Cours 2 : Méthodes d'analyse des algorithmes récursifs - Illustration sur les algorithmes de tri

Jean-Stéphane Varré

Université Lille 1

jean-stephane.varre@univ-lille1.fr

Rappels

Algorithme récursif :

- un algorithme qui se rappelle lui-même
- type de récursivité
  - simple ou linéaire (par exemple factorielle, tours de Hanoï)
  - croisée ou mutuelle (par exemple test de parité)
- on peut dessiner les appels grâce à un arbre

Rappels

Conception :

- tout algorithme récursif doit distinguer plusieurs cas
- il doit y avoir au moins un cas qui ne comporte pas d'appel récursif : les cas de base
  (par exemple lorsque le tableau possède 0 éléments)
- définir les bons cas de base : ils doivent être atteignables quelque soit l'exemplaire en entrée
  (par exemple en s'assurant que le tableau dans l'appel récursif est plus petit que celui en entrée)

Permet souvent d'exprimer de manière simple la résolution d'un problème

? Quelle différence entre ces deux programmes ?

```
1  # factorielle recursive
2  def fact(n):
3    if n == 0:
4      return 1
5    else:
6      return n * fact(n-1) * est effectue apres l'appel a fact
```

```
1  # somme recursive
2  def somme (n,r):
3    if n <= 1:
4      return r + n
5    else:
6      return somme(n - 1,r + 1) + est effectue avant l'appel a somme
```

Récursivité terminale

se dit d'une récursivité dont l'appel récursif est la toute dernière instruction réalisée

- ce n'est pas le cas dans `fact` : après l'appel récursif il faut faire une multiplication : `n * (fact (n-1))`

  tous les résultats des appels à `fact(n-1)`, `fact(n-2)`, etc. doivent être stockés dans une pile.

  $$\begin{array}{ccccccccccc} & & \text{fact}(5) & \leftarrow & \text{fact}(4) & \leftarrow & \text{fact}(3) & \leftarrow & \text{fact}(2) & \leftarrow & \text{fact}(1) \\ 120 & \times 5 & 24 & \times 4 & 6 & \times 3 & 2 & \times 2 & 1 \end{array}$$

- c'est le cas dans `somme` : l'addition `r + n` est faite avant l'appel

  pas de stockage de résultats intermédiaires, les appels successifs sont vus comme des égalités

  $\text{som}(5,1) = \text{som}(4,5) = \text{som}(3,9) = \text{som}(2,12) = \text{som}(1,14) = 15$

Paramètre d'accumulation

Application du même principe pour le calcul de la factorielle :

```
1  # factorielle avec parametre d'accumulation
2  def aux (n,a):
3      if n == 0:
4        return a
5      else:
6        return aux(n-1,n * a)
7
8  def fact_acc(n):
9    return aux(n,1)
```

Le paramètre ajouté qui stocke le résultat est appelé paramètre d'accumulation.

Avantages :

- théoriquement plus de nécessité de garder en mémoire la pile d'appels récursifs
- ces programmes peuvent être écrits facilement de manière itérative

"Dérécursivation"

```
1  def fact(n):
2    if n == 0:
3      return 1
4    else:
5      return n * fact(n-1)
```

```
1  def aux (nn,a):
2      if nn == 0:
3        return a
4      else:
5        return aux(nn-1,nn * a)
6
7  def fact_acc(n):
8    return aux(n,1)
```

```
1  def fact_iter(n):
2    a = 1
3    nn = n
4    while (nn > 0):
5      a = a * nn
6      nn = nn - 1
7    return a
```


La suite de Fibonacci

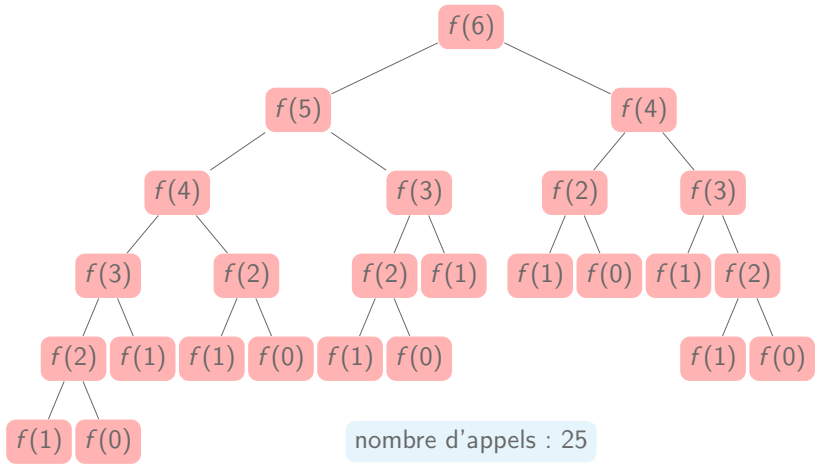
Définition du problème :

