Programmation fonctionnelle

Dr. Mounir El Araki Tantaoui

Avec l'aimable autorisation du Professeur Jean Paul Roy
http://deptinfo.unice.fr/~roy/

Agenda

- Langage d’expressions préfixées
- Les fonctions
- Programmer avec des images / Animations
- **Programmer par récurrence**
- Les listes (chainées)
- Les calculs itératifs
- Type abstraits et généralisation
- Les arbres binaires

Qu'est-ce que la récurrence ?

1. Une méthode **majeure** de démonstration de théorèmes en maths !

- N.B. Le cœur de la stratégie réside dans l'obtention d'une **relation de récurrence** reliant $S(n)$ et $S(n-1)$, ici : $S(n) = S(n-1) + n$
- Soit $n \geq 1$ et $S(n) = 1 + 2 + ... + n$. **Prouver** que $S(n) = n (n + 1)$

PREUVE. Par récurrence sur $n \geq 1$. En deux temps :

a) si $n = 1$, c'est évident : $1 = 1 (1 + 1) /2$

b) montrons que si la propriété est vraie pour $n-1$, alors elle reste vraie pour n. Or :

- $S(n) = S(n-1) + n$

  $= (n-1)n/2 + n$ par **hypothèse de récurrence**

  $= n[(n-1)/2 + 1] = n(n+1)/2$

  d'où la récurrence ! CQFD

- N.B. Le coeur de la stratégie réside dans l'obtention d'une **relation de récurrence** reliant $S(n)$ et $S(n-1)$, ici : $S(n) = S(n-1) + n$

- **2. Une méthode majeure de construction d'objets mathématiques !**

- Un entier naturel est 0 ou bien le successeur d'un entier naturel. On génère ainsi les entiers sous la forme 0, S(0), S(S(0)), etc.

- La dérivée nème d'une fonction est la dérivée de la dérivée (n-1)ème. On calcule ainsi de proche en proche f, f', f'', f(3), etc.

- Un quadruplet s'obtient en prenant un couple (u,x) où u est un triplet.

- etc. Et oui, ce ne sont *que des maths* !
- Mais **comme les maths sont le domaine premier de la rigueur, ne nous en éloignons que si nous avons de très bonnes raisons de le faire !**

- **3. Une méthode majeure de raisonnement en programmation !**

- Soit $n \geq 1$ et $S(n) = 1 + 2 + ... + n$. **Programmer** la fonction S.

PROGRAMMATION. Par récurrence sur $n \geq 1$. En deux temps :

- a) si $n = 1$, c'est évident : $S(1) = 1$
- b) montrons que si je sais calculer $S(n-1)$, alors je sais calculer $S(n)$.

Ceci découle de la relation de récurrence $S(n) = S(n-1) + n$ d'où la programmation par récurrence ! CQFP

```
(define (S n)      ; n entier ≥ 1
  (if (= n 1)
      1
      (+ (S (- n 1)) n)))
```

```
> S(10)
55
> S(0)
```
dans les choux !

Bien entendu il était ici possible de *résoudre la récurrence* en faisant des maths, pour obtenir $S(n) = n(n+1)/2$.

Mais ceci est trop difficile dans le cas général en programmation.

- Une fonction est **récursive** si elle est programmée par récurrence.

```
(define (fac n)     ; n entier ≥ 0
  (if (= n 0)
      1
      (* (fac (- n 1)) n)))
```

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \quad \text{si } n > 0 \end{aligned}$$

- Autrement dit, elle s'utilise elle-même dans sa définition !
- Mécanique à deux temps :
  - si n=0, je sais calculer n! puisque 0! = 1
  - si n>0, je suppose que je sais calculer (n-1)! et j'en déduis le calcul de n!

- Il est vital de prévoir un **cas de base** [souvent n=0] sur lequel le cas général doit finir par converger. Erreur typique :

```
(define (fac n)
  (* n (fac (- n 1))))
```

$n \to -\infty$

- Somme des carrés des entiers de [0,100] $\qquad \sum_{k=0}^{n} k^2$

- On **généralise** le problème : somme des carrés des entiers de [0,n] avec n ≥ 0.

$$\boxed{0^2 + 1^2 + 2^2 + \cdots + n^2} = \underbrace{\boxed{0^2 + 1^2 + 2^2 + \cdots + (n-1)^2}}_{\text{HR}} + n^2$$

```
(define (somme-carres n)      ; n entier ≥ 0
  (if (= n 0)
      0
      (+ (somme-carres (- n 1)) (sqr n))))
```

```
> (somme-carres 100)
338350

> (somme-carres -100)
User break
```

