

# Complexité

Quentin Fortier

September 30, 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
  (* teste si e appartient à l *)
  | [] -> false
  | x::q -> x = e || mem e q
```

```
let rec mem e l = match l with
  (* teste si e appartient à l *)
  | [] -> 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
  (* teste si e appartient à l *)
  | [] -> 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 au plus  $3n$ .

# Complexité

```
let rec mem e l = match l with
  (* teste si e appartient à l *)
  | [] -> 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 au plus  $3n$ .

Remarque : Si on trouve `e` dès le début de la liste, on va effectuer moins de  $n$  opérations. Par défaut, on s'intéresse à la **complexité dans le pire cas**.

# Complexité

On considère un algorithme et  $n$  la taille de son entrée (par exemple, taille de la liste en argument).

## Complexité dans le pire cas

La complexité dans le pire cas est le nombre maximum d'opérations pour une entrée de taille  $n$ .

Quand on ne donne pas de précision, c'est de cette complexité dont on parle.



# Complexité

On considère un algorithme et  $n$  la taille de son entrée (par exemple, taille de la liste en argument).

## Complexité dans le pire cas

La complexité dans le pire cas est le nombre maximum d'opérations pour une entrée de taille  $n$ .

Quand on ne donne pas de précision, c'est de cette complexité dont on parle.

## Complexité dans le meilleur cas

La complexité dans le meilleur cas est le nombre minimum d'opérations pour une entrée de taille  $n$

# Complexité

On considère un algorithme et  $n$  la taille de son entrée (par exemple, taille de la liste en argument).

## Complexité dans le pire cas

La complexité dans le pire cas est le nombre maximum d'opérations pour une entrée de taille  $n$ .

Quand on ne donne pas de précision, c'est de cette complexité dont on parle.

## Complexité dans le meilleur cas

La complexité dans le meilleur cas est le nombre minimum d'opérations pour une entrée de taille  $n$

## complexité en moyenne

La complexité en moyenne est le nombre moyen d'opérations sur toutes les entrées de taille  $n$

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::1`
- $O(\ln(n))$  : **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.

Intérêts de la complexité :

- Comparer plusieurs algorithmes pour choisir celui dont la complexité est la plus faible (  $\implies$  plus rapide)

Intérêts de la complexité :

- Comparer plusieurs algorithmes pour choisir celui dont la complexité est la plus faible (  $\implies$  plus rapide)
- Estimer le temps d'exécution d'un algorithme  
Si on doit utiliser un algorithme en complexité  $O(n^2)$  sur une liste de taille  $n = 10^6$  avec un processeur à 1Ghz, on peut estimer le temps d'exécution à :

Intérêts de la complexité :

- Comparer plusieurs algorithmes pour choisir celui dont la complexité est la plus faible (  $\implies$  plus rapide)
- Estimer le temps d'exécution d'un algorithme  
Si on doit utiliser un algorithme en complexité  $O(n^2)$  sur une liste de taille  $n = 10^6$  avec un processeur à 1Ghz, on peut estimer le temps d'exécution à :

$$\frac{(10^6)^2}{10^9} = \frac{10^{12}}{10^9} = 1000s$$

# Exemples

```
let s = ref 0 in
for i=1 to n do
  s := !s + i
done
```

Complexité :

# Exemples

```
let s = ref 0 in  
for i=1 to n do  
  s := !s + i  
done
```

Complexité :  $O(n)$



# Exemples

```
let s = ref 0 in
for i=1 to n do
  s := !s + i
done
```

Complexité :  $O(n)$

```
for i=0 to n - 1 do
  for j=0 to i do
    print_int j
  done
done
```

Complexité :

# Exemples

```
let s = ref 0 in
for i=1 to n do
  s := !s + i
done
```

Complexité :  $O(n)$

```
for i=0 to n - 1 do
  for j=0 to i do
    print_int j
  done
done
```

Complexité :  $\sum_{i=0}^n i = O(n^2)$

# Exemples

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.

# Exemples

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 récurifs 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 récurifs de `exp_rapide a n`.

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

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 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 true  
         else if t.(m) < e then aux (m + 1) j  
         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 true  
         else if t.(m) < e then aux (m + 1) j  
         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 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 true  
      else if t.(m) < e then aux (m + 1) j  
      else aux i (m - 1)  
  in aux 0 (Array.length t - 1)
```

À chaque appel récursif, on divise au moins par 2 la taille de l'intervalle où on recherche e.

# 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 true  
          else if t.(m) < e then aux (m + 1) j  
          else aux i (m - 1)  
  in aux 0 (Array.length t - 1)
```

À chaque appel récursif, on divise au moins par 2 la taille de l'intervalle où on recherche e.

Donc au bout de  $p$  appels récursifs, la taille de cet intervalle est  $\leq \frac{n}{2^p}$



# 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 true  
          else if t.(m) < e then aux (m + 1) j  
          else aux i (m - 1)  
  in aux 0 (Array.length t - 1)
```

À chaque appel récursif, on divise au moins par 2 la taille de l'intervalle où on recherche e.

