Récursion

Fonctions récursives

Les structures de données récursives : traitées au semestre suivant

Plan sur 2 séances

Récursion :

définitions et premiers exemples

Fonctions récursives : cinq exemples, appels, intérêts et limitations

factorielle, Fibonaci, exponentiation rapide, recherche dichotomique, tri fusion

Exercices dirigés :

arbre/pile des appels, environnements, itératif → récursif, tours de Hanoi

Terminaison, correction, complexité

Compléments :

formes de récursion, dérécursivation de forme terminale

Récursivité

Définition, premiers exemples

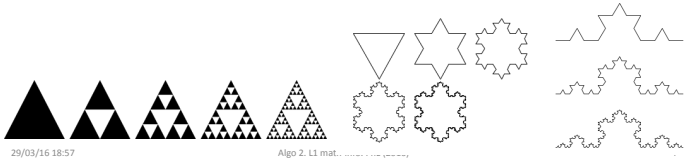
Récursion

Définition

Une construction est récursive si elle se définit à partir d'elle-même.

Exemples basiques

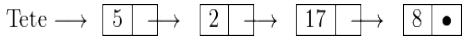
- la fonction factorielle de n : n! = 1 x 2 x 3 x ... x (n-1) x n et **n! = n x (n-1)!**
- la fonction somme des n premiers entiers : **s(n) = n + s(n-1)**
- La suite de Fibonacci : **f**(n+1) = **f**(n)+**f**(n-1)
- le triangle de Sierpinsky, le flocon de Von Koch :



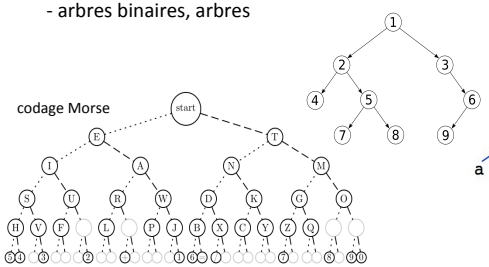
Structures de données récursives

Structures de données non linéaires

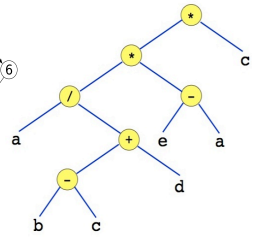
- liste chaînée



- arbres binaires, arbres



[a / ((b-c)+d) * (e-a)] * c



Récursion

Récursion et récurrence

La correction d'un traitement récursif est souvent fourni par une démonstration par récurrence

Récursion et terminaison

Les appels récursifs sont "de plus en plus petits" jusqu'à un ensemble de cas terminaux qui couvrent toutes le valeurs non traitées dans les appels récursifs.

Des récursions importantes

- **diviser pour régner**

  on divise le problème en des problèmes similaires de taille moindre, et ce récursivement

- la recherche dichotomique

  on cherche la solution sur un ensemble de taille réduite par 2, et ce récursivement

- exponentiation rapide

  $x^{2p} = (x^p)^2$ et $x^{2p+1} = x \times (x^p)^2$ donnent une solution récursive plus efficace que la solution itérative

Récursion

Avantages

Une solution algorithmique récursive est souvent plus simple, plus lisible, plus facile à prouver qu'une solution itérative

- Exemple : résoudre le problème des tours de Hanoi

« Tower of Hanoi 4 » par André Karwath aka Aka — Travail personnel. Sous licence CC BY-SA 2.5 via Wikimedia Commons. http://commons.wikimedia.org/wiki/File:Tower_of_Hanoi_4.jpg#mediaviewer/File:Tower_of_Hanoi_4.gif

Inconvénients

L'exécution de la solution récursive est plus compliquée : gestion d'une pile d'appels de fonctions (la pile d'exécution)

L'exécution de la solution récursive peut provoquer des débordements de la mémoire

Fonctions récursives

1- cinq exemples, appels, intérêts et limitations

factorielle, Fibonaci, exponentiation rapide, recherche dichotomique, tri fusion

Factorielle

... trop classique mais instructif

Factorielle : version itérative (rappel)

```
fonction factorielle(n : entier) retourne entier
  res : entier = 1     // j'accumule dans res
  i : entier           // iterateur
debut
  pour i de 1 à n faire
     res = res * i
  fin pour
  retourne res
fin fonction
```


