

# Complexité

Quentin Fortier

September 21, 2021

## Définition : Algorithme

Un algorithme est composé de :

- Une (des) entrée(s)
- Une (des) sortie(s)
- Des instructions pour passer de l'entrée à la sortie

## Complexité (en temps)

La complexité d'un algorithme est le nombre d'opérations élémentaires (+, -, \*, ...) qu'il effectue, en fonction de la taille de l'entrée.

```
let rec mem e l = match l with  
  | [] -> false  
  | x::q -> x = e || mem e q
```

```
let rec mem e l = match l with  
  | [] -> false  
  | x::q -> x = e || mem e q
```

Les opérations élémentaires effectuées par mem sont match,  $x = e$ ,  $||$ .

```
let rec mem e l = match l with  
  | [] -> false  
  | x::q -> x = e || mem e q
```

Les opérations élémentaires effectuées par `mem` sont `match`, `x = e`, `||`.  
La complexité de `mem e l` pour une liste de taille  $n$  est donc  $3n$ .

# Complexité

En pratique, on veut juste avoir un ordre de grandeur du nombre d'opérations en fonction de  $n$ .

En pratique, on veut juste avoir un ordre de grandeur du nombre d'opérations en fonction de  $n$ .



En pratique, on veut juste avoir un ordre de grandeur du nombre d'opérations en fonction de  $n$ .

## Notation $O$ (grand $O$ )

Soit  $f$  et  $g$  deux fonctions. On dit que  $f(n) = O(g(n))$  si :

$$\exists A \geq 0, \exists N, \forall n \geq N, f(n) \leq Ag(n)$$

En pratique, on veut juste avoir un ordre de grandeur du nombre d'opérations en fonction de  $n$ .

## Notation $O$ (grand $O$ )

Soit  $f$  et  $g$  deux fonctions. On dit que  $f(n) = O(g(n))$  si :

$$\exists A \geq 0, \exists N, \forall n \geq N, f(n) \leq Ag(n)$$

«  $f(n)$  est inférieur à  $g(n)$ , à une constante près et pour  $n$  assez grand »

## Notation O (grand O)

Soit  $f$  et  $g$  deux fonctions. On dit que  $f(n) = O(g(n))$  si :

$$\exists A \geq 0, \exists N, \forall n \geq N, f(n) \leq Ag(n)$$

Exemples :

- $3n =$

## Notation O (grand O)

Soit  $f$  et  $g$  deux fonctions. On dit que  $f(n) = O(g(n))$  si :

$$\exists A \geq 0, \exists N, \forall n \geq N, f(n) \leq Ag(n)$$

Exemples :

- $3n = O(n)$
- $n \ln(n) + 2n =$

## Notation $O$ (grand $O$ )

Soit  $f$  et  $g$  deux fonctions. On dit que  $f(n) = O(g(n))$  si :

$$\exists A \geq 0, \exists N, \forall n \geq N, f(n) \leq Ag(n)$$

Exemples :

- $3n = O(n)$
- $n \ln(n) + 2n = O(n \ln n)$
- $5 \ln(n) + 2\sqrt{n} =$

## Notation $O$ (grand $O$ )

Soit  $f$  et  $g$  deux fonctions. On dit que  $f(n) = O(g(n))$  si :

$$\exists A \geq 0, \exists N, \forall n \geq N, f(n) \leq Ag(n)$$

Exemples :

- $3n = O(n)$
- $n \ln(n) + 2n = O(n \ln n)$
- $5 \ln(n) + 2\sqrt{n} = O(\sqrt{n})$

## Notation $O$ (grand $O$ )

Soit  $f$  et  $g$  deux fonctions. On dit que  $f(n) = O(g(n))$  si :

$$\exists A \geq 0, \exists N, \forall n \geq N, f(n) \leq Ag(n)$$

Exemples :

- $3n = O(n)$
- $n \ln(n) + 2n = O(n \ln n)$
- $5 \ln(n) + 2\sqrt{n} = O(\sqrt{n})$

On conserve dans le  $O(\dots)$  le terme qui augmente le plus vite quand  $n \rightarrow \infty$ , sans la constante.