- Dans la fonction somme-carrés, le cas de base **n = 0** n'est pas fameux car il restreint le domaine de définition de la fonction à ***N***. En fait, il ne respecte pas la spécification mathématique de la fonction :

$$S(n) = \sum_{k=0}^{n} k^2 = \sum_{0 \leq k \leq n} k^2$$

- Or pour n < 0, l'inéquation 0 ≤ k ≤ n n'a pas de solution, donc :

$$S(n) = \sum_{k \in \emptyset} k^2 = 0 \quad \text{si } n < 0$$

```
(define (somme-carres n)        ; n entier quelconque
  (if (<= n 0)
      0
      (+ (somme-carres (- n 1)) (sqr n))))
```

```
> (somme-carres 100)
338350
```

```
> (somme-carres -100)
0
```

- La **fonction puissance** (expt x n) avec n entier ≥ 0 VERSION 1
- La fonction primitive (expt a b) calcule a^b.
- Comment programmerions-nous (expt x n) avec n entier ≥ 0 si elle n'existait pas ? Nommons-la $expt *pour ne pas tuer la primitive* !

$$x^0 = 1$$
$$x^n = x \times x^{n-1} \quad \text{si } n > 0$$

```
(define ($expt x n)                ; x complexe, n entier ≥ 0
  (if (= n 0)
      1
      (* x ($expt x (- n 1)))))
```

```
> ($expt 2 10)
1024
```

- **Complexité** : le nombre de multiplications est d'ordre n : **en O(n)**.

- La **fonction puissance** (expt x n) avec n entier ≥ 0 VERSION 2

- Pour **faire baisser la complexité**, on essaye une **DICHOTOMIE** sur n.

$$
\begin{array}{lcll}
x^0 & = & 1 & \\
x^n & = & (x^2)^{n/2} & \textit{si } n \textit{ est pair} \\
x^n & = & x \times (x^2)^{(n-1)/2} & \textit{si } n \textit{ est impair}
\end{array}
$$

DICHOTOMIE : action de couper en deux !

```
(define ($expt x n)      ; x complexe, n entier ≥ 0
  (cond ((= n 0) 1)
        ((even? n) (sqr ($expt x (quotient n 2))))
        (else (* x (sqr ($expt x (quotient n 2)))))))
```

- **Complexité** : le nombre de multiplications est d'ordre le nombre de fois que l'on peut diviser n par 2 avant de tomber sur 0.
- C'est donc le **logarithme** en base 2 de n. **Complexité O(log n)**.

Les ordres de grandeur en complexité

- Lorsqu'on **analyse la complexité d'un algorithme** :
  - On choisit une **unité de mesure**. Par exemple le nombre d'opérations, ou le temps de calcul, ou l'espace mémoire consommé...
  - On se place toujours dans **le pire des cas**. Tel algorithme sera rapide pour certaines données, et lent pour d'autres.
- Si l'on s'intéresse au **temps de calcul**, on utilise la primitive time :

```
> (time (sqrt 67567563257846384730958398509836578365894534534509823))
cpu time: 0 real time: 0 gc time: 0
2.5993761416510383e+26

> (time (* 0 (fac 5000)))
cpu time: 154 real time: 156 gc time: 112
0
```

```
(define (fac n)          ; n ≥ 0
  (if (= n 0)
      1
      (* n (fac (- n 1)))))
```

154 - 112 = 42 ms

gc *= temps passé à nettoyer la mémoire [Garbage Collector]*

- ATTENTION : le nombre d'opérations n'est pas nécessairement proportionnel au temps de calcul ! En effet, multiplier des nombres à 3 chiffres ou à 300 chiffres ?... Exemple avec la factorielle :

```
> (time (* 0 (fac 5000)))
cpu time: 154 real time: 155 gc time: 111
0
> (time (* 0 (fac 10000)))
cpu time: 528 real time: 532 gc time: 355
0
```

$N = 5000$	$Temps = 43$
$N = 10000$	$Temps = 173$

- La complexité du calcul de (fac n) si l'on mesure **le nombre de multiplications**, est **linéaire** : d'ordre n.
- La complexité du calcul de (fac n) si l'on mesure **le temps de calcul n'est pas linéaire**. Sinon, le temps de calcul pour N = 10000 aurait été le double de celui pour N = 5000. Or c'est 4 fois plus !

