# Cours 2 : Fonctions récursives, analyse récursive, terminaison et complexité

2019 - 2020

# La récursivité

- Les fonctions peuvent être **récursives**.
- L'identificateur de la fonction peut apparaître dans sa propre définition.
- Ce mécanisme est similaire aux objets définis par récurrence en mathématiques.

## Syntaxe par l'exemple

- Définition de $n!$ par la suite récurrente (où le symbole ! apparaît des deux côtés de l'équation définissant $n!$) :

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \ \ (n \neq 0) \end{aligned}$$

- Définition récursive de la fonction factorielle :

```
let rec fact n =
 if n = 0 then 1
          else n * fact (n - 1)
```

- La fonction `fact` contient un **appel récursif**.
- Pour qu'un identificateur puisse apparaître dans sa propre définition, il faut utiliser le mot-clé `rec`.

## Rappel : Sémantique des appels

- Rappel de la fonction `carre`
  - `let carre x = x*x`
  - On a l'équivalence entre `carre (a+1)` et `let x = (a + 1) in x * x`.
- On peut également interpréter les appels récursifs avec des `let`.
- ```
    fact (2+1)
  = let n = 2+1
    in if n = 0 then 1 else n * fact (n - 1)
  = let n = 2+1
    in if n = 0 then 1 else n * let n = n - 1
                                in if n = 0 then 1 else n * fact (n - 1)
  = ...
  ```
  ```

## Sémantique des appels récursifs

- La vision en termes d'environnement et d'**arbre d'appel** permet de mieux saisir les mécanismes mis en jeu.
- Exemple de la factorielle

    fact (2+1)

## Sémantique des appels récursifs

- La vision en termes d'environnement et d'**arbre d'appel** permet de mieux saisir les mécanismes mis en jeu.
- Exemple de la factorielle

  fact (2+1)  appel par valeur dans l'environnement initial I

## Sémantique des appels récursifs

- La vision en termes d'environnement et d'**arbre d'appel** permet de mieux saisir les mécanismes mis en jeu.
- Exemple de la factorielle

| | |
|---|---|
| fact (2+1) | appel par valeur dans l'environnement initial I |
| | - calcul du paramètre réel : 3 |

## Sémantique des appels récursifs

- La vision en termes d'environnement et d'**arbre d'appel** permet de mieux saisir les mécanismes mis en jeu.
- Exemple de la factorielle

fact (2+1) appel par valeur dans l'environnement initial I
- calcul du paramètre réel : 3
- construction d'un environnement E1=(n,3);I

## Sémantique des appels récursifs

- La vision en termes d'environnement et d'**arbre d'appel** permet de mieux saisir les mécanismes mis en jeu.
- Exemple de la factorielle

| | |
|---|---|
| fact (2+1) | appel par valeur dans l'environnement initial I |
| | - calcul du paramètre réel : 3 |
| | - construction d'un environnement E1=(n,3);I |
| | - calcul dans E1 de `if n=0 then 1 else n*(fact (n-1))` |

## Sémantique des appels récursifs

- La vision en termes d'environnement et d'**arbre d'appel** permet de mieux saisir les mécanismes mis en jeu.