Exemples de calculs sur les  $O(\dots)$  :

- $O(n) + O(n^2) =$



Exemples de calculs sur les  $O(\dots)$  :

- $O(n) + O(n^2) = O(n^2)$
- $O(n) \times O(n^2) =$

Exemples de calculs sur les  $O(\dots)$  :

- $O(n) + O(n^2) = O(n^2)$
- $O(n) \times O(n^2) = O(n^3)$

# Complexité

Complexités classiques, de la meilleure (plus petite) à la plus mauvaise :

- $O(1)$  : **complexité constante**  
→ `a.(i)`, `Array.length`, `e::l`

# Complexité

Complexités classiques, de la meilleure (plus petite) à la plus mauvaise :

- $O(1)$  : **complexité constante**  
→ `a.(i)`, `Array.length`, `e::l`
- $O(\ln(n))$  : **complexité logarithmique**  
→ dichotomie, exponentiation rapide

# Complexité

Complexités classiques, de la meilleure (plus petite) à la plus mauvaise :

- $O(1)$  : **complexité constante**  
→ `a.(i)`, `Array.length`, `e::l`
- $O(\ln(n))$  : **complexité logarithmique**  
→ dichotomie, exponentiation rapide
- $O(n)$  : **complexité linéaire**  
→ dernier élément d'une liste, `Array.make`

# Complexité

Complexités classiques, de la meilleure (plus petite) à la plus mauvaise :

- $O(1)$  : **complexité constante**  
→ `a.(i)`, `Array.length`, `e::l`
- $O(\ln(n))$  : **complexité logarithmique**  
→ dichotomie, exponentiation rapide
- $O(n)$  : **complexité linéaire**  
→ dernier élément d'une liste, `Array.make`
- $O(n \log(n))$  : **complexité presque linéaire**  
→ complexité optimale d'un tri (ex : tri fusion)

# Complexité

Complexités classiques, de la meilleure (plus petite) à la plus mauvaise :

- $O(1)$  : **complexité constante**  
→ `a.(i)`, `Array.length`, `e::1`
- $O(\ln(n))$  : **complexité logarithmique**  
→ dichotomie, exponentiation rapide
- $O(n)$  : **complexité linéaire**  
→ dernier élément d'une liste, `Array.make`
- $O(n \log(n))$  : **complexité presque linéaire**  
→ complexité optimale d'un tri (ex : tri fusion)
- $O(a^n)$  : **complexité exponentielle**  
→ force brute (tester toutes les possibilités)

# Complexité

Complexités classiques, de la meilleure (plus petite) à la plus mauvaise :

- $O(1)$  : **complexité constante**  
→ `a.(i)`, `Array.length`, `e::1`
- $O(\ln(n))$  : **complexité logarithmique**  
→ dichotomie, exponentiation rapide
- $O(n)$  : **complexité linéaire**  
→ dernier élément d'une liste, `Array.make`
- $O(n \log(n))$  : **complexité presque linéaire**  
→ complexité optimale d'un tri (ex : tri fusion)
- $O(a^n)$  : **complexité exponentielle**  
→ force brute (tester toutes les possibilités)

**Remarque** : Un algorithme en  $O(n)$  est aussi en  $O(n^2)$ ,  $O(2^n)$ ... On donnera la meilleure borne possible.



## Exercice

Quelle complexité pour calculer la somme des termes d'une matrice  $n \times p$ ?

## Exercice

Quelle complexité pour calculer la somme des termes d'une matrice  $n \times p$ ?

```
let somme m =  
  let res = ref 0 in  
  let n, p = Array.length m, Array.length m.(0) in  
  for i = 0 to n - 1 do  
    for j = 0 to p - 1 do  
      res := !res + m.(i).(j)  
    done  
  done;  
  !res
```

```
let somme m =  
  let res = ref 0 in  
  let n, p = Array.length m, Array.length m.(0) in  
  for i = 0 to n - 1 do  
    for j = 0 to p - 1 do  
      res := !res + m.(i).(j)  
    done  
  done;  
  !res
```

```
let somme m =  
  let res = ref 0 in  
  let n, p = Array.length m, Array.length m.(0) in  
  for i = 0 to n - 1 do  
    for j = 0 to p - 1 do  
      res := !res + m.(i).(j)  
    done  
  done;  
  !res
```

L'instruction `res := !res + m.(i).(j)` est répétée  $n \times p$  fois.

