

Algorithmique et structure de données

Réversivité 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

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

- 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 $n \geq 0$

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

Lien avec les mathématiques: la factorielle

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

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

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 $n \geq 0$

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

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 $(n - 1)!$

```
1: function fact(n)  
2:   if n == 0 then  
3:     return 1  
4:   else  
5:     return n × fact(n - 1)
```

Exécution de l'algorithme de la factorielle

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

$$\textit{fact}(4) \rightarrow 4 \times \textit{fact}(3)$$

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!:

$$\text{fact}(4) \rightarrow 4 \times \text{fact}(3)$$

$$\text{fact}(4) \rightarrow 4 \times 3 \times \text{fact}(2)$$

$$\text{fact}(4) \rightarrow 4 \times 3 \times 2 \times \text{fact}(1)$$

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

$$\text{fact}(4) \rightarrow 4 \times 3 \times 2 \times 1 \times 1$$

Exécution de l'algorithme de la factorielle

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

$$\text{fact}(4) \rightarrow 4 \times \text{fact}(3)$$

$$\text{fact}(4) \rightarrow 4 \times 3 \times \text{fact}(2)$$

$$\text{fact}(4) \rightarrow 4 \times 3 \times 2 \times \text{fact}(1)$$

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

$$\text{fact}(4) \rightarrow 4 \times 3 \times 2 \times 1 \times 1$$

$$\text{fact}(4) \rightarrow 24$$

Autre algorithme de factorielle

Considérons l'algorithme suivant :

```
1: function fact2(n)  
2:   return  $n \times \textit{fact}(n - 1)$ 
```

Autre algorithme de factorielle

Considérons l'algorithme suivant :

```
1: function fact2(n)  
2:   return  $n \times \textit{fact}(n - 1)$ 
```

Quel est le soucis avec cette fonction ?

Autre algorithme de factorielle

Considérons l'algorithme suivant :

```
1: function fact2(n)  
2:   return  $n \times \textit{fact}(n - 1)$ 
```

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!:

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

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

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

$$fact2(1) \rightarrow \dots$$

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

```
1: function fact3(n)  
2:   if n == 0 then  
3:     return 1  
4:   else  
5:     return n × fact3(n + 1)
```

Reprenons notre exemple de la factorielle

```
1: function fact3(n)  
2:   if n == 0 then  
3:     return 1  
4:   else  
5:     return n × fact3(n + 1)
```

Quel est le soucis avec cette fonction ?

Reprenons notre exemple de la factorielle

```
1: function fact3(n)  
2:   if n == 0 then  
3:     return 1  
4:   else  
5:     return n × fact3(n + 1)
```

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!:

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

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

Exécution de l'algorithme de la factorielle

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

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

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

$$fact3 \rightarrow 1 \times 2 \times 3 \times \dots$$

Deuxième règle implicite

```
1: function fact3(n)  
2:   if n == 0 then  
3:     return 1  
4:   else  
5:     return n × fact3(n + 1)
```


Deuxième règle implicite

```
1: function fact3(n)  
2:   if n == 0 then  
3:     return 1  
4:   else  
5:     return n × fact3(n + 1)
```

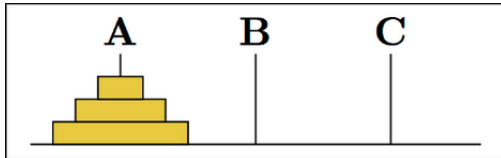
Deuxième règle

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

Un autre exemple: les tours de Hanoï

Principe du jeu

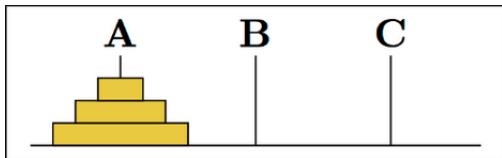
- On dispose de 3 tours côte à côte sur lesquelles on peut empiler des disques de diamètre croissant;
- On ne peut déplacer qu'un disque à la fois et on ne doit pas poser un disque plus large sur un plus petit;
- L'objectif est de trouver la suite de déplacements qui permet de placer tous les disques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

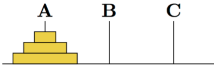
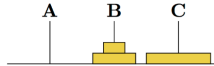
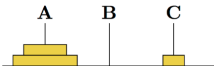
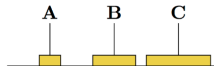
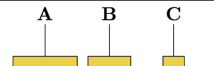
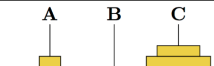

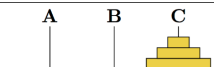


A vous de jouer ! En combien de mouvements est-ce possible ?