- Exemple de la factorielle

fact (2+1) appel par valeur dans l'environnement initial I
- calcul du paramètre réel : 3
- construction d'un environnement E1=(n,3);I
- calcul dans E1 de `if n=0 then 1 else` $\boxed{\texttt{n*(fact (n-1))}}$

## Sémantique des appels récursifs

- La vision en termes d'environnement et d'**arbre d'appel** permet de mieux saisir les mécanismes mis en jeu.
- Exemple de la factorielle

fact (2+1)  appel par valeur dans l'environnement initial I
- calcul du paramètre réel : 3
- construction d'un environnement E1=(n,3);I
- calcul dans E1 de `if n=0 then 1 else` $\boxed{\texttt{n*(fact (n-1))}}$

n*(fact (n-1))  appel par valeur dans l'environnement E1
- calcul du paramètre réel : $n - 1 = 2$
- construction d'un environnement E2=(n,2);E1
- calcul dans E2 de `if n=0 then 1 else` $\boxed{\texttt{n*(fact (n-1))}}$

## Sémantique des appels récursifs

- La vision en termes d'environnement et d'**arbre d'appel** permet de mieux saisir les mécanismes mis en jeu.
- Exemple de la factorielle

fact (2+1) appel par valeur dans l'environnement initial I
- calcul du paramètre réel : 3
- construction d'un environnement E1=(n,3);I
- calcul dans E1 de `if n=0 then 1 else` $\boxed{\texttt{n*(fact (n-1))}}$

n*(fact (n-1)) appel par valeur dans l'environnement E1
- calcul du paramètre réel : $n - 1 = 2$
- construction d'un environnement E2=(n,2);E1
- calcul dans E2 de `if n=0 then 1 else` $\boxed{\texttt{n*(fact (n-1))}}$

n*(fact (n-1)) appel par valeur dans l'environnement E2
- calcul du paramètre réel : $n - 1 = 1$
- construction d'un environnement E3=(n,1);E2
- calcul dans E3 de `if n=0 then 1 else` $\boxed{\texttt{n*(fact (n-1))}}$

## Sémantique des appels récursifs

- La vision en termes d'environnement et d'**arbre d'appel** permet de mieux saisir les mécanismes mis en jeu.
- Exemple de la factorielle

fact (2+1) appel par valeur dans l'environnement initial I
- calcul du paramètre réel : 3
- construction d'un environnement E1=(n,3);I
- calcul dans E1 de `if n=0 then 1 else` $\boxed{\texttt{n*(fact (n-1))}}$

n*(fact (n-1)) appel par valeur dans l'environnement E1
- calcul du paramètre réel : $n-1=2$
- construction d'un environnement E2=(n,2);E1
- calcul dans E2 de `if n=0 then 1 else` $\boxed{\texttt{n*(fact (n-1))}}$

n*(fact (n-1)) appel par valeur dans l'environnement E2
- calcul du paramètre réel : $n-1=1$
- construction d'un environnement E3=(n,1);E2
- calcul dans E3 de `if n=0 then 1 else` $\boxed{\texttt{n*(fact (n-1))}}$

n*(fact (n-1)) appel par valeur dans l'environnement E3
- calcul du paramètre réel : $n-1=0$
- construction d'un environnement E4=(n,0);E3
- calcul dans E4 de `if n=0 then` $\boxed{\texttt{1}}$ `else ...`

## Sémantique des appels récursifs

- La vision en termes d'environnement et d'**arbre d'appel** permet de mieux saisir les mécanismes mis en jeu.
- Exemple de la factorielle

fact (2+1) appel par valeur dans l'environnement initial I