→ Complexité  $O(np)$

Quand on **imbrique** des boucles for (l'un dans l'autre), on **multiplie** les complexités

Quand on **enchaîne** des instructions (l'un après l'autre), on **additionne** les complexités.

Pour trouver la complexité d'une fonction récursive/boucle while, lorsque ce n'est pas évident, on cherche souvent une équation de récurrence sur le nombre d'appels rékursifs / d'itérations.

# Exponentiation rapide

## Problème

Calculer  $a^n$ .

Méthode 1 : utiliser  $a^n = \underbrace{a \times a \dots \times a}_n \rightarrow n - 1$  multiplications

# Exponentiation rapide

## Problème

Calculer  $a^n$ .

Méthode 1 : utiliser  $a^n = \underbrace{a \times a \dots \times a}_n \rightarrow n - 1$  multiplications

Méthode 2 :

$$\begin{cases} a^n = (a^{\frac{n}{2}})^2 & \text{si } n \text{ est pair} \\ a^n = a \times (a^{\frac{n-1}{2}})^2 & \text{sinon} \end{cases}$$



## Exponentiation rapide

```
let rec exp_rapide a n =  
  if n = 0 then 1  
  else let b = exp_rapide a (n/2) in  
    if n mod 2 = 0 then b*b  
    else a*b*b
```

# Exponentiation rapide

```
let rec exp_rapide a n =  
  if n = 0 then 1  
  else let b = exp_rapide a (n/2) in  
    if n mod 2 = 0 then b*b  
    else a*b*b
```

Soit  $C(n)$  le nombre d'appels récurrents de `exp_rapide` a  $n$ .

# Exponentiation rapide

```
let rec exp_rapide a n =  
  if n = 0 then 1  
  else let b = exp_rapide a (n/2) in  
    if n mod 2 = 0 then b*b  
    else a*b*b
```

Soit  $C(n)$  le nombre d'appels récurifs de `exp_rapide a n`.

$$\begin{aligned}C(n) &= 1 + C(n/2) & (*) \\&= 1 + 1 + C(n/4) \\&= \underbrace{1 + \dots + 1}_p + C(n/2^p)\end{aligned}$$

# Exponentiation rapide

```
let rec exp_rapide a n =  
  if n = 0 then 1  
  else let b = exp_rapide a (n/2) in  
    if n mod 2 = 0 then b*b  
    else a*b*b
```

Soit  $C(n)$  le nombre d'appels récurrents de `exp_rapide a n`.

$$\begin{aligned}C(n) &= 1 + C(n/2) & (*) \\&= 1 + 1 + C(n/4) \\&= \underbrace{1 + \dots + 1}_p + C(n/2^p)\end{aligned}$$

En appliquant  $p = \log_2(n)$  ( $\pm 1$ ) fois  $(*)$ , on obtient :

$$C(n) = \log_2(n) + C(1) = \boxed{O(\log_2(n))}$$

# Exponentiation rapide

```
let rec exp_rapide a n =  
  if n = 0 then 1  
  else let b = exp_rapide a (n/2) in  
    if n mod 2 = 0 then b*b  
    else a*b*b
```

`exp_rapide a n` effectue  $O(\log(n))$  appels récursifs et chaque appel récursif effectue un nombre constant d'opérations (en dehors de l'appel récursif).

# Exponentiation rapide

```
let rec exp_rapide a n =  
  if n = 0 then 1  
  else let b = exp_rapide a (n/2) in  
    if n mod 2 = 0 then b*b  
    else a*b*b
```

`exp_rapide a n` effectue  $O(\log(n))$  appels récursifs et chaque appel récursif effectue un nombre constant d'opérations (en dehors de l'appel récursif).

Donc `exp_rapide a n` est en complexité  $O(\log(n))$ .