Donc au bout de  $p$  appels récursifs, la taille de cet intervalle est  $\leq \frac{n}{2^p}$

Quand  $p \geq \log_2(n)$ , il y a donc au plus  $\frac{n}{n} = 1$  élément à chercher donc la fonction s'arrête. Donc :

# 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 true  
          else if t.(m) < e then aux (m + 1) j  
          else aux i (m - 1)  
  in aux 0 (Array.length t - 1)
```

À chaque appel récursif, on divise au moins par 2 la taille de l'intervalle où on recherche e.

Donc au bout de  $p$  appels récursifs, la taille de cet intervalle est  $\leq \frac{n}{2^p}$

Quand  $p \geq \log_2(n)$ , il y a donc au plus  $\frac{n}{n} = 1$  élément à chercher donc la fonction s'arrête. Donc :

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

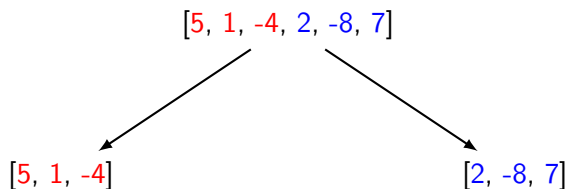
Un algorithme de tri permet de trier par ordre croissant une liste ou un tableau.

- 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)

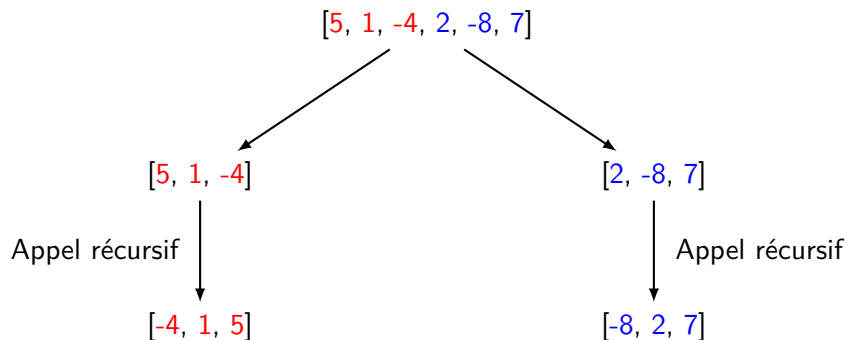
## Tri fusion : principe de l'algorithme

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

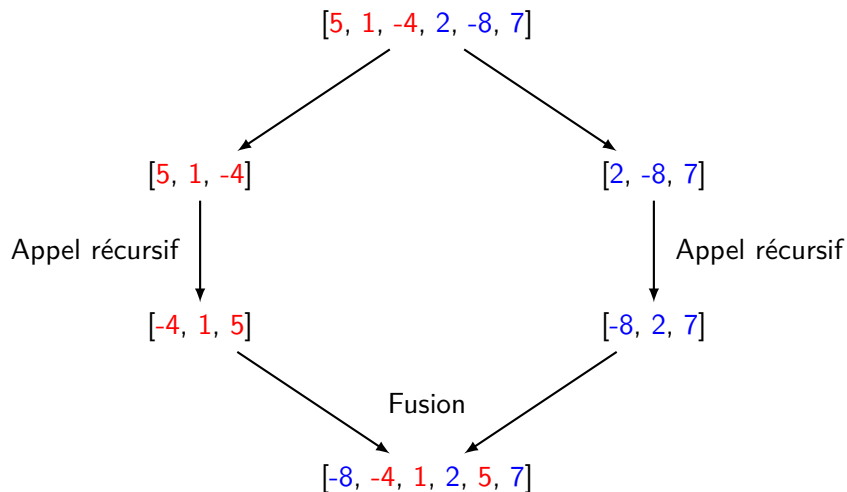
## Tri fusion : principe de l'algorithme



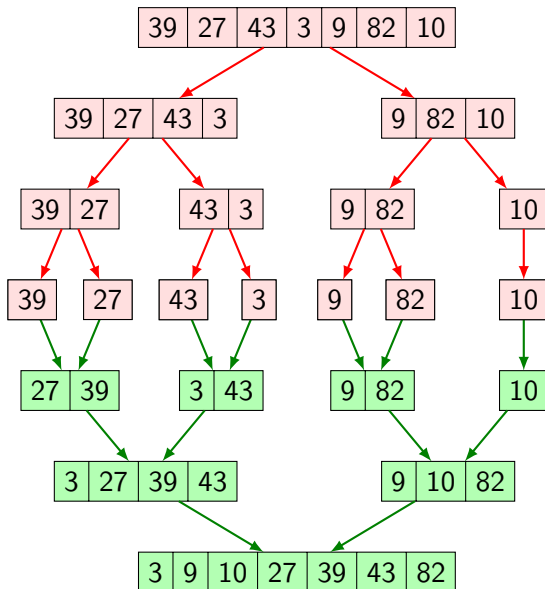
# Tri fusion : principe de l'algorithme



# Tri fusion : principe de l'algorithme



## Tri fusion : exemple





## Tri fusion : division

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 :

## Tri fusion : division

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 : 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 : 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