- calcul du paramètre réel : 3
- construction d'un environnement E1=(n,3);I
- calcul dans E1 de `if n=0 then 1 else` $\boxed{\texttt{n*(fact (n-1))}}$

n*(fact (n-1)) appel par valeur dans l'environnement E1
- calcul du paramètre réel : $n - 1 = 2$
- construction d'un environnement E2=(n,2);E1
- calcul dans E2 de `if n=0 then 1 else` $\boxed{\texttt{n*(fact (n-1))}}$

n*(fact (n-1)) appel par valeur dans l'environnement E2
- calcul du paramètre réel : $n - 1 = 1$
- construction d'un environnement E3=(n,1);E2
- calcul dans E3 de `if n=0 then 1 else` $\boxed{\texttt{n*(fact (n-1))}}$

n*(fact (n-1)) appel par valeur dans l'environnement E3
- calcul du paramètre réel : $n - 1 = 0$
- construction d'un environnement E4=(n,0);E3
- calcul dans E4 de `if n=0 then` $\boxed{\texttt{1}}$ `else ...`

multiplications en suspens!

1

## Sémantique des appels récursifs

- La vision en termes d'environnement et d'**arbre d'appel** permet de mieux saisir les mécanismes mis en jeu.
- Exemple de la factorielle

fact (2+1) appel par valeur dans l'environnement initial I
- calcul du paramètre réel : 3
- construction d'un environnement E1=(n,3);I
- calcul dans E1 de `if n=0 then 1 else` $\boxed{\texttt{n*(fact (n-1))}}$

n*(fact (n-1)) appel par valeur dans l'environnement E1
- calcul du paramètre réel : $n-1=2$
- construction d'un environnement E2=(n,2);E1
- calcul dans E2 de `if n=0 then 1 else` $\boxed{\texttt{n*(fact (n-1))}}$

n*(fact (n-1)) appel par valeur dans l'environnement E2
- calcul du paramètre réel : $n-1=1$
- construction d'un environnement E3=(n,1);E2
- calcul dans E3 de `if n=0 then 1 else` $\boxed{\texttt{n*(fact (n-1))}}$

1*1

n*(fact (n-1)) appel par valeur dans l'environnement E3
- calcul du paramètre réel : $n-1=0$
- construction d'un environnement E4=(n,0);E3
- calcul dans E4 de `if n=0 then` $\boxed{\texttt{1}}$ `else ...`

1

### Sémantique des appels récursifs

- La vision en termes d'environnement et d'**arbre d'appel** permet de mieux saisir les mécanismes mis en jeu.
- Exemple de la factorielle

fact (2+1) appel par valeur dans l'environnement initial I
- calcul du paramètre réel : 3
- construction d'un environnement E1=(n,3);I
- calcul dans E1 de `if n=0 then 1 else` $\boxed{\texttt{n*(fact (n-1))}}$

n*(fact (n-1)) appel par valeur dans l'environnement E1
- calcul du paramètre réel : $n - 1 = 2$
- construction d'un environnement E2=(n,2);E1
- calcul dans E2 de `if n=0 then 1 else` $\boxed{\texttt{n*(fact (n-1))}}$

2*1

n*(fact (n-1)) appel par valeur dans l'environnement E2
- calcul du paramètre réel : $n - 1 = 1$
- construction d'un environnement E3=(n,1);E2
- calcul dans E3 de `if n=0 then 1 else` $\boxed{\texttt{n*(fact (n-1))}}$

1*1

n*(fact (n-1)) appel par valeur dans l'environnement E3
- calcul du paramètre réel : $n - 1 = 0$
- construction d'un environnement E4=(n,0);E3
- calcul dans E4 de `if n=0 then` $\boxed{\texttt{1}}$ `else ...`