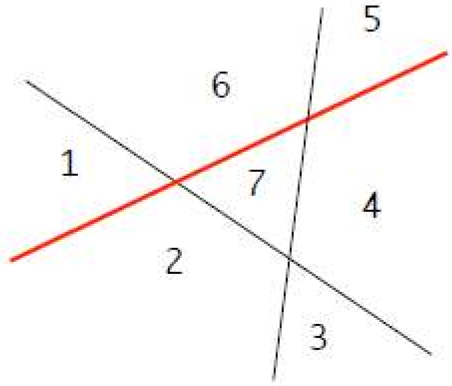
Cela semble indiquer un comportement peut-être ***quadratique*** *en temps de calcul : en* ***O(n²)*** *?...*

- Les grandes **CLASSES DE COMPLEXITE** [coût] d'algorithmes :

- Les algorithmes de **coût constant O(1)**. Leur complexité ne dépend pas de la taille n des données.
- Les algorithmes de **coût logarithmique O(log n).** Leur complexité est *dans le pire des cas* de l'ordre de log(n).
- Les algorithmes de **coût linéaire O(n).** Leur complexité est *dans le pire des cas* de l'ordre de n.
- Les algorithmes de **coût quasi-linéaire O(n log n).** Presqu'aussi bons que les algorithmes linéaires...
- Les algorithmes de **coût quadratique O(n^2).** Pas fameux...
- Les algorithmes de **coût exponentiel O(2^n).** Catastrophique !!

• Résolution du problème combinatoire nb-régions

• Etant données n droites du plan en position générale, calculer le nombre R_n de régions (bornées ou pas).



a) Si n = 0, il est clair que $R_0 = 1$

b) Supposons connu R_{n-1} et introduisons une n$^{\text{ème}}$ droite dans une configuration de n-1 droites.

La n$^{\text{ème}}$ droite va rajouter une région pour chaque droite de la configuration, plus une dernière pour la région non bornée, ok ?

$$R_n = R_{n-1} + (n-1) + 1 = R_{n-1} + n$$

En résumé :

```
(define (nb-regions n)
  (if (= n 0)
      1
      (+ (nb-regions (- n 1)) n)))
```

```
> (nb-regions 3)
7
> (nb-regions 10)
56
```

• La **complexité** est **O(n)** en le nombre d'opérations.

- **Peut-on résoudre une récurrence ?**
- Etant donnée une formule de récurrence, peut-on éliminer la récurrence ? Trouver une **formule directe** pour le terme général ? Peut-on **résoudre une récurrence** ?
- **REPONSE** : C'est difficile voire impossible en général ! On le peut dans certains cas très simples, par exemple pour nb-regions.

On déroule la récurrence :

$$\begin{cases} R_0 = 1 \\ R_n = R_{n-1} + n \end{cases}$$

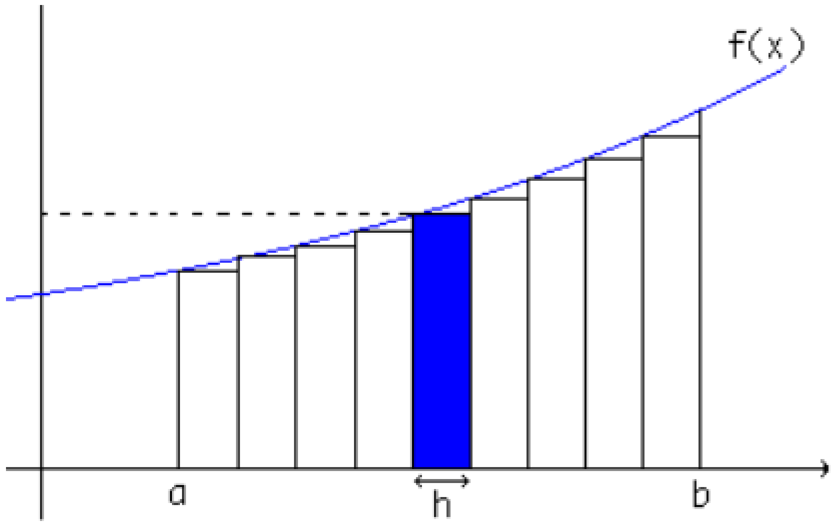
$$\begin{aligned}
R_n &= R_{n-1} + n \\
R_n &= R_{n-2} + (n-1) + n \\
R_n &= R_{n-3} + (n-2) + (n-1) + n \\
&\dots \\
R_n &= R_0 + 1 + 2 + \cdots + n \\
R_n &= 1 + (1 + 2 + \cdots + n) \\
R_n &= 1 + \frac{n(n+1)}{2} \\
R_n &= \frac{n^2 + n + 2}{2}
\end{aligned}$$

formule que l'on peut ensuite... prouver par récurrence !

```
(define (nb-regions-direct n)
  (quotient (+ (* n (+ n 1)) 2) 2))
```