# Recherche par dichotomie

La recherche par dichotomie permet de savoir si un élément appartient à un tableau **trié** plus rapidement que la recherche séquentielle.

```
let dichotomie t e =  
  (* détermine si e appartient au tableau trié t *)  
  let rec aux i j =  
    (* détermine si e appartient à t.(i), ..., t.(j) *)  
    if i > j then false (* aucun élément *)  
    else let m = (i + j)/2 in (* milieu *)  
         if t.(m) < e  
         then aux (m + 1) j (* regarde à droite *)  
         else aux i (m - 1) (* regarde à gauche *)  
  in aux 0 (Array.length t - 1)
```

# Recherche par dichotomie

La recherche par dichotomie permet de savoir si un élément appartient à un tableau **trié** plus rapidement que la recherche séquentielle.

```
let dichotomie t e =  
  (* détermine si e appartient au tableau trié t *)  
  let rec aux i j =  
    (* détermine si e appartient à t.(i), ..., t.(j) *)  
    if i > j then false (* aucun élément *)  
    else let m = (i + j)/2 in (* milieu *)  
         if t.(m) < e  
         then aux (m + 1) j (* regarde à droite *)  
         else aux i (m - 1) (* regarde à gauche *)  
  in aux 0 (Array.length t - 1)
```

Attention : la dichotomie est inutile pour une liste car l'accès au milieu demande une complexité linéaire.



# Recherche par dichotomie

```
let dico t e =  
  let rec aux i j =  
    if i > j then false  
    else let m = (i + j)/2 in  
      if t.(m) < e  
      then aux (m + 1) j  
      else aux i (m - 1)  
  in aux 0 (Array.length t - 1)
```

Soit  $C(k)$  le nombre d'appels récurifs de `aux i j` si  $j - i = k$ .

# Recherche par dichotomie

```
let dichotomie t e =  
  let rec aux i j =  
    if i > j then false  
    else let m = (i + j)/2 in  
      if t.(m) < e  
      then aux (m + 1) j  
      else aux i (m - 1)  
  in aux 0 (Array.length t - 1)
```

Soit  $C(k)$  le nombre d'appels récursifs de `aux i j` si  $j - i = k$ .

On divise au moins par 2 la taille de l'intervalle à chaque appel récursif :

$$C(k) \leq 1 + C\left(\frac{k}{2}\right)$$

C'est le même type d'équation que l'exponentiation rapide donc :

`dichotomie` est en complexité  $O(\log(n))$  sur un tableau trié de taille  $n$

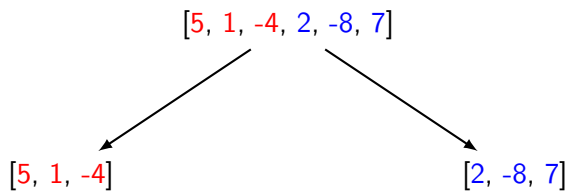
Un algorithme de tri permet de trier par ordre croissant une liste ou un tableau.

- Tri par insertion :  $O(n^2)$
- Tri par selection :  $O(n^2)$
- Tri rapide :  $O(n^2)$
- Tri fusion :  $O(n \log(n))$  (optimale)
- Tri par tas :  $O(n \log(n))$  (optimale)

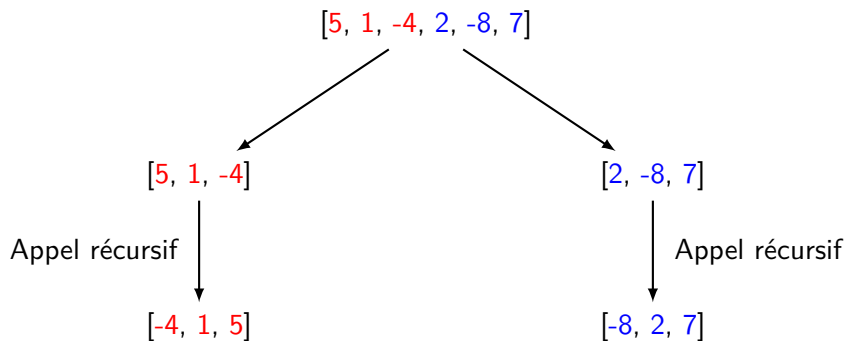
# Tri fusion

[5, 1, -4, 2, -8, 7]