1

## Sémantique des appels récursifs

- La vision en termes d'environnement et d'**arbre d'appel** permet de mieux saisir les mécanismes mis en jeu.
- Exemple de la factorielle

fact (2+1) appel par valeur dans l'environnement initial I
- calcul du paramètre réel : 3
- construction d'un environnement E1=(n,3);I
- calcul dans E1 de `if n=0 then 1 else` $\boxed{\texttt{n*(fact (n-1))}}$

3*2

n*(fact (n-1)) appel par valeur dans l'environnement E1
- calcul du paramètre réel : $n - 1 = 2$
- construction d'un environnement E2=(n,2);E1
- calcul dans E2 de `if n=0 then 1 else` $\boxed{\texttt{n*(fact (n-1))}}$

2*1

n*(fact (n-1)) appel par valeur dans l'environnement E2
- calcul du paramètre réel : $n - 1 = 1$
- construction d'un environnement E3=(n,1);E2
- calcul dans E3 de `if n=0 then 1 else` $\boxed{\texttt{n*(fact (n-1))}}$

1*1

n*(fact (n-1)) appel par valeur dans l'environnement E3
- calcul du paramètre réel : $n - 1 = 0$
- construction d'un environnement E4=(n,0);E3
- calcul dans E4 de `if n=0 then` $\boxed{\texttt{1}}$ `else ...`

1

## Fonctions récursives - exemples

- À partir du schéma suivant, donner une fonction qui calcule le $n$-ème itéré de la suite de Fibonacci.

$$\begin{aligned} f_0 &= 0 \\ f_1 &= 1 \\ f_n &= f_{n-1} + f_{n-2} \ \ (n \neq 0, 1) \end{aligned}$$

- À partir du schéma suivant, donner une fonction qui calcule la puissance $n$-ème entière d'un nombre flottant.

$$\begin{aligned} x^0 &= 1 \\ x^n &= x \times x^{n-1} \ \ (n \neq 0) \end{aligned}$$

- La récursion peut aussi décroître plus rapidement que par pas fixes, ex. la puissance “indienne”:

$$\begin{aligned} x^0 &= 1 \\ x^{2k} &= (x \times x)^k \\ x^{2k+1} &= x \times (x \times x)^k \end{aligned}$$

# Conception des algorithmes

♪♪♪

### Raffinage

- **But** : Définir un enchaînement d'opérations élémentaires pour passer de manière efficace d'une donnée $D$ à un résultat $R$.
- **Méthodologie : décomposition de problèmes**
- On emploie aussi le terme de **raffinage**.

### Trois façons de procéder :

- Méthode **descendante** : on décompose au fur et à mesure des difficultés.
- Méthode **ascendante** : on commence par définir des (librairies de) fonctions utiles, puis on compose.
- Souvent un mélange des deux.

### La décomposition fonctionnelle

La décomposition **fonctionnelle** d'un problème repose sur l'introduction des structures de contrôle déjà vues :

- définition,
- conditionnelle,
- filtrage,
- appel/composition de fonctions
- et surtout sur l'**analyse récursive**.

### Récurrence

Lorsque nous disposons d'un schéma par récurrence comme spécification, nous pouvons simplement transcrire le schéma en fonction récursive, comme dans les exemples de la factorielle, fibonacci, etc...

### Analyse récursive d'un problème

Sinon, nous allons chercher à construire une fonction récursive

$$f : D \to D'$$

en identifiant:

