminaison*.

Est-ce-que c'est suffisant ?

Reprenons notre exemple de la factorielle

```
function fact3(n)
    if n == 0 then
        return 1
    else
        return n × fact3(n + 1)
```

Reprenons notre exemple de la factorielle

```
function fact3(n)
    if n == 0 then
        return 1
    else
        return n × fact3(n + 1)
```

Quel est le soucis avec cette fonction ?

Reprenons notre exemple de la factorielle

```
function fact3(n)
    if n == 0 then
        return 1
    else
        return n × fact3(n + 1)
```

Quel est le soucis avec cette fonction ?

L'évaluation de $fact3(1)$ conduit à un calcul infini :

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$fact3(1) \rightarrow 1 \times fact3(2)$$

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$fact3(1) \rightarrow 1 \times fact3(2)$$

$$fact31 \rightarrow 1 \times 2 \times fact3(3)$$

Exécution de l'algorithme de la factorielle

Prenons l'exemple du déroulement du calcul récursif de 4! :

$$fact3(1) \rightarrow 1 \times fact3(2)$$

$$fact31 \rightarrow 1 \times 2 \times fact3(3)$$

$$fact31 \rightarrow 1 \times 2 \times 3 \times \ldots$$

Deuxième règle implicite

```
function fact3(n)
    if  n == 0  then
        return 1
    else
        return n × fact3(n + 1)
```

Deuxième règle implicite

```
function fact3(n)
    if n == 0 then
        return 1
    else
        return n × fact3(n + 1)
```

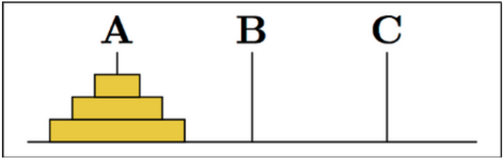
Deuxième règle

À chaque appel récursif il faut se rapprocher des conditions de terminaison.

Un autre exemple : les tours de Hanoï

Principe du jeu

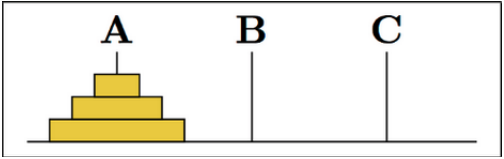
- On dispose de 3 tours côte à côte sur lesquelles on peut empiler des disques de diamètre croissant ;
- On ne peut déplacer qu'un disque à la fois et on ne doit pas poser un disque plus large sur un plus petit ;
- L'objectif est de trouver la suite de déplacements qui permet de placer tous les disques sur la tour la plus à droite.



Un autre exemple : les tours de Hanoï

Principe du jeu

- On dispose de 3 tours côte à côte sur lesquelles on peut empiler des disques de diamètre croissant ;
- On ne peut déplacer qu'un disque à la fois et on ne doit pas poser un disque plus large sur un plus petit ;
- L'objectif est de trouver la suite de déplacements qui permet de placer tous les disques sur la tour la plus à droite.

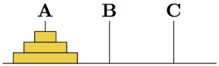
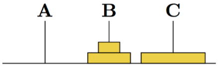
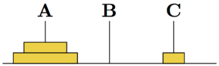
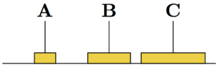
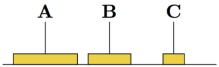
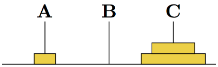
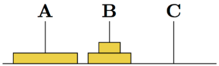
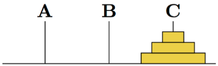


À vous de jouer ! En combien de mouvements est-ce possible ?

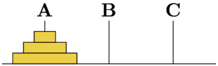
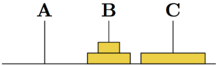
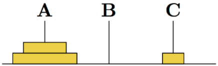
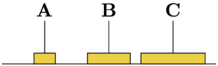
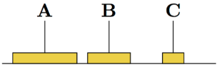
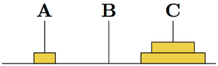
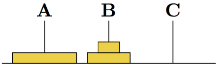
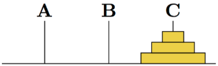
Résolution du problème de Hanoï

La réponse : 7 !

