Cours 2 : Fonctions récursives, analyse récursive, terminaison et complexité, récursivité terminale

1 La récursivité

Les fonctions peuvent être **récursives**, i.e. l'identificateur de la fonction peut apparaître dans sa propre définition. Ce mécanisme est similaire aux objets définis par récurrence en mathématique.

Par exemple, la définition de $n!$ par la suite récurrente (où le symbole ! apparaît des deux côtés de l'équation définissant $n!$) :

$$0! = 1$$
$$n! = n \times (n-1)! \quad (n \neq 0)$$

correspond naturellement à une 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`.

Sémantique des appels récursifs On peut toujours interpréter les appels avec des `let`. Ce qui donne par exemple :

```
  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)
= ...
```

Le problème est que le dépliage peut toujours être prolongé, puisque `fact` apparaît toujours dans sa définition. Il sera difficile de s'apercevoir que dans la suite des définitions imbriquées de n, l'un d'entre eux finira par être nul, et donc que la branche "else" de sa conditionnelle qui contient l'appel récursif ne sera pas exécutée, et qu'il est donc inutile de déplier plus avant la définition de `fact` dans cette même conditionnelle. Ceci n'est pas satisfaisant.

La vision en termes d'environnement et d'**arbre d'appels** permet de mieux saisir les mécanismes mis en jeu.

2 Conception des algorithmes : analyse récursive d'un problème

But Définir un enchaînement d'opérations élémentaires pour passer de manière efficace d'une donnée D à un résultat R.

Méthodologie : décomposition de problèmes Comme en mathématiques, quand un théorème est complexe on démontre des lemmes. 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 démontrer des lemmes, puis on compose.

— Souvent un mélange des deux.

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.

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.

Sinon, nous allons chercher à construire une fonction récursive `f : D -> D'` en identifiant :

— le domaine d'entrée (type + précondition éventuelle) sur lequel `f` s'applique

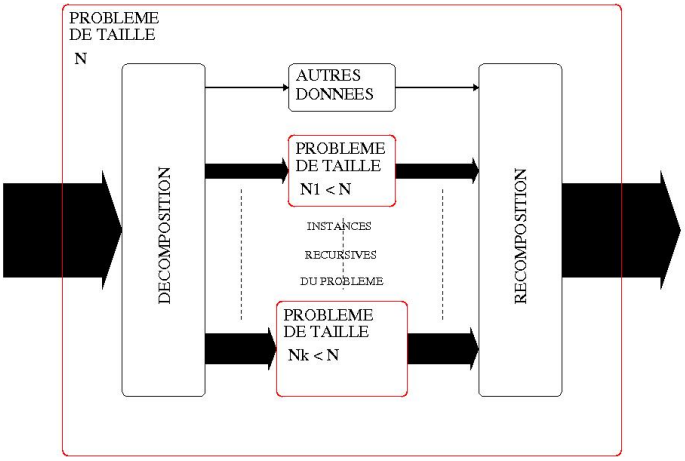
— le domaine de sortie (type + postcondition éventuelle) dont `f` produit des valeurs.

Il reste alors à déterminer :

— un ordre $\preceq$ **bien fondé** sur les valeurs de `D`, i.e. qui ne possède pas de chaîne infinie décroissante. Souvent, cet ordre est induit par une mesure de la taille du problème, i.e. une fonction mathématique de `D` dans N, telle que $x \preceq y \triangleq taille(x) \leq taille(y)$.

— les valeurs de `D` pour lesquelles `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 `D`, on cherche une expression de `f x` en fonction de `f` $\mathtt{x}_1, \ldots,$ `f` $\mathtt{x}_n$, avec $\forall i, \mathtt{x}_i \prec \mathtt{x}$: **les cas généraux ou récursifs**. Ces cas sont illustrés par la figure suivante.



Pour savoir si une décomposition récursive est envisageable (si l'on dispose de la taille), il suffit de se poser la question : est-ce que savoir résoudre un problème de taille N quelconque m'aiderait à résoudre des problèmes de taille supérieure, typiquement $N+1$ ou $2N$? Il s'agit d'une analogie avec le passage $n \to n+1$ dans une démonstration par récurrence.

3 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.

On retrouve bien sûr les éléments de cette analyse sur les fonctions récursives déjà vues.

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

Nous avons une suite strictement décroissante d'entiers positifs ou nuls, la terminaison est donc assurée.

4 Complexité des algorithmes

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**.

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.

Taille du problème Mesure (à définir) de la structure de données manipulée par la fonction, dans $\mathtt{D} \to \mathrm{N}$, 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 mesure, 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.