$$\begin{cases} f(0) & = & 0 \\ f(1) & = & 1 \\ f(n) & = & f(n-1) + f(n-2), n > 1 \end{cases}$$

Programmation directe :

```
1  # version recursive
2  def fibonacci(n):
3    if (n == 1) or (n == 0):
4      return n
5    else:
6      return (fibonacci (n-1)) + (fibonacci (n-2))
```

Nombre d'appels de fonction



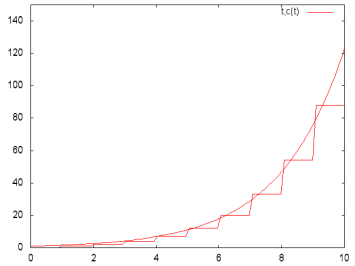
Expression du nombre d'appels

- expression du nombre d'appels à la fonction :

$$\begin{cases} c(0) &= 1 \\ c(1) &= 1 \\ c(n) &= 1 + c(n-1) + c(n-2) \end{cases}$$

- comportement asymtotique :

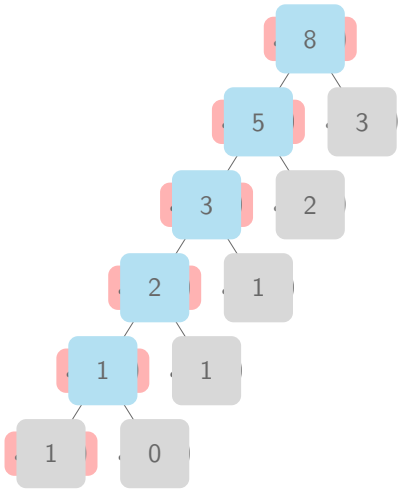
$$c(n) = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$



La suite de Fibonacci, récursive à mémoire

```
1  # version recursive avec stockage de donnees calculees
2  def aux (t,n):
3    if n > 1:
4      # t[n] = 0 <=> t[n] n'a pas encore ete calcule
5      if t[n] == 0:
6        # on a besoin de t[n-1]
7        t[n-1] = aux(t,n-1)
8        # mais on sait que t[n-2] aura aussi ete calcule,
9        # par effet de bord de l'appel a aux(t,n-1),
10       # inutile de faire un appel a aux (n-2)
11       t[n] = t[n-1] + t[n-2]
12   return t[n]
13
14 def fibonacci_mem(n):
15   t = [0 for i in range (0,n+1)]
16   # initialisation
17   t[0] = 0
18   t[1] = 1
19   return aux(t,n)
```


Fonctionnement et nombre d'appels récursifs



	0	1	2	3	4	5	6
t :	0	1	1	2	3	5	8

nombre d'appels : 6

Le tri par insertion dichotomique

- principe, exemple
- code de l'insertion (la partie tri ne change pas)

```
1  # insertion de t[i] dans la tranche t[0:i+1]
2  def binary_insert (t,i):
3      index = binary_search (t,0,i-1,t[i])
4      aux = t[i]
5      # decalage des valeurs
6      for k in reversed(range(index,i)):
7          t[k+1] = t[k]
8      # placement de la nouvelle valeur
9      t[index] = aux
```



```
1  # recherche dichotomique de la position
2  # d'insertion de v dans la tranche t[a:b+1]
3  def binary_search (t,a,b,v):
4      if a >= b - 1:
5          if cmp(v,t[a]):
6              return a
7          elif cmp(v,t[b]):
8              return b
9          else:
10             return b + 1
11     else:
12         m = (a + b) // 2
13         if cmp(v,t[m]):
14             return binary_search(t,a,m,v)
15         else:
16             return binary_search(t,m,b,v)
```

Tri par insertion dichotomique - analyse