La réponse : 7 !

Mouvement	Position	Mouvement	Position
Position initiale		4 : **A** vers **C**	
1 : **A** vers **C**		5 : **B** vers **A**	
2 : **A** vers **B**		6 : **B** vers **C**	
3 : **C** vers **B**		7 : **A** vers **C**	

La réponse : 7 !

Mouvement	Position	Mouvement	Position
Position initiale		4 : **A** vers **C**	
1 : **A** vers **C**		5 : **B** vers **A**	
2 : **A** vers **B**		6 : **B** vers **C**	
3 : **C** vers **B**		7 : **A** vers **C**	

Quel est l'algorithme ?

Algorithme de résolution des tours de Hanoï dans le cas général

Cet algorithme est la fonction *hanoi* qui prend trois paramètres :

1. n le nombre de disques à déplacer ;
2. d la tour de départ où se trouvent ces disques ;
3. a la tour d'arrivée où doivent aller les disques

On suppose que l'on a une fonction $\textit{deplacerdisque}(d, a)$ qui permet de déplacer un disque de d jusqu'à a.

Algorithme de résolution des tours de Hanoï dans le cas général

Cet algorithme est la fonction *hanoi* qui prend trois paramètres :

1. n le nombre de disques à déplacer ;
2. d la tour de départ où se trouvent ces disques ;
3. a la tour d'arrivée où doivent aller les disques

On suppose que l'on a une fonction $deplacerdisque(d, a)$ qui permet de déplacer un disque de d jusqu'à a.

```
function hanoi(n, d, a)
    if  n > 0 then
        aux ← le piquet différent de d et a
        hanoi(n − 1, d, aux)
        deplacerdisque(d, a)
        hanoi(n − 1, aux, a)
```

Petit exercice

Exercice

Écrire une fonction qui calcule la somme des inverses des carrés des n premiers entiers naturels non nuls.

Petit exercice

Exercice

Écrire une fonction qui calcule la somme des inverses des carrés des n premiers entiers naturels non nuls.

```
function sum_inv(n)
    if  n == 0 then
        return 1
    else
        return 1/n ∗ ∗2 + sum_inv(n − 1)
```

Type de récursivité

Récursivité simple ou linéaire

Un algorithme récursif est *simple* ou *linéaire* si chaque cas qu'il distingue se résout en au plus un appel récursif.

```
function mystere(n)
    if n == 0 then
        return 1
    else
        return mystere(n − 1) × 2
```

Récursivité simple ou linéaire

Un algorithme récursif est *simple* ou *linéaire* si chaque cas qu'il distingue se résout en au plus un appel récursif.

```
function mystere(n)
    if  n == 0 then
        return 1
    else
        return mystere(n − 1) × 2
```

Que fait cet algorithme ?

Récursivité terminale

Une définition de fonction f est récursive terminale quand tout appel récursif est de la forme return f(...) ; La valeur retournée est directement la valeur obtenue par un appel récursif, sans qu'il n'y ait aucune opération sur cette valeur.

Récursivité terminale

Une définition de fonction f est récursive terminale quand tout appel récursif est de la forme return f(...) ; La valeur retournée est directement la valeur obtenue par un appel récursif, sans qu'il n'y ait aucune opération sur cette valeur.

```
function recursionTerminale(n)
    ...
    return recursionTerminale(n − 1)
```

Récursivité terminale

Une définition de fonction f est récursive terminale quand tout appel récursif est de la forme return f(...) ; La valeur retournée est directement la valeur obtenue par un appel récursif, sans qu'il n'y ait aucune opération sur cette valeur.