- le domaine d'entrée (type + précondition éventuelle) sur lequel $f$ s'applique,
- le domaine de sortie (type + postcondition éventuelle) dont $f$ produit des valeurs.

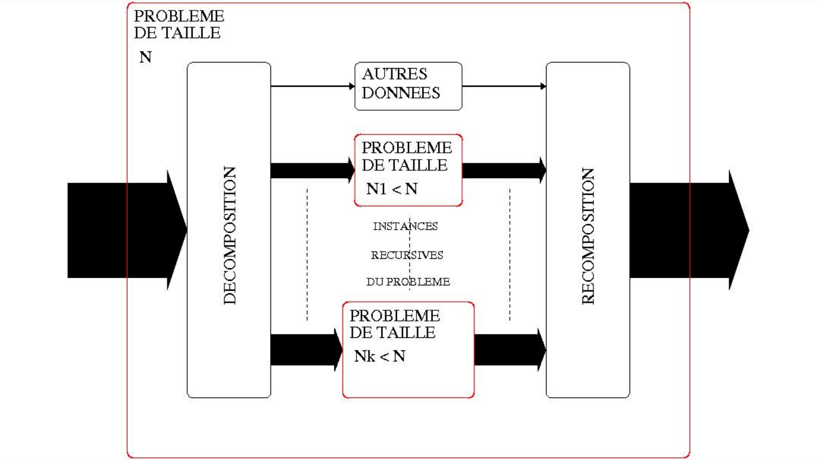
## Analyse récursive d'un problème

Il reste alors à déterminer :

- un ordre $\prec$ **bien fondé** sur les valeurs de $\mathtt{D}$, i.e. qui ne possède pas de chaîne infinie décroissante.
  - ordre est induit par une mesure de la taille du problème
  - i.e. une fonction mathématique de $\mathtt{D}$ dans $\mathbb{N}$, telle que $x \prec y \triangleq \mathit{taille}(x) < \mathit{taille}(y)$.
- les valeurs de $\mathtt{D}$ pour lesquelles $\mathtt{f}$ est calculable directement sans récursivité: **les cas terminaux**. Il doit s'agir des valeurs minimales pour l'ordre considéré (ou des valeurs de taille minimale, le plus souvent 0 ou 1).
- pour les autres valeurs de $\mathtt{D}$, on cherche une expression de $\mathtt{f\ x}$ en fonction de $\mathtt{f\ x}_1, \ldots, \mathtt{f\ x}_n$, avec $\forall i, \mathtt{x}_i \prec \mathtt{x}$ : **les cas généraux ou récursifs**.

## Analyse récursive d'un problème - schéma global

Cas généraux ou récursifs :



**Analyse récursive d'un problème**

Les 3 questions à se poser en début d'analyse :

- Quelle est la taille du problème ?
- Quels sont les cas terminaux et les résultats associés ?
- Si je sais **(ou suppose que je sais)** résoudre un problème de taille $N - 1$ ($N - 2, N - 3, \ldots$), comment puis-je résoudre un problème de taille $N$ ?

## Argument de terminaison

- Un des problèmes principaux lors de l'écriture d'une fonction récursive est de bien s'assurer de la **terminaison** de celle-ci.
- L'utilisation de la relation d'ordre ou de la taille sur les données d'entrée permet justement de garantir cette terminaison.

## Éléments de cette analyse sur les fonctions récursives déjà vues

| | taille | cas terminaux | cas généraux |
|---|---|---|---|
| `fact` | $\textit{taille}(n) \triangleq n$ | $n = 0$ | $n - 1$ |
| `fib` | $\textit{taille}(n) \triangleq n$ | $n = 0, 1$ | $n - 1, n - 2$ |

Nous avons une suite strictement décroissante d'entiers positifs ou nuls, la terminaison est donc assurée (à condition d'avoir écrit comme précondition à ces 2 fonctions que $n \in \mathbb{N}$).

**Exemple : P.G.C.D.**

Voici une spécification du PGCD de deux nombres entiers naturels.

$$\begin{aligned}
pgcd(x,0) &= x\\
pgcd(0,y) &= y\\
pgcd(x,y) &= pgcd(x \; mod \; y, y) \; (x \geq y)\\
pgcd(x,y) &= pgcd(x, y \; mod \; x) \; (y > x)
\end{aligned}$$

- Vérifier que ce schéma se prête à une décomposition récursive.
- Établir le contrat et les tests de la fonction correspondante.
- Écrire la fonction correspondante.

| | taille | cas terminaux | cas généraux |
|---|---|---|---|
| `pgcd` | $\mathit{taille}(x, y) \triangleq x + y$ | $(\{0\} \times \mathbb{N}) \cup (\mathbb{N} \times \{0\})$ | $(x \text{ mod } y, y)\ (x \geq y)$<br>$(x, y \text{ mod } x)\ (y > x)$ |