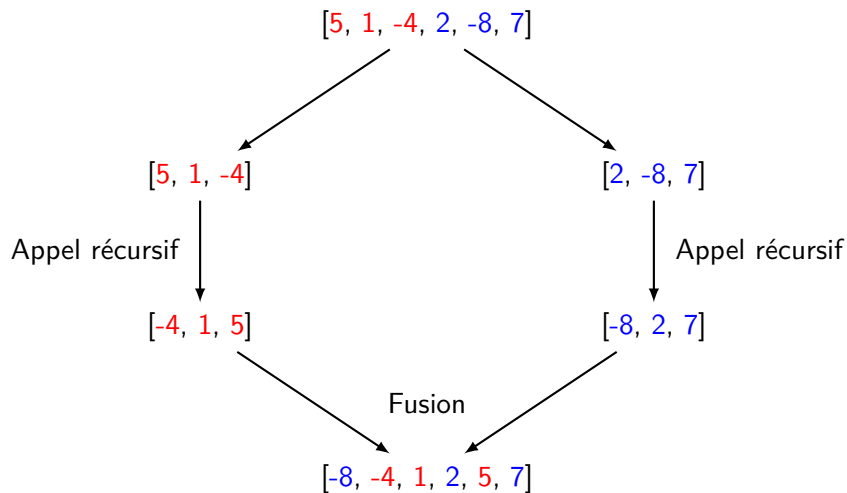
# Tri fusion



# Tri fusion



# Tri fusion



Diviser une liste en deux :

```
let rec split = function
  | [] -> [], []
  | [e] -> [e], []
  | e1::e2::q -> let q1, q2 = split q in
                  e1::q1, e2::q2
```

Complexité :



Diviser une liste en deux :

```
let rec split = function
  | [] -> [], []
  | [e] -> [e], []
  | e1::e2::q -> let q1, q2 = split q in
                  e1::q1, e2::q2
```

Complexité :  $O(n)$  où  $n$  est la taille de la liste

# Tri fusion

Fusionner deux listes triées :

```
let rec fusion l1 l2 = match l1, l2 with
| [], _ -> l2
| _, [] -> l1
| e1::q1, e2::q2 when e1 < e2 -> e1::fusion q1 l2
| e1::q1, e2::q2 -> e2::fusion l1 q2
```

Complexité :

# Tri fusion

Fusionner deux listes triées :

```
let rec fusion l1 l2 = match l1, l2 with
| [], _ -> l2
| _, [] -> l1
| e1::q1, e2::q2 when e1 < e2 -> e1::fusion q1 l2
| e1::q1, e2::q2 -> e2::fusion l1 q2
```

Complexité :  $O(n)$  où  $n$  est la taille de la plus petite liste

# Tri fusion

Tri fusion :

```
let rec tri = function
  | [] -> []
  | [e] -> [e] (* tri ne termine pas sans ce cas *)
  | l -> let l1, l2 = split l in
          fusion (tri l1) (tri l2);;
```

Complexité :

# Tri fusion

Tri fusion :

```
let rec tri = function
  | [] -> []
  | [e] -> [e] (* tri ne termine pas sans ce cas *)
  | l -> let l1, l2 = split l in
         fusion (tri l1) (tri l2);;
```

Complexité : Soit  $C(n)$  la complexité de tri  $l$  pour  $l$  de taille  $n$ .

# Tri fusion

Tri fusion :

```
let rec tri = function
  | [] -> []
  | [e] -> [e] (* tri ne termine pas sans ce cas *)
  | l -> let l1, l2 = split l in
         fusion (tri l1) (tri l2);;
```

Complexité : Soit  $C(n)$  la complexité de tri  $l$  pour  $l$  de taille  $n$ .

$$\begin{aligned} C(n) &= \underbrace{O(n)}_{\text{split}} + \underbrace{O(n)}_{\text{fusion}} + 2C(n/2) = O(n) + 2C(n/2) \\ &= O(n) + 2O(n/2) + 4C(n/4) = O(n) + O(n) + 4C(n/4) \\ &= pO(n) + 2^p C(n/2^p) \underset{p=\log_2(n)}{=} \boxed{O(n \log_2(n))} \end{aligned}$$

## Complexité en espace

La complexité en espace d'un algorithme est l'espace mémoire qu'il a besoin d'utiliser, en fonction de la taille de l'entrée.