Factorielle : version récursive *partielle*

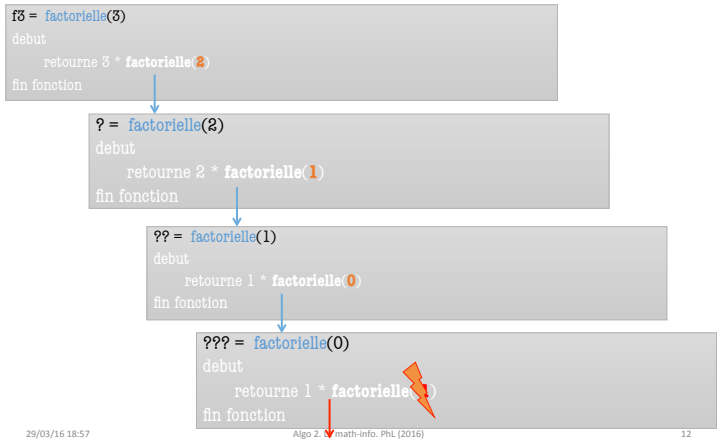
Définition de fonction:

```
fonction factorielle(n : entier) retourne entier
//Première étape : la relation  de récurrence pour n ≥ 1
debut
    retourne n * factorielle(n-1)
fin fonction
```

Appel de fonction classique :

```
declare
    f3 : entier
début
    f3 =  factorielle(3)
fin
```


Factorielle récursive : arrêter les appels !



Factorielle : version récursive qui termine

```
fonction factorielle(n : entier) retourne entier
//  1!=1 et n! = n*(n-1)! n > 1
debut
    si n==1 alors
        retourne 1
    sinon
        retourne n * factorielle(n-1)
    fin si
fin fonction
```

2 instructions de retour
- la première retourne une valeur terminale
- la seconde provoque un appel récursif (à elle-même) **avant** d'effectuer un quelconque **calcul** !

Factorielle : autre version récursive qui termine

```
fonction factorielle(n : entier) retourne entier
//  1!=1 et n! = n*(n-1)! n > 1
    res : entier
debut
    si n==1 alors
        res = 1
    sinon
        res = n * factorielle(n-1)
    fin si
    retourne res
fin fonction
```

Version équivalente à la précédente
- 1 variable locale res
- 1 seule instruction de retour

Factorielle : version récursive détaillée

Objectif : bien identifier les variables intermédiaires et qui seront "re-créées" à chaque appel récursif. Version équivalente à la précédente

```
fonction factorielle(n : entier) retourne
entier
//  1!=1 et n! = n*(n-1)! n > 1
    r, f  : entier
debut
    si n==1 alors
        retourne 1
    sinon
        r = factorielle(n-1)
        f = n * r
        retourne f
    fin si
fin fonction
```

Chaque appel récursif à factorielle () introduit 2 variables locales : r et f.

Ces variables doivent être évaluées dans le contexte (au niveau) de l'appel concerné :

pour n, f nécessite que r soit évaluée, elle-même nécessitant que **f** soit évaluée pour n-1, et ce récursivement ...

Fonction récursive : première synthèse

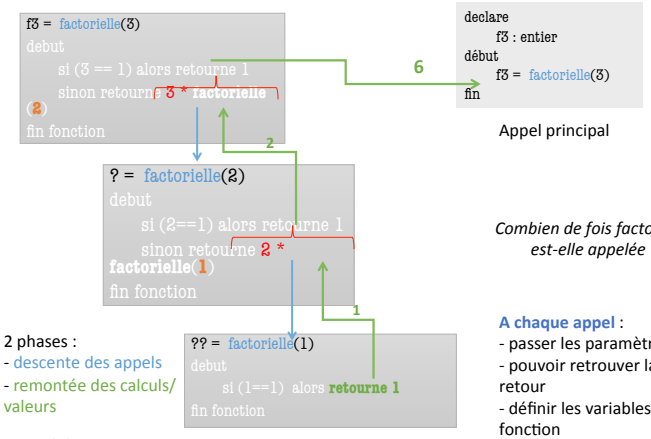
Une fonction récursive (simple) :
- s'appelle elle-même sur de nouvelles données
- inclut un test d'arrêt qui interrompt les appels récursifs : cas terminaux

  Il est conseillé de traiter ces cas terminaux en début de programme, cad. avant le premier appel récursif

Une fonction récursive s'appelle comme une fonction classique. Elles fournit un résultat (affectation dans une variable) ou non (effet de bord possible)

D'autres formes de fonctions récursives sont présentées dans les compléments en fin de ce chapitre :
- récursion forte
- fonctions mutuellement récursives

Factorielle récursive : appels, retours et calculs



2 phases :
- descente des appels
- remontée des calculs/ valeurs

A chaque appel :
- passer les paramètres d'appels
- pouvoir retrouver la variable de retour
- définir les variables locales de la fonction