```
(*
  pgcd : int -> int -> int
  calcule le pgcd de deux entiers positifs
  Parametres x, y : int, nombres dont on veut le pgcd
  Resultat : int, pgcd de x et y
  Precondition : x, y positfs
  Post-condition: resultat positif
*)

let rec pgcd x y =
  if y = 0 then x
  else if x = 0 then y
  else if x >= y then pgcd (x mod y) y
  else pgcd x (y mod x)

let%test _ = pgcd 0 0 = 0
let%test _ = pgcd 0 15 = 15
let%test _ = pgcd 32 0 = 32
let%test _ = pgcd 11 17 = 1
let%test _ = pgcd 42 30 = 6
```

# Complexité des algorithmes

### Définition

La **complexité** est une mesure *abstraite* du temps d'exécution d'un algorihme $A$ en fonction de la taille des données à traiter.

### Intérêts

- prévoir **les temps d'exécution** (ou la mémoire nécessaire) quelque soit la machine et le langage utilisés,
- savoir si des données sont **effectivement traitables**.

## Problème vs algorithme

- À un problème donné on peut associer plusieurs algorithmes. Il faut faire un choix en fonction de :
  - la facilité d'écriture, de mise-au-point, de relecture, de maintenance:
    - non mesurable,
    - à privilégier si l'algorithme est utilisé **peu de fois** ou sur des données de **petite taille**.
  - exécution rapide et/ou occupation mémoire faible
    - à privilégier si l'algorithme est exécuté **de nombreuses fois**, sur des données de **taille importante** ou en **temps contraint**.
- Il y a donc un compromis à trouver entre ressources humaines et ressources machine.
- On dira qu'un algorithme est **optimal** s'il n'existe pas d'algorithme de complexité inférieure.

## Taille du problème

- Taille (à définir) de la structure de données manipulée par la fonction, dans $\mathtt{D} \to \mathbb{N}$,
  - en général le nombre d'éléments primitifs,
  - ou la profondeur maximum (i.e. la distance à l'élément le plus "éloigné"),
  - ou bien encore une projection de ces grandeurs sur les sous-structures coûteuses à parcourir / construire.
- Cette taille, quoique largement arbitraire, doit tout de même être monotone et décroissante, i.e. lorsqu'on décompose récursivement la structure de données, la taille des sous-structures doit être strictement inférieure.
- On peut utiliser la même taille à la fois pour prouver la terminaison et pour mesurer la complexité.

## Principe de calcul

- On cherche une fonction $C^A(n)$ telle que **le temps d'exécution** de $A$ sur une donnée de taille $n$ soit **proportionnel** à $C^A(n)$.
- On peut s'intéresser à trois fonctions pour une donnée de taille $n$ :
  - $C^A_{min}(n)$ dans le meilleur des cas
  - $C^A_{moy}(n)$ en moyenne (de quoi ??)
  - $C^A_{max}(n)$ dans le pire des cas
- La mesure de complexité fondamentale concerne l'exécution du pire cas, celui (ou ceux) pour lequel les données sont pathologiquement néfastes. Il est toujours plus prudent d'adopter cette position pessimiste, si l'on se réfère à la célèbre loi de Murphy.

**La complexité du cas moyen**

On invoque souvent la complexité du cas moyen.

Exemple de l'algorithme du tri rapide :

- pire cas, complexité quadratique
- cas moyen (celui où les données d'entrées sont à peu près bien mélangées), complexité $n * log(n)$
- recommandé de consacrer un petit temps fixe au mélange aléatoire (donc rapide) des données à trier par le tri rapide, pour se retrouver dans le cas moyen et éviter le pire cas !

### La complexité du cas moyen

- la complexité moyenne est problématique
- elle considère implicitement que tous les cas d'entrées se produisent équitablement avec la même probabilité, ce qui est très improbable en pratique.
- Par exemple, il est fréquent d'être amené à trier une structure de données qui est presque totalement déjà dans le bon ordre, à part quelques éléments fraîchement introduits.

En conclusion, à moins de posséder une loi de distribution pour les données en entrée, la complexité moyenne n'a pas de réelle signification et est à proscrire.

**La complexité amortie**

- elle mesure la complexité d'un algorithme sur la durée
- i.e. en temps cumulé après plusieurs invocations sur une structure de données dont on suit l'évolution.

## Équation de complexité

- On se restreint aux fonctions à 1 paramètre.
- On choisit une ou plusieurs **opérations fondamentales**, telles que le temps d'exécution de $A$ soit proportionnel au nombre de fois que ces opérations sont exécutées.
- Cadre de programmation fonctionnelle : nombre d'appels de la fonction représentant $A$.
- Une **équation de complexité** se présente sous la forme d'une suite (à 1 paramètre) $C(n)$ définie par récurrence sur la taille $n$ du problème.
- Une telle équation est calquée sur le comportement récursif de la fonction à analyser et est donc simple à construire.

## Équation de complexité - Exemple Fibonacci

```
let rec fib n =
 if n < 2 then
        else fib (n - 1) + fib (n - 2)
```

- Taille du problème : valeur de $n$
- Complexité en terme : d'appels à `fib`
- Equations
  - $C(0) = C(1) = 1$
  - $C(n) = 1 + C(n-1) + C(n-2)$
- Résolution
  - On peut montrer par récurrence que $C(n) = 2 * fib(n+1) - 1$
  - Formule de Binet : $fib(n) = 1/\sqrt{5}(\Phi^n - (-1/\Phi)^n)$
  - Complexité exponentielle

## Résolution des équations de complexité : encadrement

- Calcul d'une expression analytique de $C(n)$ impossible

$\Rightarrow$ encadrement ou simplement une borne supérieure

- Construction de $s(n)$ et $S(n)$, suites intégrables telles que :

$$s(n) \leq C(n) \leq S(n)$$

- si possible, $s(n)$ et $S(n)$ avec même comportement pour déterminer le comportement asymptotique de $C(n)$ (par ex. $s(n) = S(n-1)$).

## Résolution par encadrement : Exemple de la puissance ”indienne”