Complexité : **O(1)** opérations !

- **Calcul approché d'une intégrale**
- Intégrale approchée d'une fonction continue f sur [a,b], avec la méthode de Riemann : découpage de [a,b] en rectangles de largeur h, où h est petit, par exemple 0.01.



- Le cas de base a lieu lorsque a > b.
- La *relation de Chasles* permet un calcul récursif :

$$\int_a^b f(x)dx = \int_a^{a+h} f(x)dx + \int_{a+h}^b f(x)dx$$

$$\int_a^b f(x)dx \approx h \times f(a) + \int_{a+h}^b f(x)dx$$

- Une récurrence double : **la suite de Fibonacci 0, 1, 1, 2, 3, 5, 8, 13...**

$$f_0 = 0, \quad f_1 = 1, \quad f_n = f_{n-1} + f_{n-2} \quad si\ n \geq 2$$

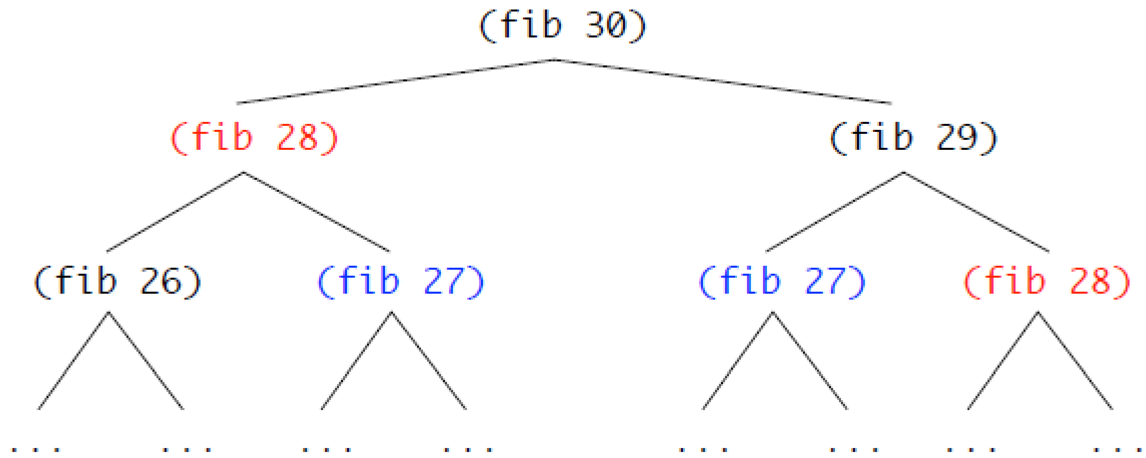
```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

Les deux appels récursifs

```
> (time (fib 15))
cpu time: 1 real time: 1 gc time: 0
610
> (time (fib 30))
cpu time: 920 real time: 941 gc time: 0
832040
```

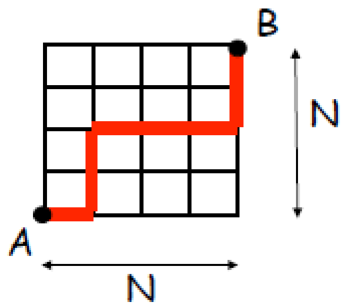
Oups !...

- *Oups ?...* Regardons **l'arbre du calcul** :



- L'algorithme passe son temps à **faire et refaire les mêmes calculs** !!
- Le nombre de calculs est EXPONENTIEL puisque le nombre de nœuds dans l'arbre est de l'ordre de 230. Très mauvais...
- Nous aurons l'occasion de voir une autre solution plus rapide..

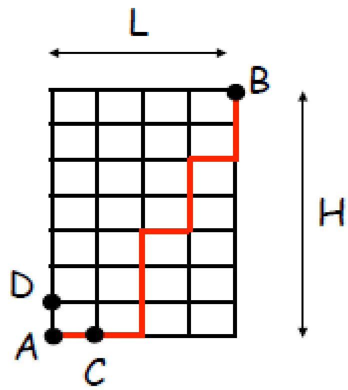
- Résolution du problème combinatoire **nb-chemins**
- Parfois, ne pas hésiter à GENERALISER LE PROBLEME !
- Exemple : un robot parcourt un monde carré de côté N. Il doit se rendre du point A au point B en suivant un chemin de longueur minimale 2N.
  - Combien de chemins possibles ?



1er ESSAI : Même si je suppose que je sais résoudre le problème dans un monde carré plus petit de côté N-1, je n'arrive pas à en déduire la solution pour un monde carré de côté N. La récurrence brutale résiste !