"Arbre" des appels récursifs

```
Appel à fact(4)
.   4*fact(3) = ?
.   Appel à fact(3)
.   .   3*fact(2) = ?
.   .   Appel à fact(2)
.   .   .   2*fact(1) = ?
.   .   .   Appel à fact(1)
.   .   .   .   1*fact(0) = ?
.   .   .   .   Appel à fact(0)
.   .   .   .   Retour de la valeur 1
.   .   .   .   1*1
.   .   .   Retour de la valeur 1
.   .   .   2*1
.   .   Retour de la valeur 2
.   .   3*2
.   Retour de la valeur 6
.   4*6
Retour de la valeur 24
```

Ici fact(n) est écrit avec la condition de terminaison :
si n == 0 alors retourne 1

Les 2 phases :
- descente = les appels
- remontée = retour des valeurs et calculs intermédiaires

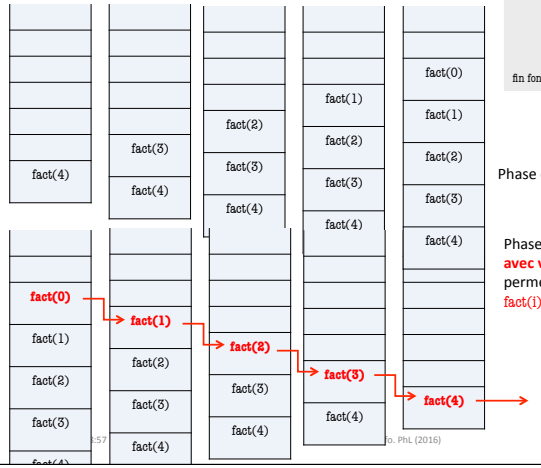
Bien identifier les appels récursifs imbriqués par le niveau d'indentation

Appels récursifs

Chaque appel récursif nécessite de construire dynamiquement un nouvel environnement : un nouvel appel de fonction
- adresse(s) de retour
- paramètres d'appel
- variables locales

La pile d'exécution : la mémoire associée nécessaire au stockage des paramètres d'appel, des adresses de retour, des variables locales
- gérée par le langage de programmation : une pile LIFO de taille préfixée
- mais risque de débordement mémoire si appels imbriqués (sans libération de la place occupée) trop nombreux :
  en python : 1000 appels au maximum
  ou trop volumineux :
  imaginons un paramètre tableau de grande taille réduit de 1 élément par 1 élément ... exemple : sommer récursivement les n valeurs d'un tableau
  qui plus est si la récursion est multiple :
  Fibonacci : f(n+1) = f(n) + f(n-1)

Pile des appels récursifs




```
fonction factorielle(n : entier) retourne entier
// i!=1 et n! = n*(n-1)! n > 1
debut
    si n==1 alors
        retourne 1
    sinon
        retourne n * factorielle(n-1)
    fin si
fin fonction
```

Phase des appels récursifs

Phase des évaluations récursives **avec valeur de retour** qui permet d'évaluer :
fact(i) = i*fact(i-1), i≥1

Fibonacci

... qui croît très vite !

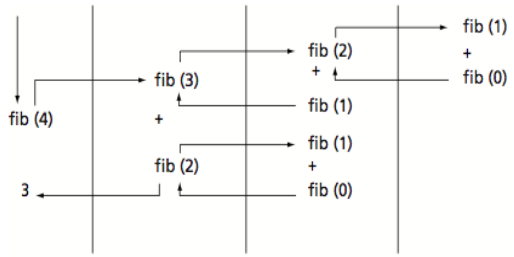
mais pourtant moins vite que son temps de calcul par l'algorithme récursif naturel : attention !

Fibonnaci: une récursion multiple

```
fonction fib(n : entier) retourne entier
// f(0)=f(1)=1, f(n+1) = f(n) + f(n-1), n > 1
      res : entier
debut
      si (n == 0) ou (n==1) alors
            res = 1
      sinon
            res = fib(n) + fib(n-1)
      fin si
      retourne res
fin fonction
```

Peut aussi s'écrire avec 2 retourne et 0 variable locale

Fibonacci ou l'inefficacité de la récursion



Que d'appels répétés !
- fib(2) est évalué 2 fois
- fib(1) est évalué 3 fois
- fib(0) est évalué 2 fois

Plus efficace : version itérative (exercice) ou stockage des valeurs déjà calculées

Fibonacci ou l'inefficacité de la récursion

Exercice présenté en dernière section :
construire l'arbre puis la pile des appels récursifs de Fib(4)