```
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

```
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

```
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);;
```

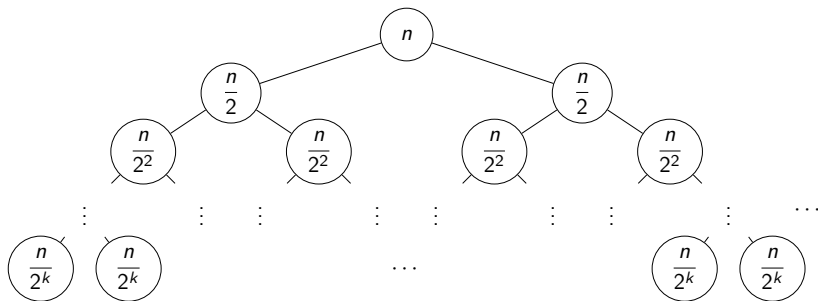
Complexité : Soit  $C(n)$  la complexité de tri  $l$  pour  $l$  de taille  $n$ .

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

où  $Kn$  est un majorant de la complexité de split plus fusion.

# Tri fusion : complexité $O(n \ln(n))$ avec l'arbre des appels récursifs

On peut représenter les appels récursifs du tri fusion sous forme d'un arbre et compter le nombre d'opération niveau par niveau :



Chaque rond (sommet) correspond à un appel récursif, avec la taille du sous-tableau à l'intérieur.





## Exercice

Avec quelle complexité pouvez-vous déterminer si une liste de taille  $n$  contient un doublon (2x le même élément) ?

## Exercice

Avec quelle complexité pouvez-vous déterminer si une liste de taille  $n$  contient un doublon (2x le même élément) ?

Méthode 1 : pour chaque élément, regarder s'il appartient à la queue.

## Exercice

Avec quelle complexité pouvez-vous déterminer si une liste de taille  $n$  contient un doublon (2x le même élément) ?

Méthode 1 : pour chaque élément, regarder s'il appartient à la queue.

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

Complexité :

## Exercice

Avec quelle complexité pouvez-vous déterminer si une liste de taille  $n$  contient un doublon (2x le même élément) ?

Méthode 1 : pour chaque élément, regarder s'il appartient à la queue.

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

Complexité : un appel à `List.mem e q` est en  $O(n)$  (car la taille de `q` est inférieure à celle de `l`).

Donc la complexité totale est  $n \times O(n) = O(n^2)$ .

## Exercice

Avec quelle complexité pouvez-vous déterminer si une liste de taille  $n$  contient un doublon (2x le même élément) ?

Méthode 2 : trier la liste puis regarder si 2 éléments consécutifs sont égaux.

## Exercice

Avec quelle complexité pouvez-vous déterminer si une liste de taille  $n$  contient un doublon (2x le même élément) ?

Méthode 2 : trier la liste puis regarder si 2 éléments consécutifs sont égaux.

```
let doublon l =  
  let rec aux = function  
    | [] | [e] -> false  
    | e1::e2::q -> e1 = e2 || aux e2::q in  
  aux (tri l)
```

Complexité :

## Exercice

Avec quelle complexité pouvez-vous déterminer si une liste de taille  $n$  contient un doublon (2x le même élément) ?

Méthode 2 : trier la liste puis regarder si 2 éléments consécutifs sont égaux.

```
let doublon l =  
  let rec aux = function  
    | [] | [e] -> false  
    | e1::e2::q -> e1 = e2 || aux e2::q in  
  aux (tri l)
```

Complexité : l'appel à `tri` est  $O(n \log(n))$ . L'appel à `aux` est en  $O(n)$ .  
Donc la complexité totale est  $O(n \log(n)) + O(n) = O(n \log(n))$ .



## Complexité en espace (ou : en mémoire)

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.

## Conseils de rédaction :

- Réfléchissez avant d'écrire du code. Si vous n'êtes pas sûr, utilisez un brouillon. Si vous vous trompez, rayez proprement.
- Utiliser si possible une couleur différente pour le code.
- Justifier toutes les complexités. Mais si la complexité est évidente (une boucle for par exemple), une ligne de justification suffit.