La réponse: 7 !





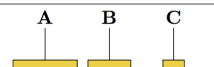
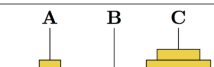

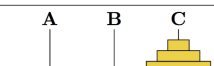
Résolution du problème de Hanoï

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 |  |

Résolution du problème de Hanoï

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 *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 .

```
1: function hanoi( $n, d, a$ )  
2:   if  $n > 0$  then  
3:      $aux \leftarrow$  le piquet différent de  $d$  et  $a$   
4:     hanoi( $n - 1, d, aux$ )  
5:     deplacerdisque( $d, a$ )  
6:     hanoi( $n - 1, aux, a$ )
```


Exercice

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

Exercice

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

```
1: function sum - inv(n)  
2:   if n == 0 then  
3:     return 1  
4:   else  
5:     return  $1/n * *2 + \textit{sum - inv}(n - 1)$ 
```

Type de récursivité

Réversivité simple ou linéaire

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

```
1: function mystere( $n$ )  
2:   if  $n == 0$  then  
3:     return 1  
4:   else  
5:     return  $\text{mystere}(n - 1) \times 2$ 
```

Réversivité simple ou linéaire

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

```
1: function mystere(n)  
2:   if n == 0 then  
3:     return 1  
4:   else  
5:     return mystere(n - 1) × 2
```

Que fait cet algorithme ?

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

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

```
1: function recursionTerminale( $n$ )  
2:   ...  
3:   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  $n + \text{recursionTerminale}(n - 1)$ 
```



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

```
1: function recursionNonTerminale(n)  
2:   ...  
3:   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*

```
1: function fact(n)  
2:   if n == 0 then  
3:     return 1  
4:   else  
5:     return n × fact(n - 1)
```

Et la fonction factorielle ?

Reprenons la fonction *factorielle*

```
1: function fact(n)  
2:   if n == 0 then  
3:     return 1  
4:   else  
5:     return n × fact(n - 1)
```

Est-elle récursive terminale ?

Et la fonction factorielle ?

Reprenons la fonction *factorielle*

```
1: function fact(n)  
2:   if n == 0 then  
3:     return 1  
4:   else  
5:     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

```
1: function f( $n, a$ )  
2:   if  $n \leq 1$  then  
3:     return  $a$   
4:   else  
5:     return  $f(n - 1, n * a)$ 
```

Exemple d'une fonction récursive terminale

```
1: function f( $n, a$ )  
2:   if  $n \leq 1$  then  
3:     return  $a$   
4:   else  
5:     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.

Transformer cette fonction récursive terminale en itération

```
1: function  $f(n, a)$   
2:   if  $n \leq 1$  then  
3:     return  $a$   
4:   else  
5:     return  $f(n - 1, n * a)$ 
```


Transformer cette fonction récursive terminale en itération

```
1: function f( $n, a$ )  
2:   if  $n \leq 1$  then  
3:     return  $a$   
4:   else  
5:     return  $f(n - 1, n * a)$ 
```

```
1: function f( $n$ )  
2:    $a \leftarrow 1$   
3:   while  $n > 1$  do  
4:      $a \leftarrow n * a$   
5:      $n \leftarrow n - 1$   
6:   return  $a$ 
```

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

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

Définition

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

```
1: function pair( $n$ )  
2:   if  $n == 0$  then  
3:     return True  
4:   else  
5:     return impair( $n - 1$ )
```