Exponentiation et exponentiation rapide

... ou le même résultat mais en temps linéaire ou logarithmique

L'exponentiation entière : version récursive classique

Objectif : calculer x^n
Principe : utiliser $x^n = x \times x^{n-1}$

```
fonction expo(x : flottant, n : entier) retourne flottant
    res : flottant
debut
    si (n == 0) alors
        res = 1.0
    sinon
        res = res * expo(x, n-1)
        fin si
    fin si
    retourne res
fin fonction
```


L'exponentiation rapide ou l'efficacité de la récursion

Objectif : calculer x^n
Principe : utiliser $x^{2p} = (x^p)^2$ et $x^{2p+1} = x \times (x^p)^2$

```
fonction expo_rapide(x : flottant, n : entier) retourne flottant
    res : flottant
debut
    si (n == 0) alors
        res = 1.0
    sinon
        res = expo_rapide(x, n/2)
        si (n%2 ==0) alors    // n est pair
            res = res * res
        sinon                 // n est impair
            res = res * res * x
        fin si
    fin si
    retourne res
fin fonction
```


Exponentiation rapide ... en effet !

Calculons 3.0^5

- Exponentiation récursive classique : x * expo(x, n-1)
  res = 3.0 x expo(3.0, 4) = 3.0 x 3.0 x expo(3.0, 3) = 3.0 x 3.0 x 3.0 x expo(3.0, 2) = 3.0 x 3.0 x 3.0 x 3.0 x expo(3.0, 1) = 3.0 x 3.0 x 3.0 x 3.0 x3.0

- Exponentiation récursive rapide
  res = expo_rapide(3.0, 5) :
  appelle expo_rapide(3.0, 2)
  sans appeler
  expo_rapide(3.0, 4)
  expo_rapide(3.0, 3)

```
fonction expo_rapide(x : flottant, n : entier) retourne
flottant
    res : flottant
debut
    si (n == 0) alors
        retourne 1.0
    sinon
        res = expo_rapide(x, n/2)
        si (n%2 ==0) alors
            retourne res * res
        sinon
            retourne res * res * x
        fin si
    fin si
fin fonction
```

Recherche dichotomique

... qui permet de réduire très vite l'espace de recherche

Recherche dichotomique : rappel du principe

Un algorithme "diviser pour régner" où chaque division réduit la recherche à un ensemble de taille moitié, l'autre ensemble n'étant plus considéré

- diviser :
  - . on partage en 2 par la moitié le tableau trié
- régner :
  - . on compare la valeur cherchée à la valeur médiane du tableau
  - . si besoin, on en déduit la moitié gauche ou droite du tableau qui contient la valeur cherchée
  - . on recommence la recherche sur la "bonne moitié"

Terminaison

la taille de l'espace de recherche est diminuée par 2 à chaque itération

- exemple : n=64→32 → 16 → 8 → 4 → 2 → 1
- si n>0 est la taille de l'espace de recherche (nombre de valeurs, longueur du tableau), cette taille évolue comme la suite n, n/2, n/4, ..., $n/2^k$, ..., soit comme une suite géométrique de raison ½ < 1, et de premier terme non nul. Cette suite géométrique converge vers 1.
- Ainsi la dichotomie construit, à terme, un ensemble de 1 élément, un singleton, égal ou non à la valeur cherchée (résultat de la comparaison : étape 1 de la phase "régner").
- Donc la dichotomie termine.

Il faut maintenant assurer que la recherche est correcte.

Dichotomie : algorithme itératif

```
fonction dichotomie(v : flottant, t : tableau de flottants, n : entier) retourne booléen
// renvoie Vrai si la valeur v est présente dans le tableau t, et Faux sinon
    present : booleen = Faux
    g : entier = 0                    // indice du premier élément du tableau t (à gauche)
    d : entier = n-1                  // indice du dernier élément du tableau t (à droite)
début
    tantque (present == Faux) et (g <= d) faire // si g > d alors t[g,d] est vide
        m = (g+d) ÷ 2                           // indice du pivot ≥ 0
        si t[m] == v alors
            present = Vrai                      // on a trouvé : le pivot est la valeur cherchée
        sinon si v < t[m] alors
            d = m-1                             // on cherchera dans la partie gauche de t
        sinon
            g = m+1                             // on cherchera dans la partie droite de t
        fin si
    fin tantque
    retourne present                            // ou m si on cherche la position de v
fin fonction
```