1: **function** *recursionTerminale*(n)
2:     **...**
3:     **return** *recursionTerminale*($n-1$)

1: **function** *recursionNonTerminale*(n)
2:     **...**
3:     **return** $n +$ *recursionTerminale*($n-1$)

Récursivité terminale

Une définition de fonction f est récursive terminale quand tout appel récursif est de la forme return f(...) ; La valeur retournée est directement la valeur obtenue par un appel récursif, sans qu'il n'y ait aucune opération sur cette valeur.

```
function recursionTerminale(n)
    ...
    return recursionTerminale(n − 1)
```

```
function recursionNonTerminale(n)
    ...
    return n + recursionTerminale(n − 1)
```

Dans le premier cas, aucune référence aux résultats précédents n'est à conserver en mémoire, tandis que dans le second cas, tous les résultats intermédiaires doivent l'être.

Et la fonction factorielle ?

Reprenons la fonction *factorielle*

```
function fact(n)
    if n == 0 then
        return 1
    else
        return n × fact(n − 1)
```

Et la fonction factorielle ?

Reprenons la fonction *factorielle*

```
function fact(n)
    if n == 0 then
        return 1
    else
        return n × fact(n − 1)
```

Est-elle récursive terminale ?

Et la fonction factorielle ?

Reprenons la fonction *factorielle*

```
function fact(n)
    if n == 0 then
        return 1
    else
        return n × fact(n − 1)
```

Est-elle récursive terminale ? Non, il y a une multiplication par n avant de retourner.

Exemple d'une fonction récursive terminale

```
function f(n, a)
    if n <= 1 then
        return a
    else
        return f(n − 1, n ∗ a)
```

Exemple d'une fonction récursive terminale

```
function f(n, a)
    if n <= 1 then
        return a
    else
        return f(n − 1, n ∗ a)
```

Que fait cette fonction ?

Avantage de la récursivité terminale

- La plupart des langages fonctionnels (comme *LISP* et *CAML*) exécutent un programme à récursivité terminale comme s'il était itératif, c'est-à-dire en espace constant.
- Une fonction récursive terminale peut toujours être transformée en itération.

Petit exercice

Transformer cette fonction récursive terminale en itération

```
function f(n, a)
    if n <= 1 then
        return a
    else
        return f(n − 1, n ∗ a)
```

Transformer cette fonction récursive terminale en itération

function f(n, a)
  if n <= 1 **then**
    return a
  else
    return $f(n - 1, n * a)$

function f(n)
  $a \leftarrow 1$
  while n > 1 **do**
    $a \leftarrow n * a$
    $n \leftarrow n - 1$
  return a

Récursivité multiple

Définition
Un algorithme récursif est multiple si l'un des cas qu'il distingue se résout avec plusieurs appels récursifs.

Prenons la suite de fibonacci comme exemple

$$fibo(n) = \begin{cases} 1 & \text{, si } n = 0. \\ 1 & \text{, si } n = 1. \\ fibo(n-1) + fibo(n-2) & \text{, si } n > 1. \end{cases} \tag{1}$$

Définition

Deux algorithmes sont dits mutuellement récursifs si l'un fait appel à l'autre et l'autre à l'un. On parle aussi de récursivité croisée.

```
function pair(n)
    if n == 0 then
        return True
    else
        return impair(n − 1)
function impair(n)
    if n == 0 then
        return False
    else
        return pair(n − 1)
```

Recherche dans un tableau

Algorithme de recherche d'un élément dans un tableau

```
function search(tab, e)
    for i < tab.size do
        if tab[i] == e then
            return i
    return nonTrouvé
```

Algorithme de recherche d'un élément dans un tableau

```
function search(tab, e)
    for i < tab.size do
        if tab[i] == e then
            return i
    return nonTrouvé
```

Quelle est la complexité de cette fonction ?

Algorithme de recherche d'un élément dans un tableau

```
function search(tab, e)
    for i < tab.size do
        if tab[i] == e then
            return i
    return nonTrouvé
```

Quelle est la complexité de cette fonction ? $\mathcal{O}(n)$

Recherche d’un élément dans un tableau

La complexité précédente est trop élevée, surtout sachant que la recherche dans un tableau est une opération de base utilisée dans de nombreux algorithmes.
Pour aller plus vite, on peut utiliser les **tableaux triés** et la **dichotomie**.