```
let rec puissance x n =
 if n = 0 then 1.0 else
 if n mod 2 = 0 then       puissance (x *. x) (n / 2)
                     else x *. puissance (x *. x) (n / 2)
```

- Taille du problème : valeur de $n$
- Complexité en terme : de nombre de multiplications
- Equations
  - $C(0) = 0$
  - $C(2n) = 1 + C(n)$
  - $C(2n + 1) = 2 + C(n)$
- Résolution
  - Sachant que $2^{\lceil \log_2 n \rceil - 1} < n \leq 2^{\lceil \log_2 n \rceil}$
  - On a $C(2^{\lceil \log_2 n \rceil - 1}) < C(n) \leq C(2^{\lceil \log_2 n \rceil})$ (car $C(n)$ monotone et croissante
  - On déduit des équations : $C(2^0) = C(1) = 2 + C(0) = 2$ et $C(2^{n+1}) = 1 + C(2^n)$
  - Donc : $C(2^n) = n + 2$
  - L'encadrement de $C(n)$ devient alors : $\lceil log_2 n \rceil + 2 - 1 < C(n) \leq \lceil \log_2 n \rceil + 2$

$\Rightarrow$ Complexité logarithmique

$\boxed{\text{Pour votre culture générale seulement.}}$

## Résolution des équations - Principe général

- établir une correspondance de $C(n)$ vers les séries formelles, par l'intermédiaire des **fonctions génératrices**
- on associe à toute suite entière $C(n)$ la série formelle / fonction

$$f(z) = \Sigma_{i \in \mathbb{N}} C(i) z^i.$$

- ou la série exponentielle

$$g(z) = \Sigma_{i \in \mathbb{N}} (C(i)/i!) z^i$$

- la relation de récurrence des $C(n)$ permet de définir une équation sur $f(z)$, quelques fois une équation différentielle.
- si on peut intégrer cette équation, alors on aura une fonction analytique de la complexité.

## Notations de Landau

- $O(g)$ pour représenter une “classe” de complexité, i.e. l’ensemble des fonctions dont le temps d’éxécution est asymptotiquement inférieur à $g(n)$.
$$f = O(g) \triangleq \exists N, C. \forall n > N. f(n) \leq C * g(n)$$
- $\Theta(g)$ pour représenter l’ensemble des fonctions dont le temps d’éxécution est asymptotiquement proportionnel à $g(n)$.
$$f = \Theta(g) \triangleq f = O(g) \wedge g = O(f)$$
- $\Theta(3n^2) = \Theta(10n^2 - 4n) = \Theta(n^2)$
- Les fonctions $e^n$, $\log n$ et $n$ sont dans des classes différentes, ainsi que leurs produits, rapports et puissances entières ou fractionnaires.
- La complexité s’exprimera toujours par un ensemble de fonctions simples :
$\log_p n,\ n \times \log_p n,\ \sqrt{n},\ n,\ n^p,\ 2^n, \ldots$

## Comparaison des complexités

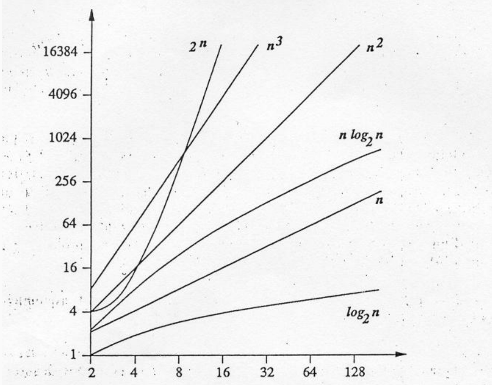



Figure 1. Rapidités de croissance comparées de certaines fonctions usuelles.

## Exemple de la fonction puissance

- On va comparer les différentes versions de la fonction puissance.
- Le critère de taille retenu sera le nombre de multiplications.

## Exercice : la puissance naïve

- ```
  let rec puissance x n = if n = 0 then 1.0 else x *. puissance x (n - 1)
  ```

- Poser les équations de complexité.
- Quelle est, à votre avis, la complexité de cette fonction ?

## Exercice : la puissance "indienne" avec duplication des calculs