```
1: function impair( $n$ )  
2:   if  $n == 0$  then  
3:     return False  
4:   else  
5:     return pair( $n - 1$ )
```

Recherche dans un tableau

Algorithme de recherche d'un élément dans un tableau

```
1: function search(tab, e)  
2:   for i < tab.size do  
3:     if tab[i] == e then  
4:       return i  
5:   return nonTrouvé
```

Algorithme de recherche d'un élément dans un tableau

```
1: function search(tab, e)  
2:   for i < tab.size do  
3:     if tab[i] == e then  
4:       return i  
5:   return nonTrouvé
```

Quelle est la complexité de cette fonction ?

Algorithme de recherche d'un élément dans un tableau

```
1: function search(tab, e)  
2:   for i < tab.size do  
3:     if tab[i] == e then  
4:       return i  
5:   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

```
1: function dichotomie(tab, min, max, e)
2:   if min == max then
3:     if tab[min] = e then
4:       return min
5:   else
6:     return nonTrouvé
7:   mid ← (min + max)/2
8:   if tab[mid] < e then
9:     return dichotomie(tab, mid + 1, max, e)
10:  else
11:    return dichotomie(tab, min, mid, e)
```

L'algorithme de la dichotomie

```
1: function dichotomie(tab, min, max, e)
2:   if min == max then
3:     if tab[min] = e then
4:       return min
5:   else
6:     return nonTrouvé
7:   mid ← (min + max)/2
8:   if tab[mid] < e then
9:     return dichotomie(tab, mid + 1, max, e)
10:  else
11:    return dichotomie(tab, min, mid, e)
```

Quelle est la complexité de cette fonction ?

L'algorithme de la dichotomie

```
1: function dichotomie(tab, min, max, e)
2:   if min == max then
3:     if tab[min] = e then
4:       return min
5:   else
6:     return nonTrouvé
7:   mid ← (min + max)/2
8:   if tab[mid] < e then
9:     return dichotomie(tab, mid + 1, max, e)
10:  else
11:    return dichotomie(tab, min, mid, e)
```

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

L'algorithme de la dichotomie itérative

```
1: function dichotomie(tab, e)
2:   min  $\leftarrow$  0
3:   max  $\leftarrow$  taille - 1
4:   while min < max do
5:     mid  $\leftarrow$  (min + max)/2
6:     if tab[mid] < e then
7:       min  $\leftarrow$  mid + 1
8:     else
9:       max  $\leftarrow$  mid
10:    if tab[min] == e then
11:      return min
12:    else
13:      return nonTrouvé
```

L'algorithme de la dichotomie itérative

```
1: function dichotomie(tab, e)
2:   min  $\leftarrow$  0
3:   max  $\leftarrow$  taille - 1
4:   while min < max do
5:     mid  $\leftarrow$  (min + max)/2
6:     if tab[mid] < e then
7:       min  $\leftarrow$  mid + 1
8:     else
9:       max  $\leftarrow$  mid
10:  if tab[min] == e then
11:    return min
12:  else
13:    return nonTrouvé
```

Quelle est la complexité de cette fonction ?

L'algorithme de la dichotomie itérative

```
1: function dichotomie(tab, e)
2:   min  $\leftarrow$  0
3:   max  $\leftarrow$  taille - 1
4:   while min < max do
5:     mid  $\leftarrow$  (min + max)/2
6:     if tab[mid] < e then
7:       min  $\leftarrow$  mid + 1
8:     else
9:       max  $\leftarrow$  mid
10:  if tab[min] == e then
11:    return min
12:  else
13:    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)

```
1: function dichotomie(tab, e)
2:   min  $\leftarrow$  0
3:   max  $\leftarrow$  taille - 1
4:   while min < max do
5:     mid  $\leftarrow$  (min + max)/2
6:     if tab[mid] == e then
7:       return mid
8:     if tab[mid] < e then
9:       min  $\leftarrow$  mid + 1
10:    else
11:      max  $\leftarrow$  mid
12:    if tab[min] == e then
13:      return min
14:    else
15:      return nonTrouvé
```

L'algorithme de la dichotomie itérative (variante)