Exemple : La dichotomie

Definition

La recherche dichotomique, ou recherche par dichotomie (en anglais : *binary search*), est un algorithme de recherche pour trouver la position d'un élément dans un tableau trié.

Le principe

L'objectif est trouver la position d'un élément dans un tableau trié.

- On trouve l'élément m avec la position la plus centrale du tableau (si le tableau est vide on s'arrête) ;
- On compare la valeur de l'élément recherché avec l'élément m ;
- Si elle est plus petite, on recommence dans le sous-tableau de gauche, sinon dans le sous-tableau de droite.

L'algorithme de la dichotomie

```
function dichotomie(tab, min, max, e)
    if min == max then
        if tab[min] = e then
            return min
        else
            return nonTrouvé
    mid ← (min + max)/2
    if tab[mid] < e then
        return dichotomie(tab, mid + 1, max, e)
    else
        return dichotomie(tab, min, mid, e)
```

L'algorithme de la dichotomie

```
function dichotomie(tab, min, max, e)
    if min == max then
        if tab[min] = e then
            return min
        else
            return nonTrouvé
    mid ← (min + max)/2
    if tab[mid] < e then
        return dichotomie(tab, mid + 1, max, e)
    else
        return dichotomie(tab, min, mid, e)
```

Quelle est la complexité de cette fonction ?

```
function dichotomie(tab, min, max, e)
    if min == max then
        if tab[min] = e then
            return min
        else
            return nonTrouvé
    mid ← (min + max)/2
    if tab[mid] < e then
        return dichotomie(tab, mid + 1, max, e)
    else
        return dichotomie(tab, min, mid, e)
```

Quelle est la complexité de cette fonction ? $\mathcal{O}(log_2(n))$

L'algorithme de la dichotomie itérative

```
function dichotomie(tab, e)
    min ← 0
    max ← taille - 1
    while min < max do
        mid ← (min + max)/2
        if tab[mid] < e then
            min ← mid + 1
        else
            max ← mid
    if tab[min] == e then
        return min
    else
        return nonTrouvé
```

L'algorithme de la dichotomie itérative

```
function dichotomie(tab, e)
    min ← 0
    max ← taille - 1
    while min < max do
        mid ← (min + max)/2
        if tab[mid] < e then
            min ← mid + 1
        else
            max ← mid
    if tab[min] == e then
        return min
    else
        return nonTrouvé
```

Quelle est la complexité de cette fonction ?

L'algorithme de la dichotomie itérative

```
function dichotomie(tab, e)
    min ← 0
    max ← taille - 1
    while min < max do
        mid ← (min + max)/2
        if tab[mid] < e then
            min ← mid + 1
        else
            max ← mid
    if tab[min] == e then
        return min
    else
        return nonTrouvé
```

Quelle est la complexité de cette fonction ? $\mathcal{O}(log_2(n))$

Exercice
Déterminer la version itérative de l'algorithme de la dichotomie

L'algorithme de la dichotomie itérative (variante)

```
function dichotomie(tab, e)
    min ← 0
    max ← taille - 1
    while min < max do
        mid ← (min + max)/2
        if tab[mid] == e then
            return mid
        if tab[mid] < e then
            min ← mid + 1
        else
            max ← mid
    if tab[min] == e then
        return min
    else
        return nonTrouvé
```

L'algorithme de la dichotomie itérative (variante)

```
function dichotomie(tab, e)
    min ← 0
    max ← taille - 1
    while min < max do
        mid ← (min + max)/2
        if tab[mid] == e then
            return mid
        if tab[mid] < e then
            min ← mid + 1
        else
            max ← mid
    if tab[min] == e then
        return min
    else
        return nonTrouvé
```


Quelle est la complexité de cette fonction ?