La dichotomie : version récursive

Principe

rappeler récursivement la recherche sur le sous-tableau (droit ou gauche) adapté et ce tant que la valeur cherchée n'est pas trouvée ou trouvable (sous-tableau vide)

Exercice

Il suffit de pouvoir mettre à jour les indices g et d de la recherche dans le sous-tableau t[g..d] → introduction de paramètres g et d

En-tête

```
fonction dichotomieRec(v : flottant, t : tableau de flottants, g : entier, d : entier) retourne booléen
```

Appel : on cherche la valeur 15.0 dans le tableau notes[0..31]

```
declare
    trouvé : booleen
debut
    trouvé= dichotomieRec(15.0, notes, 0, 31)      // bien noter la forme
fin                                                // de l'appel principal
```

La dichotomie : version récursive

```
fonction dichotomieRec(v : flottant, t : tableau de flottants, g : entier, d : entier) retourne booléen
// renvoie Vrai si la valeur v est présente dans le tableau t [g..d], et Faux sinon
début
    si (g > d) alors                          // alors t[g,d] est vide
        retourne Faux                         // premier cas terminal
    fin si
    m = (g+d) ÷ 2                             // indice du pivot ≥ 0
    si t[m] == v alors                        // on a trouvé : le pivot est la valeur cherchée
        retourne Vrai                         // second cas terminal
    fin si
    si v < t[m] alors
        retourne dichotomieRec(v, t, g, m-1)  // on cherche dans le sous-tableau gauche de t
    sinon
        retourne dichotomieRec(v, t, m+1, d)  // on cherche dans la sous-tableau droit de t
    fin si
fin fonction
```


Tri fusion

... à venir dans un chapitre prochain

Fonctions récursives

2 - exercices dirigés

Exercices dirigés

Arbre, pile des appels et environnements successifs de l'exécution de factorielle(3) en version détaillée

Arbre, pile des appels et environnements successifs de l'exécution de fibonacci(3) en version détaillée

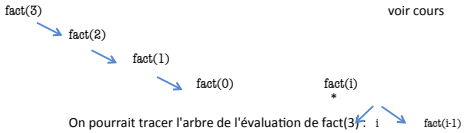
Arbre, pile des appels de l'exécution de fibonacci(4)

Formes récursives d'une boucle pour d'indices décroissants ou croissants

Tours de Hanoi
- solution récursive
- exécution : tracé des déplacements consécutifs
- complexité

Arbre, piles des appels et environnements successifs de l'exécution de **fact(3)** en version détaillée

Arbre des appels :



Pile des appels :

voir cours

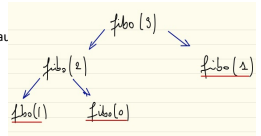
Environnements

	0	1	2	3	4
1	res=fact(4)	n=3			
2		y = fact(2)	n=2		
3			y=fact(1)	n=1	
4				y = fact(0)	n=0
5					retourne 1
6				retourne 1*1	
7			retourne 2*1		

Arbre, pile des appels et environnements successifs de l'exécution de **fibo(3)** en version détaillée

Arbre des appels

rouge : cas terminaux

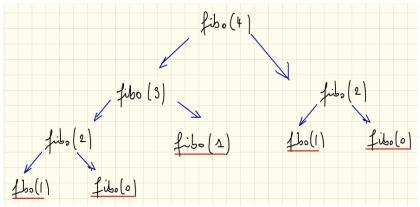


Environnements

couleurs : appels et environnement associés

	0	1	2	3
1	res=fibo(3)	n=3		
2		y = fibo(2)	n=2	
3			y=fibo(1)	n=1
4				retourne 1
5			y=fibo(0)	
6				n=0
7				retourne 1
8			retourne 1+1	
9		y=fibo(1)		
10			n=1	
11			retourne 1	

Arbre, pile des appels de l'exécution de **fibo(4)**



Piles des appels (en notant fibo(n) par f(n))

```
                f(1) f(0)
          f(2) f(2) f(2) f(2) f(1)      f(1) f(0)
     f(3) f(3) f(3) f(3) f(3) f(3) f(3)f(2) f(2) f(2) f(2)
f(4) f(4) f(4) f(4) f(4) f(4) f(4) f(4)f(4) f(4) f(4) f(4) f(4)
```


Formes récursives d'une boucle **pour** d'indices décroissants ou croissants

```
fonction boucle_vers_1(n : entier)
// boucle d'indices décroissants
debut
    si n>0 faire
        afficher(n)
        boucle_vers_1(n-1)
    fin si
fin fonction
```

```
fonction boucle_depuis_1(n : entier)
// boucle d'indices croissants
debut
    si n>0 faire
        boucle_depuis_1(n-1)
        afficher(n)
    fin si
fin fonction
```