```
1: function dichotomie(tab, e)
2:   min  $\leftarrow$  0
3:   max  $\leftarrow$  taille - 1
4:   while min < max do
5:     mid  $\leftarrow$  (min + max)/2
6:     if tab[mid] == e then
7:       return mid
8:     if tab[mid] < e then
9:       min  $\leftarrow$  mid + 1
10:    else
11:      max  $\leftarrow$  mid
12:    if tab[min] == e then
13:      return min
14:    else
15:      return nonTrouvé
```

L'algorithme de la dichotomie itérative (variante)

```
1: function dichotomie(tab, e)
2:   min  $\leftarrow$  0
3:   max  $\leftarrow$  taille - 1
4:   while min < max do
5:     mid  $\leftarrow$  (min + max)/2
6:     if tab[mid] == e then
7:       return mid
8:     if tab[mid] < e then
9:       min  $\leftarrow$  mid + 1
10:    else
11:      max  $\leftarrow$  mid
12:    if tab[min] == e then
13:      return min
14:    else
15:      return nonTrouvé
```

- 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

```
1: function Insertion(tab, e)  
2:   i  $\leftarrow$  taille  
3:   while i > 0 and tab[i - 1] > e do  
4:     tab[i]  $\leftarrow$  tab[i - 1]  
5:     i  $\leftarrow$  i - 1  
6:   tab[i]  $\leftarrow$  e  
7:   taille  $\leftarrow$  taille + 1
```

Insertion dans un tableau trié

```
1: function Insertion(tab, e)  
2:   i ← taille  
3:   while i > 0 and tab[i - 1] > e do  
4:     tab[i] ← tab[i - 1]  
5:     i ← i - 1  
6:   tab[i] ← e  
7:   taille ← taille + 1
```

Quelle est la complexité de cette fonction ?

Insertion dans un tableau trié

```
1: function Insertion(tab, e)  
2:   i ← taille  
3:   while i > 0 and tab[i - 1] > e do  
4:     tab[i] ← tab[i - 1]  
5:     i ← i - 1  
6:   tab[i] ← e  
7:   taille ← taille + 1
```

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


```
1: function Insertsort(tab,)
2:    $i \leftarrow 1$ 
3:   for  $i < \text{taille}$  do
4:      $e \leftarrow t[i]$ 
5:      $j \leftarrow i$ 
6:     while  $j > 0$  and  $\text{tab}[j - 1] > e$  do
7:        $\text{tab}[j] \leftarrow \text{tab}[j - 1]$ 
8:        $j \leftarrow j - 1$ 
9:      $\text{tab}[j] \leftarrow e$ 
```

Tri par insertion

```
1: function Insertsort(tab,)
2:    $i \leftarrow 1$ 
3:   for  $i < \text{taille}$  do
4:      $e \leftarrow t[i]$ 
5:      $j \leftarrow i$ 
6:     while  $j > 0$  and  $\text{tab}[j - 1] > e$  do
7:        $\text{tab}[j] \leftarrow \text{tab}[j - 1]$ 
8:        $j \leftarrow j - 1$ 
9:      $\text{tab}[j] \leftarrow e$ 
```

Quelle est la complexité de cette fonction ?

Tri par insertion

```
1: function Insertsort(tab,)
2:   i ← 1
3:   for i < taille do
4:     e ← t[i]
5:     j ← i
6:     while j > 0 and tab[j - 1] > e do
7:       tab[j] ← tab[j - 1]
8:       j ← j - 1
9:     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.

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 tirer chaque moitié ?

```
1: function fusion( $A[a_1, a_2, \dots, a_n]$ ,  $B[b_1, b_2, \dots, b_n]$ )  
2:   if A est vide then  
3:     return B  
4:   if B est vide then  
5:     return A  
6:   if  $A[a_1] \leq B[b_1]$  then  
7:     return  $A[a_1] + \text{fusion}(A[a_2, \dots, a_n], B)$   
8:   else  
9:     return  $B[b_1] + \text{fusion}(A, B[b_2, \dots, b_n])$ 
```

```
1: function fusion( $A[a_1, a_2, \dots, a_n], B[b_1, b_2, \dots, b_n]$ )  
2:   if A est vide then  
3:     return B  
4:   if B est vide then  
5:     return A  
6:   if  $A[a_1] \leq B[b_1]$  then  
7:     return  $A[a_1] + \text{fusion}(A[a_2, \dots, a_n], B)$   
8:   else  
9:     return  $B[b_1] + \text{fusion}(A, B[b_2, \dots, b_n])$ 
```