L'algorithme de la dichotomie itérative (variante)

```
function dichotomie(tab, e)
    min ← 0
    max ← taille - 1
    while min < max do
        mid ← (min + max)/2
        if tab[mid] == e then
            return mid
        if tab[mid] < e then
            min ← mid + 1
        else
            max ← mid
    if tab[min] == e then
        return min
    else
        return nonTrouvé
```

Quelle est la complexité de cette fonction ? $\mathcal{O}(log_2(n))$

Autres applications de la recherche dichotomique

- Jeu du nombre inconnu où l'on répond soit "plus grand" soit "plus petit" soit "gagné" ;
- Calcul d'une racine d'une fonction croissante ;
- Algorithme de pointage et de visée ;
- Recherche de l'apparition d'un bug dans l'histoire d'un programme.

Tri de tableaux et algorithmes de tris

Insertion dans un tableau trié

function Insertion(tab, e)
 $i \leftarrow taille$
 while i $>$ 0 and tab[i - 1] $>$ e **do**
  $tab[i] \leftarrow tab[i-1]$
  $i \leftarrow i-1$
 $tab[i] \leftarrow e$
 $taille \leftarrow taille + 1$

Insertion dans un tableau trié

```
function Insertion(tab, e)
    i ← taille
    while i > 0 and tab[i - 1] > e do
        tab[i] ← tab[i − 1]
        i ← i − 1
    tab[i] ← e
    taille ← taille + 1
```

Quelle est la complexité de cette fonction ?

Insertion dans un tableau trié

function Insertion(tab, e)
    $i \leftarrow taille$
    while i > 0 and tab[i - 1] > e **do**
        $tab[i] \leftarrow tab[i-1]$
        $i \leftarrow i - 1$
    $tab[i] \leftarrow e$
    $taille \leftarrow taille + 1$

Quelle est la complexité de cette fonction ? $\mathcal{O}(n)$

Tri par insertion

```
function Insertsort(tab)
    i ← 1
    for i < taille do
        e ← t[i]
        j ← i
        while j > 0 and tab[j - 1] > e do
            tab[j] ← tab[j – 1]
            j ← j – 1
        tab[j] ← e
```

Tri par insertion

```
function Insertsort(tab)
    i ← 1
    for i < taille do
        e ← t[i]
        j ← i
        while j > 0 and tab[j - 1] > e do
            tab[j] ← tab[j − 1]
            j ← j − 1
        tab[j] ← e
```

Quelle est la complexité de cette fonction ?

Tri par insertion

```
function Insertsort(tab)
    i ← 1
    for i < taille do
        e ← t[i]
        j ← i
        while j > 0 and tab[j - 1] > e do
            tab[j] ← tab[j − 1]
            j ← j − 1
        tab[j] ← e
```

Quelle est la complexité de cette fonction ? $\mathcal{O}((n^2))$

Diviser pour régner

Approche diviser pour régner

En informatique, **diviser pour régner** est une technique algorithmique consistant à :

1. Diviser : découper un problème initial en sous-problèmes ;
2. Régner : résoudre les sous-problèmes (récursivement ou directement s'ils sont assez petits) ;
3. Combiner : calculer une solution au problème initial à partir des solution des sous-problèmes.

Approche diviser pour régner

En informatique, **diviser pour régner** est une technique algorithmique consistant à :

1. Diviser : découper un problème initial en sous-problèmes ;
2. Régner : résoudre les sous-problèmes (récursivement ou directement s'ils sont assez petits) ;
3. Combiner : calculer une solution au problème initial à partir des solution des sous-problèmes.

Comment obtenir un tableau trié, si l'on sait trier chaque moitié ?

function *fusion*$(A[a_1, a_2, \ldots, a_n], B[b_1, b_2, \ldots, b_n])$
  if A est vide **then**
    return B
  if B est vide **then**
    return A
  if $A[a_1] <= B[b_1]$ **then**
    return A$[a_1]$ + fusion$(A[a_2, \ldots, a_n], B)$
  else
    return B$[b_1]$ + fusion$(A, B[b_2, \ldots, b_n])$

Fusion de tableaux triés

function *fusion*$(A[a_1, a_2, \ldots, a_n], B[b_1, b_2, \ldots, b_n])$
  if A est vide **then**
    return B
  if B est vide **then**
    return A
  if $A[a_1] <= B[b_1]$ **then**
    return A$[a_1]$ + fusion$(A[a_2, \ldots, a_n], B)$
  else
    return B$[b_1]$ + fusion$(A, B[b_2, \ldots, b_n])$

Quelle est la complexité de cette fonction ?