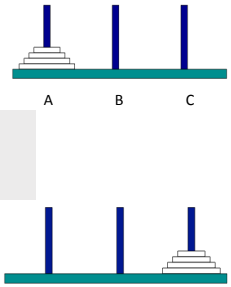
```
// l'appel affiche : 10 9 8 7 6 5 4 3 2 1
declare
    utilise fonction boucle_vers_1(n : entier)
debut
    boucle_vers_1(10)
fin
```

```
// l'appel affiche : 1 2 3 4 5 6 7 8
declare
    utilise fonction boucle_depuis_1(n : entier)
debut
    boucle_depuis_1(8)
fin
```

Tours de Hanoi : solution récursive

```
fonction hanoi(n : entier, depart : entier, arrivee : entier, interm : entier,)
//  n disques depart → arrivee
debut
        si n>0 faire
                hanoi( n-1, depart, interm , arrivee )
                afficher("Déplacer un disque de", depart, " -> ", arrivee
                hanoi( n-1, interm , arrivee , depart)
        fin si
fin fonction
```

```
// l'appel affiche : 10 9 8 7 6 5 4 3 2 1
declare
        A, B, C : entiers
        utilise fonction deplacer(n : entier, depart : entier, arrivee : entier, interm :
entier)
debut
        hanoi(4, A, C, B)
fin
```



Tours de Hanoi : exécution pour Hanoi(3, A, C, B)

Déplacements successifs

```
Déplacer un disque de A--> B
Déplacer un disque de A--> C
Déplacer un disque de B--> C
Déplacer un disque de A--> B
Déplacer un disque de C--> A
Déplacer un disque de C--> B
Déplacer un disque de A--> B
```

Nombre de disques

	Socle A		
Départ	3	2	1
A → B	3	2	
A → C	3		
B → C	3		
A → B			
C → A	1		
C → B	1		
A → B			

Socle B		
1		
1		
3		
3		
3	2	
3	2	1

Socle C		
2		
2	1	
2	1	
2		

Fonctions récursives

3- terminaison, correction, complexité

Correction et terminaison de factorielle(n)

H_n : factorielle(n) termine et renvoie la valeur n!

Par récurrence sur n≥0 :
- H_0 est vérifiée car factorielle(0) se réduit à retourne 1 = 0!.
- Pour n>0, supposons H_{n-1} et montrons H_n.

La calcul commence par l'appel à factorielle(n-1) dont on sait (H_{n-1}) qu'il termine et renvoie la valeur (n-1)!. Le calcul continue avec le produit de ce résultat par n et son renvoie. Donc l'appel à factorielle(n) termine et renvoie bien nx(n-1)!=n!. Ce qui démontre H_n.

Remarque :
- Rien n'est prouvé pour n<0. Dans ces cas, factorielle(n) peut ne pas terminer ou renvoyer des valeurs sans signification.

```
fonction factorielle(n : entier) retourne
entier
//  {l=1 et n! = n*(n-1)! n > 1
debut
        si n==0 alors
                retourne 1
        sinon
                retourne n * factorielle(n-1)
        fin si
fin fonction
```

Complexité de factorielle(n)

C(n) : nombre d'opérations arithmétiques dans factorielle(n)

C(0) = 0
C(1) = 1
...
C(n-1) = C(n-2) +1
C(n) = C(n-1) + 1
~~C(n) = 1 + 1 + ... + 1~~ (n fois) = n

Complexité en temps linéaire pour les formes itératives et récursives de factorielle(n)

```
fonction factorielle(n : entier) retourne entier
// 1! = 1 et n! = n*(n-1)! n > 1
debut
    si n==0 alors
        retourne 1
    sinon
        retourne n * factorielle(n-1)
    fin si
fin fonction
```


Complexité de l'exponentiation entière rapide

Vérifions sur le calcul de x^{100} que expo_rapide est beaucoup plus rapide que expo :

- 100 est pair. On obtient x^{100} en élevant x au carré, $x_1 = x^2$ (1ère mult.) ; on devra élever x_1 à l'exposant 50.
- 50 est pair. On obtient $(x_1)^{50}$ en élevant x_1 au carré, $x_2 = (x_1)^2$ (2ème mult.) ; on devra élever x_2 à l'exposant 25.
- 25 est impair. On obtient $(x_2)^{25}$ en élevant x_2 au carré, $x_3 = (x_2)^2$ (3ème mult.) ;

on devra élever x_3 à l'exposant 12 puis multiplier par x_2 (4ème mult.).