Quelle est la complexité de cette fonction ?


```
1: function fusion( $A[a_1, a_2, \dots, a_n], B[b_1, b_2, \dots, b_n]$ )  
2:   if A est vide then  
3:     return B  
4:   if B est vide then  
5:     return A  
6:   if  $A[a_1] \leq B[b_1]$  then  
7:     return  $A[a_1] + \text{fusion}(A[a_2, \dots, a_n], B)$   
8:   else  
9:     return  $B[b_1] + \text{fusion}(A, B[b_2, \dots, b_n])$ 
```

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

Tri par fusion (MergeSort)

```
1: function trifusion( $T[t_1, t_2, \dots, t_n]$ )  
2:   if  $t_n \leq 1$  then  
3:     return  $T$   
4:   else  
5:     return fusion(triFusion( $T[t_1, \dots, t_{n/2}]$ ),  
    triFusion( $T[t_{n/2+1}, \dots, t_n]$ ))
```

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

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

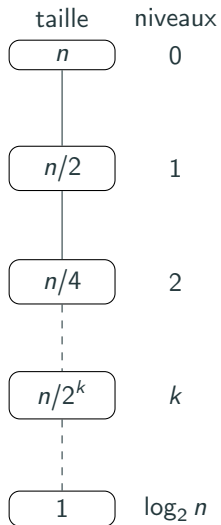
Supposons que $f(n) = c * n^k$, on a : $T(n) = aT(n/b) + cn^k$

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

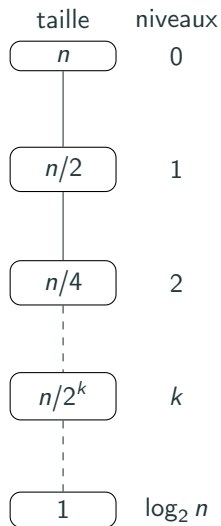
$$a = b^k \rightarrow T(n) = \mathcal{O}(n^k \log_b n)$$

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

Recherche Dichotomique : Arbre récursif



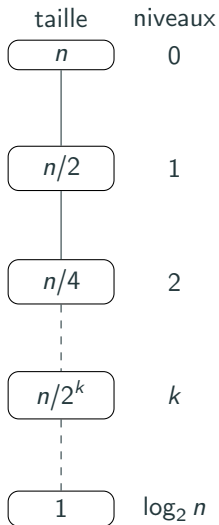
Recherche Dichotomique : Arbre récursif



Niveau k

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

Recherche Dichotomique : Arbre récursif



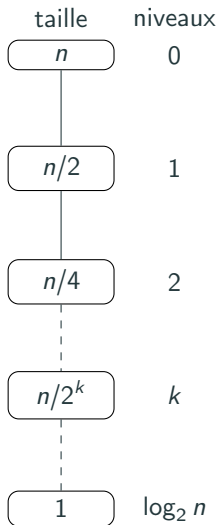
Niveau k

- 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.

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.
- Complexité = $\mathcal{O}(\log n)$

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.

```
1: function                                di-  
   dichotomie(tab, min, max, e)  
2:   if min == max then  
3:     if tab[min] = e then  
4:       return min  
5:   else  
6:     return nonTrouvé  
7:   mid ← (min + max)/2  
8:   if tab[mid] < e then  
9:     return dichotomie(tab, mid +  
   1, max, e)  
10:  else  
11:    return  
    dichotomie(tab, min, mid, e)
```

Complexité de recherche dichotomique

Rappel

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

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

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

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

On a $a = 1$, $b = 2$, $k = 0 \rightarrow a = b^k$

D'après le théorème, on a donc:

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