Fusion de tableaux triés

function *fusion*$(A[a_1, a_2, \ldots, a_n], B[b_1, b_2, \ldots, b_n])$
    if A est vide **then**
        return B
    if B est vide **then**
        return A
    if $A[a_1] <= B[b_1]$ **then**
        return A$[a_1]$ + fusion$(A[a_2, \ldots, a_n], B)$
    else
        return B$[b_1]$ + fusion$(A, B[b_2, \ldots, b_n])$

Quelle est la complexité de cette fonction ? $\mathcal{O}((t_a + t_b))$

Tri par fusion (MergeSort)

function *trifusion*($T[t_1, t_2, \ldots, t_n]$)
    if $t_n <= 1$ **then**
        return T
    else
        return fusion(triFusion(T[$t_1, \ldots, t_{n/2}$]), triFusion(T[$t_{n/2+1}, \ldots, t_n$]))

Tri par fusion (MergeSort)

```
function trifusion(T[t_1, t_2, ..., t_n])
    if t_n <= 1 then
        return T
    else
        return fusion(triFusion(T[t_1, ..., t_{n/2}]), triFusion(T[t_{n/2+1}, ..., t_n]))
```

Quelle est la complexité de cette fonction ?

Tri par fusion (MergeSort)

```
function trifusion(T[t_1, t_2, ..., t_n])
    if t_n <= 1 then
        return T
    else
        return fusion(triFusion(T[t_1, ..., t_{n/2}]), triFusion(T[t_{n/2+1}, ..., t_n]))
```

Quelle est la complexité de cette fonction ? $\mathcal{O}((t_a + t_b))$

Complexité d'un algorithme diviser pour régner

La complexité

Le temps d'exécution d'un algorithme *diviser pour régner* se décompose suivant les trois étapes du paradigme de base.

- Diviser : le problème en **a** sous-problèmes chacun de taille **n/b**. Soit **D(n)** le temps nécessaire à la division du problème en sous-problèmes ;
- Régner : soit **a T(n/b)** le temps de résolution des **a** sous-problèmes ;
- Combiner : soit **C(n)** le temps nécessaire pour construire la solution finale à partir des solutions aux sous-problèmes.

Finalement, le temps d'exécution global de l'algorithme est :

$$T(n) = aT(n/b) + D(n) + C(n)$$

Soit la fonction $f(n)$ qui regroupe D(n) et C(n). T(n) est alors définie de la façon suivante :

$$T(n) = aT(n/b) + f(n)$$

La complexité

Le temps d'exécution d'un algorithme *diviser pour régner* se décompose suivant les trois étapes du paradigme de base.

- Diviser : le problème en **a** sous-problèmes chacun de taille **n/b**. Soit **D(n)** le temps nécessaire à la division du problème en sous-problèmes ;
- Régner : soit **a T(n/b)** le temps de résolution des **a** sous-problèmes ;
- Combiner : soit **C(n)** le temps nécessaire pour construire la solution finale à partir des solutions aux sous-problèmes.

Finalement, le temps d'exécution global de l'algorithme est :

$$T(n) = aT(n/b) + D(n) + C(n)$$

Soit la fonction $f(n)$ qui regroupe D(n) et C(n). T(n) est alors définie de la façon suivante :

$$T(n) = aT(n/b) + f(n)$$

On s'appelle ce théorème le *master theorem*

Théorème de résolution de la récurrence

Master theorem

$$T(n) = aT(n/b) + f(n)$$

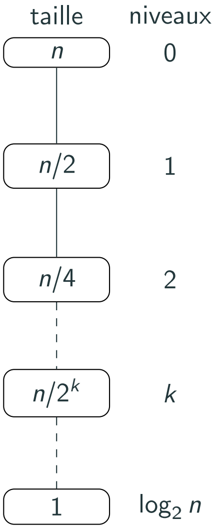
Supposons que $f(n) = c * n^k$, on a : $T(n) = aT(n/b) + cn^k$