- 12 est pair. On obtient $(x_3)^{12}$ en élevant x_3 au carré, $x_4 = (x_3)^2$ (5ème mult.) ; on devra élever x_4 à l'exposant 6.
- 6 est pair. On obtient $(x_4)^6$ en élevant x_4 au carré, $x_5 = (x_4)^2$ (6ème mult.) ; on devra élever x_4 à l'exposant 3.
- 3 est impair. On obtient $(x_5)^3$ en élevant x^5 au carré, (7ème mult.) puis en multipliant par x_5 (8ème mult.).

Au total : 8 multiplications, contre 100 pour l'algorithme naïf !

Complexité de l'exponentiation rapide

C(n) : nombre d'opérations arithmétiques dans expo_rapide(n)

C(0) = 0
C(1) = 0
...
C(n) = C(n/2) + 1 pour n pair, ou
C(n) = C(n/2) + 2 pour n impair
~~C(n) ... un peu compliqué~~ dans le cas général

Si $n=2^p$,
$C(2^p) = C(2^{p-1}) + 1$
$C(2^{p-1}) = C(2^{p-2}) + 1$
...
$C(2^2) = C(2^1) + 1$
$C(2) = C(1) + 1 = 1$
$C(n) = \log_2(n)$

p fois = $\log_2(n)$ fois

```
fonction expo_rapide(x : flottant, n : entier) retourne flottant
    res : flottant
debut
    si (n == 0) alors
        retourne 1.0
    fin si
    si (n == 1) alors
        retourne x
    fin si
    res = expo_rapide(x, n/2)
    si (n%2 ==0) alors        // n est pair
        retourne res * res
    sinon                     // n est impair
        retourne res * res * x
    fin si
fin fonction
```

La **complexité** en temps de expo_rapide(n) est **logarithmique**
ce qui est bien plus rapide que la linéarité de expo(n).

Complexité des tours de Hanoi

si n==1,
    déplacer c(1) = 1

sinon
    hanoi(n-1, ...) c(n-1)
    "déplacer" 1
    hanoi (n-1, ...) c(n-1)

on compte :
$c(n) = 2 \times c(n-1) + 1$
$c(n-1) = 2 \times c(n-2) + 1$
...
$c(2) = 2 \times c(1) + 1$
$c(1) = 1$

n fois

Donc $c(n) = 1 + 2 + ... + 2^n = 2^{n+1} - 1$!!! Complexité exponentielle : aie !

Complexités classiques sous forme récurrente

Savoir que ça existe pour le retrouver si besoin.

Soient a ≥ 1, b > 1.

Exemple : réduction linéaire de la dimension du problème

C(n) = a C(n-1) + b
C(0) = 0

Solution :
C(n) = b(a^n – 1) / (a – 1)

Application :
Hanoi n disques, nombre de noeuds d'un arbre binaire complet de hauteur n :
a = 2 , b = 1
$C_{Hanoi,\ abc}(n) = 2^n - 1$
Aie : exponentiel !

Exemple : diviser (pour régner) un problème de taille n en *b* parties égales

$$C(1) = 1, \text{ et pour } n \geq 2 :$$
$$C(n) = a \cdot C(\lceil n/b \rceil) + d(n),$$

Solution générale compliquée

Quelques solutions pour $n=2^k$:
- exponentiation rapide : a=1, b=2, d(n)=1 ou 2
  $C_{expo\ rapide}(n) = \log_2 n$
  Bien : logarithmique

- tri fusion : a = b = 2 , d(n) = n-1
  $C_{tri\ fusion}(n) = n \log_2 n + 1$
  Bien : semi-logarithmique

Fibonaci : complexité

Que mesurer ?
- le nombre d'appels récursifs : a_n
- le nombre de sommes : s_n

Relations de récurrences : $a_0=a_1=s_0=s_1=0$ et

$$a_n = 1 + a_{n-1} + 1 + a_{n-2}$$
$$s_n = s_{n-1} + 1 + s_{n-2}$$

On montre que :

$$a_n = \left(\frac{1+\sqrt{5}}{2}\right)^n - 2$$
$$s_n = \left(\frac{1+\sqrt{5}}{2}\right)^n - 1$$

Plus technique à établir
- sera traité en exercice

```
fonction fib(n : entier) retourne entier
// f(0)=f(1)=1, f(n+1) = f(n) + f(n-1), n > 1
    res : entier
debut
    si (n == 0) ou (n==1) alors
        res = 1
    sinon
        res = fib(n) + fib(n-1)
    fin si
    retourne res
fin fonction
```

Aie, aie, aie !!
Le nombre d'or = 1.618… > 1
donc complexités exponentielles des appels et des additions.