On invoque souvent la complexité du cas moyen, par exemple pour l'algorithme du tri rapide, dont le pire cas est quadratique, mais dont le cas moyen, celui où les données d'entrées sont à peu près bien mélangées, est en $n * log(n)$. D'ailleurs, il est pseudo-paradoxalement 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 !

Quoi qu'il en soit, la complexité moyenne est plus problématique, car 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.

Enfin, il reste la **complexité amortie**, abordée plus tard, qui 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.

Optimalité On dira qu'un algorithme est **optimal** s'il n'existe pas d'algorithme de complexité inférieure.

4.1 Equation de complexité

On se restreint aux fonctions à 1 paramètre. La méthode reste essentiellement la même lorsqu'on a plusieurs paramètres.

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. La plupart du temps, dans un cadre de programmation fonctionnelle, on comptabilisera les appels de fonctions (en tout cas, 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.

Exemple Fibonacci.

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

Si on prend comme taille du problème la valeur de n et que l'on compte les appels de `fib`, on a les équations suivantes :

$$C(0) = C(1) = 1, \quad C(n) = 1 + C(n-1) + C(n-2)$$

i.e. $C(n)$ ressemble fortement à `fib(n)`. On peut montrer ici par récurrence que :

$$C(n) = 2 * fib(n+1) - 1$$

Or la formule de Binet donne :

$$fib(n) = 1/\sqrt{(5)}(\Phi^n - \Phi'^n)$$

où Φ nombre d'or, i.e. $\Phi^2 = \Phi + 1$ et $\Phi' = -1/\Phi$.

Or, $\Phi = (1 + \sqrt{5})/2 = 1,6180339887...$, donc la complexité est exponentielle.

Encadrement Malheureusement, le calcul d'une expression analytique de $C(n)$ est en général impossible, et on doit alors se contenter d'un encadrement ou simplement d'une borne supérieure. On va donc essayer de construire quelque chose comme : $s(n) \leq C(n) \leq S(n)$, où les suites $s(n)$ et $S(n)$ sont intégrables. Il est préférable, afin de déterminer le comportement asymptotique de $C(n)$, de faire que $s(n)$ et $S(n)$ aient ce même comportement, par exemple lorsque $s(n) = S(n-1)$...

Exemple Puissance 'indienne"

Le critère de taille retenu sera le nombre de multiplications effectuées. On se départit légèrement du critère standard : le nombre d'appels récursifs.

```
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);;
```

On a les équations de complexité suivantes :

$$\begin{array}{ll} C(0) & = 0 \\ C(2n) & = 1 + C(n) \\ C(2n+1) & = 2 + C(n) \end{array}$$

Sachant que :

$$2^{\lceil \log_2 n \rceil - 1} < n \leq 2^{\lceil \log_2 n \rceil}$$

On a l'encadrement suivant, sous réserve de démontrer que $C(n)$ est bien monotone croissante :

$$C(2^{\lceil \log_2 n \rceil - 1}) < C(n) \leq C(2^{\lceil \log_2 n \rceil})$$

La preuve de monotonie peut être admise, elle s'établit par récurrence généralisée sur n pour la propriété suivante qui traite en même temps les cas n pair et impair :

$$C(2n) \leq C(2n+1) \leq C(2n+2)$$

— Pour $n = 0$, trivial.
— Pour $n > 0$, la propriété se décompose en : $1+C(n) \leq 2+C(n) \leq 2+C(n+1)$ pour laquelle l'hypothèse de récurrence permet de conclure.

On obtient alors de nouvelles équations (plus simples) de complexité en fonction de $C(2^n)$:

$$\begin{array}{ll} C(2^0) & = C(1) = 2 + C(0) = 2 \\ C(2^{n+1}) & = 1 + C(2^n) \end{array}$$

On obtient donc $C(2^n) = n + 2$, et en revenant à l'encadrement initial on a :

$$\lceil log_2 n \rceil + 2 - 1 < C(n) \leq \lceil \log_2 n \rceil + 2$$

Les deux bornes étant équivalentes à $\log_2 n$, on a $C(n) \sim \log_2 n$ (et donc également $C(n) = \Theta(\log_2 n)$).

4.2 Résolution des équations

Pour résoudre et intégrer une équation de complexité, la méthode générale consiste à 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

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

Comme ici, $C(i)$ est un nombre, on peut également lui associer la série exponentielle

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

ou autre, suivant la simplification éventuelle des calculs. La relation de récurrence des $C(n)$ permet de définir une équation sur $f(z)$, quelquefois une équation différentielle. Si on peut intégrer cette équation, alors on aura une fonction analytique de la complexité.