$$a > b^k \rightarrow T(n) = \mathcal{O}(n^{log_b a})$$

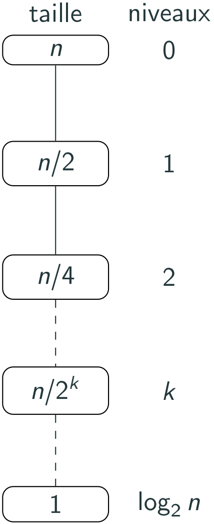
$$a = b^k \rightarrow T(n) = \mathcal{O}(n^k log_b n)$$

$$a < b^k \rightarrow T(n) = \mathcal{O}(f(n)) = O(n^k)$$

Recherche Dichotomique : Arbre récursif

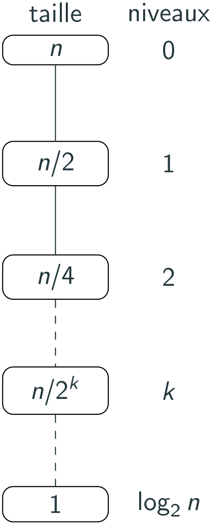


Recherche Dichotomique : Arbre récursif



Niveau k

- 1 sous problème de taille $n/2^k$
- $\mathcal{O}(1)$ opérations élémentaires par niveau



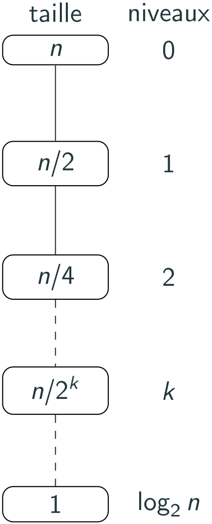
Niveau k

- 1 sous problème de taille $n/2^k$
- $\mathcal{O}(1)$ opérations élémentaires par niveau

Total :

- taille du problème divisée par 2 à chaque niveau $\implies \log_2 n + 1$ niveaux.

Recherche Dichotomique : Arbre récursif



Niveau k

- 1 sous problème de taille $n/2^k$
- $\mathcal{O}(1)$ opérations élémentaires par niveau

Total :

- taille du problème divisée par 2 à chaque niveau $\implies$ $\log_2 n + 1$ niveaux.
- <u>Complexité</u> $= \mathcal{O}(\log n)$

En utilisant le *master theorem*

- Diviser : le problème en **a** sous-problèmes chacun de taille **n/b**. Soit **D(n)** le temps nécessaire à la division du problème en sous-problèmes ;
- Régner : soit **a T(n/b)** le temps de résolution des **a** sous-problèmes ;
- Combiner : soit **C(n)** le temps nécessaire pour construire la solution finale à partir des solutions aux sous-problèmes.

```
function dichotomie(tab, min, max, e)
    if min == max then
        if tab[min] = e then
            return min
    else
        return nonTrouvé
    mid ← (min + max)/2
    if tab[mid] < e then
        return dichotomie(tab, mid + 1, max, e)
    else
        return dichotomie(tab, min, mid, e)
```

Complexité de recherche dichotomique

Rappel

Supposons que $f(n) = c * n^k$, on a : $T(n) = aT(n/b) + cn^k$

$$a > b^k \rightarrow T(n) = \mathcal{O}(n^{log_b a})$$

$$a = b^k \rightarrow T(n) = \mathcal{O}(n^k log_b n)$$

$$a < b^k \rightarrow T(n) = \mathcal{O}(f(n)) = O(n^k)$$

On a a = 1, b = 2, k = 0 → a = b^k

D'après le théorème, on a donc :

$$T(n) = \mathcal{O}(n^k log_b n) = \mathcal{O}(log(n))$$