```
(define (nb-chemins N)
  (if (= N 0)
      1
      (local [(define HR (nb-chemins (- N 1)))]
         ???)))
```

- Bloqué ! J'essaye de GENERALISER le problème.



- 2ème ESSAI : Je vais donc **relaxer les données** et supposer que je travaille dans un **rectangle** de largeur L et de hauteur H.
- Pour aller de A vers B, je dois passer par C ou par D... et je me retrouve encore chaque fois dans un problème rectangulaire ! D'où la récurrence :

```
(define (nb-chemins-rect L H)             ; monde rectangulaire
  (if (or (= L 0) (= H 0))
      1
      (+ (nb-chemins-rect (- L 1) H)       ; en passant par C
         (nb-chemins-rect L (- H 1)))))    ; en passant par D


(define (nb-chemins N)               ; le carré comme cas particulier
  (nb-chemins-rect N N))
```

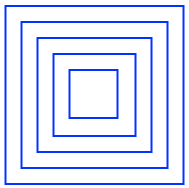
N.B.** On peut ensuite **localiser nb-chemins-rect *à l'intérieur de* nb-chemins*.*

- Construction d'images par récurrence !

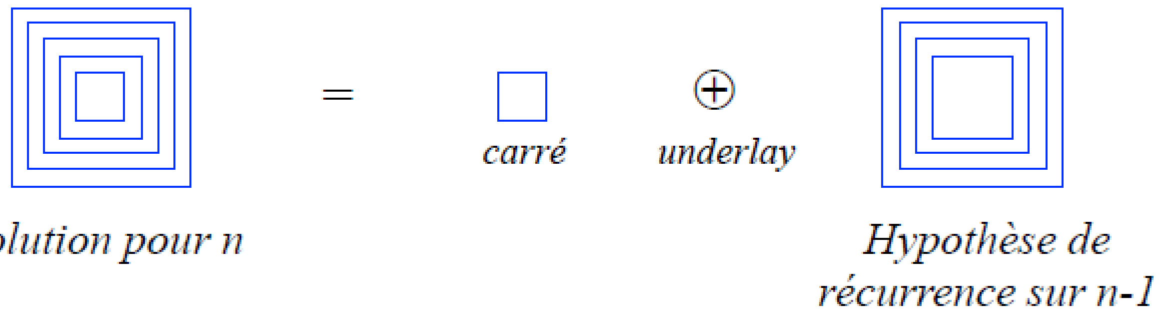
  - Une image de n carrés emboîtés :

```
(define (carrés-emboîtés n size incr)        ; récurrence sur n ≥ 1
  (if (= n 1)
      (carré size)
      (underlay (carré size)
                (carrés-emboîtés (- n 1) (+ size incr) incr))))
```

```
> (carrés-emboîtés 5 30 20)
```



N.B.



Méthodologie

- Lorsque vous êtes coincés [dans la vie, en maths, en programmation], essayez de PARLER, de VERBALISER...
- Lors de la rédaction d'une fonction récursive f(n), n'hésitez pas à **DONNER UN NOM** [par exemple HR] **A L'HYPOTHESE DE RECURRENCE**

```
(define (fac n)    ; n entier ≥ 0
  (if (= n 0)
      1
      (local [(define HR (fac (- n 1)))]  ; supposons que...
        (* HR n))))
```

- Ceci est particulièrement vital si HR est utilisé plusieurs fois dans la relation de récurrence (on ne calcule JAMAIS plusieurs fois HR !)...

$$u_n = u_{n-1} + \sqrt{u_{n-1}} \implies \textit{Calcul naïf en } O(2^n)$$