Notations de Landau On utilise les notations de Landau suivantes :

— $O(g)$ 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 $g(n)$.

$$f = \Theta(g) \triangleq f = O(g) \wedge g = O(f)$$

On a $\Theta(3n^2) = \Theta(10n^2 - 4n) = \Theta(n^2)$. 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, $2^n, \ldots$

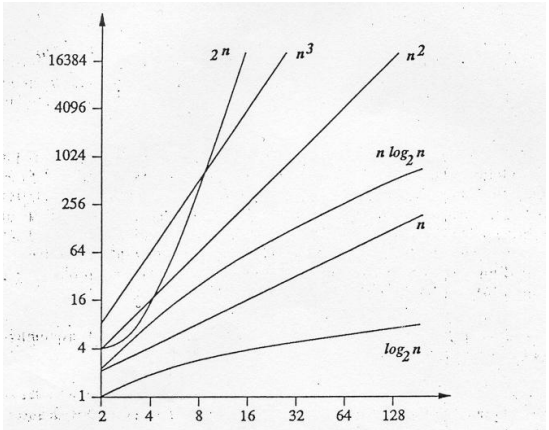


Figure 1. Rapidités de croissance comparées de certaines fonctions usuelles.

5 Récursivité terminale

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

— *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. L'évaluation est plus simple et peut être optimisée. Une fonction récursive terminale est directement équivalente à une **boucle** en programmation impérative. Voici l'équivalent du pgcd dans le langage impératif ADA :

```
function pgcd(x, y : in natural) return natural is
 xtmp     : natural ;
 ytmp     : natural ;
 resultat : natural ;
 begin
  -- xtmp et ytmp simulent les paramètres
  -- de la fonction récursive
  xtmp := x ;
  ytmp := y ;
  -- détection des cas terminaux
  -- condition de sortie de boucle
  while (xtmp <> 0 and ytmp <> 0) loop
  -- traitement des cas récursifs
  -- par affectations des 'paramètres' 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 ;
```

— *fact* n'est pas récursive terminale.

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

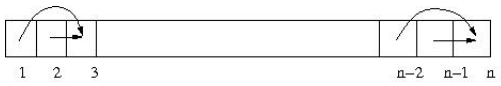
La conséquence de la non-terminalité est que :

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

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

Comment rendre une fonction terminale ? On peut utiliser la technique de la **marelle** (voir figure ci-dessous), qui consiste, lorsque la décomposition récursive ne dépend que d'un nombre constant d'appels récursifs, à introduire des paramètres supplémentaires, appelés **accumulateurs**, représentant les résultats de chacun de ces appels. Il faut évidemment adapter la fonction en conséquence : le cas de base est maintenant pour la valeur d'appel et le cas général doit être modifié pour introduire l'accumulateur.

```
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
```



Une fonction récursive terminale, étant équivalente à une boucle en programmation impérative, peut s'exécuter plus efficacement car on n'a pas besoin d'utiliser la pile d'appels.

6 Exercices

6.1 Définition d'une fonction récursive

▷ **Exercice 1**

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

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

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

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

— *Pour la puissance, proposer d'autres schémas, et d'autres fonctions, permettant de réduire le nombre d'opérations effectuées.*

▷ **Exercice 2 (P.G.C.D.)** *Voici une spécification du PGCD de deux nombres entiers naturels.*

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

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

▷ **Exercice 3 (Zéro d'une fonction)** *On cherche à déterminer, à ϵ près, le zéro d'une fonction continue sur un intervalle $[a, b]$.*
— *Proposer un contrat et une décomposition récursive par dichotomie pour une telle fonction.*
— *Établir le contrat et les tests de la fonction correspondante.*
— *Écrire la fonction correspondante. Combien d'appels réalise la fonction ?*

6.2 Calcul de complexité : exemple de la fonction puissance

On va comparer différentes versions de la fonction puissance. Pour rappel, le critère de taille retenu est le nombre de multiplications effectuées.

▷ **Exercice 4 (puissance naïve)**
— *Poser les équations de complexité (en nombre de multiplications) de la fonction ci-dessous.*
— *Quelle est, à votre avis, la complexité de cette fonction ?*

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

▷ **Exercice 5 (puissance "indienne" avec duplication des calculs)**
— *Poser les équations de complexité (en nombre de multiplications) de la fonction ci-dessous.*
— *Quelle est, à votre avis, la complexité de cette fonction ?*

```
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)
```

6.3 Récursivité terminale

▷ **Exercice 6**
— *Sur le même principe, donner une version récursive terminale de la fonction* `fibonacci` *:*

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