- ```
  let rec puissance x n =
   if n = 0 then 1.0 else
   if n mod 2 = 0 then      puissance x (n / 2) *. puissance x (n / 2)
                  else x *. puissance x (n / 2) *. puissance x (n / 2)
  ```

- Poser les équations de complexité.
- Quelle est, à votre avis, la complexité de cette fonction ?

# Remarque Finale

Dans la plupart des cas, on n'aura ni le temps, ni la patience, ni le talent pour calculer la complexité de cette façon rigoureuse (voire rigoriste). On devra user de l'intuition.

# Récursivité terminale

♪♪♪

### Définition

- Un appel récursif est terminal s'il n'est pas argument d'un opérateur ou d'une fonction.
- Une fonction est récursive terminale si et seulement si tous ses appels récursifs sont terminaux.

### Exemple de la fonction *pgcd*

- *pgcd* est récursive terminale

```
let rec pgcd x y =
  if x = 0  then y else
  if y = 0  then x else
  if x >= y then pgcd (x mod y) y
            else pgcd x         (y mod x)
```

- La conséquence de la terminalité est que :

$$pgcd\ 45\ 24 = pgcd\ 21\ 24 = pgcd\ 21\ 3 = pgcd\ 0\ 3 = 3$$

- Le résultat 3 est connu au moment du dernier appel récursif.

### Exemple de la fonction *fact*

- *fact* n'est pas récursive terminale.

```
let rec fact n =
 if n = 0 then 1 else n * fact (n - 1)
```

- La conséquence de la non-terminalité est que :

$$\mathit{fact}\ 4 = 4 \times (\mathit{fact}\ 3) = 4 \times 3 \times (\mathit{fact}\ 2) = 4 \times 3 \times 2 \times (\mathit{fact}\ 1) = 4 \times 3 \times 2 \times 1 \times (\mathit{fact}\ 0)$$

- Après le dernier appel récursif, il reste à faire les multiplications.

### Intérêt de la récursivité terminale

- L'évaluation est plus simple et peut être optimisée.
- Une fonction récursive terminale est directement équivalente à une **boucle** en programmation impérative.
- Utilisation constante de la pile
  $\Rightarrow$ évite les débordements *(stack overflows)* qui peuvent survenir en cas d'utilisation sur des données de grande taille.

## code ADA du pgcd

```
function pgcd(x, y : in natural) return natural is
 xtmp      : natural;
 ytmp      : natural;
 resultat  : natural;
 begin
  -- xtmp et ytmp simulent les parametres
  -- de la fonction recursive
  xtmp := x;
  ytmp := y;
  -- detection des cas terminaux
  -- condition de sortie de boucle
  while (xtmp <> 0 and ytmp <> 0) loop
  -- traitement des cas recursifs
  -- par affectations des 'parametres' xtmp et ytmp
    if xtmp > ytmp then
      xtmp := xtmp mod ytmp;
    else
      ytmp := ytmp mod xtmp;
    end if;
  end loop;
  -- traitement des cas terminaux identique
  if xtmp <> 0 then
    resultat := xtmp;
  else
    resultat := ytmp;
  end if;
  return resultat;
 end pgcd;
```

# Récursivité terminale

### Comment rendre une fonction terminale ?

- introduction de paramètres supplémentaires, appelés **accumulateurs**, qui représentent les résultats de chacun de ces appels,
- adaptation de la fonction
  - le cas de base est maintenant pour la valeur d'appel
  - modification du cas général pour introduire l'accumulateur
- on ne change pas le type de la fonction $\Rightarrow$ introduction d'une fonction auxiliaire.

# Récursivité terminale

**Exemple : fact récursif terminal**

- Écrire la fonction factorielle de façon récursive terminale
- Solution :

# Récursivité terminale

## Exemple : fact récursif terminal

- Écrire la fonction factorielle de façon récursive terminale
- Solution :

```
let fact n =
 let rec fact_term p fact_p =
   if p >= n then fact_p
   else fact_term (p + 1) ((p+1) * fact_p)
 in
 fact_term 0 1
```