Conclusion :
une solution itérative, SVP !
complexité linéaire !

Synthèse : correction, terminaison, complexité

Correction
- Arme fatale : démonstration par récurrence
- Plus facilement identifiable que les invariants de boucle : les répétitions sont cachées dans les appels qui, eux-mêmes, découlent de la définition …

Terminaison
- Paramètre(s) d'appel strictement décroissant : diviser-pour-régner
- Cas terminaux : ne pas en oublier ! … origine classique du *stack overflow*

Complexité
- Expression naturellement récurrente
- Dichotomie → supposer $n= 2^k$ simplifie l'analyse et donne l'ordre de grandeur de la complexité
- Des formules générales … un peu chimiques mais utiles !

Fonctions récursives

4 - compléments : autres formes, élimination de la récursivité terminale

Autres formes de fonctions récursives

Récursion forte : le ou les appels récursifs d'ordre n s'effectuent sur des valeurs strictement inférieures à n-1.

Exemple : Fibonacci (qui croît très vite), exponentiation rapide, diviser-pour-régner

Fonctions mutuellement récursives ou récursivité croisée : la définition de la fonction *f* fait appel à la fonction *g* et celle de la fonction *g* à la fonction *f*.

Exemple : une fonction pair qui détermine si un entier est pair par récurrence mutuelle avec une fonction impair qui détermine si un entier est impair

pair(n) retourne n==0 ou impair(n-1)

impair(n) retourne n!=0 ou pair(n-1)

Récursions imbriquées : les données de l'appel récursif contiennent un appel récursif

Exemple : fonction d'Ackermann-Péter (qui croît extrêmement rapidement ... attention !)

A(0,n) = n+1 A(m+1, 0) = A(m, 1) A(m+1, n+1) = A(m, A(m+1, n))

Récursivité terminale

Définition

Une fonction est récursive terminale si tout appel récursif de f est de la forme **retourne f(..)**

Exemple

factorielle(n) : retourne n*factorielle(n-1) n'est pas récursive terminale ...

mais une version récursive terminale est introduite à la diapo suivante

Intérêt

La valeur retournée est directement la valeur obtenue par l'appel récursif, sans aucun calcul ni modification de celle-ci.

Il n'y a pas besoin de stocker dans la pile d'exécution la valeur retournée par l'appel

Dérécursivation

L'appel à une fonction récursive terminale peut être transformée en une boucle (version itérative) sans occupation de mémoire supplémentaire

Factorielle sous forme récursive terminale

```
fonction factorielleRecursiveTerminale(n : entier, res : entier) retourne entier
// version récursive terminale : ajout d'un paramètre qui "remonte" les résultats partiels
debut
    si n==1 alors
        retourne res
    sinon
        retourne factorielle(n-1, n * res)
    fin si
fin fonction
```

Appel

Les calculs s'effectuent en descendant l'arbre des appels

```
r = factorielleRecursiveTerminale(3, 1) // factoriel(3)
    r = factorielleRecursiveTerminale(2, 3)
        r = factorielleRecursiveTerminale(1, 6)
                                          → 6
```


Dérécursivation de factorielle

```
fonction factorielleDerecursivee(n : entier) retourne entier
//  version qui devrait vous rappeller quelque chose !
    res : entier = 1
debut
    tant que n>1 faire
        res = res *n
        n = n-1
    fin tantque
    retourne res
fin fonction
```

Il existe un principe général de dérécursivation : voir diapo suivante

La transformation récursif terminal → itératif est automatisée par les compilateurs

Dérécursivation de forme récursive terminale

```
fonction RecursiveTerminale(p : type_parametre) retourne type_resultat
//  version récursive terminale
debut
      Bloc_Instructions_0          // Toujours exécuté
      si C alors                   // C est la condition de terminaison
          Bloc_Intructions_1              // Exécuté si C est vraie
      sinon
          Bloc_Intructions_2              // Exécuté avant l'appel récursif
          RecursiveTerminale (f(p))       // f : fonction de transformation des
paramètres
      fin si
fin fonction
```

```
fonction Derecursivee (p : type_parametre) retourne type_resultat
//  version itérative transformée
debut
      Bloc_Instructions_0
      tant que (non C) faire
          Bloc_Intructions_2
          p = f(p)
          Bloc_Intructions_0
      fin tant que
      Bloc_Instructions_1
fin fonction
```