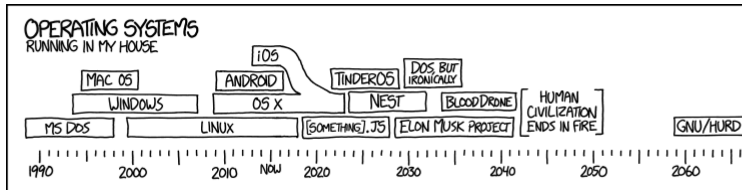
Récursion vs. itération

En passant par la récursion terminale

stephane@gonnord.org - http://blog.psi945.fr

Lycée du parc - Lyon

Vendredi 8 mai 2015 – Luminy



Plan

1. Le terrain de jeu
   - Des histoires de passeport
   - Un exemple jouet
2. Euclide étendu
   - Algorithme d'Euclide
   - Version étendue
   - Calcul itératif des coefficients
3. Tri rapide
   - Principe et problème
   - Solution théorique
   - Mise en place itérative
4. Mais encore
   - Hanoï
   - Fibonacci
   - Ackerman

Il y a environ 3 mois...

« Ralala, tous ces profs de prépa qui font calculer les coefficients de Bezout en récursif... »

$$\cdots \mathrm{SL}_2(\mathbb{Z}) \cdots$$

- Qu'est-ce qu'un bon algorithme ?
  - complexité en temps...
  - en espace ;
  - robustesse, preuve, passage à l'échelle.
- Qu'est-ce qu'un bon algorithme à raconter à nos élèves ?
  - complexités ;
  - idées naturelles ;
  - code simple, qu'ils peuvent écrire eux-même ;
  - exécutable à la main ?
- Alors, récursif ou itératif ?

Des histoires de passeport

Le petit François fait une demande de passeport...

- dans sa mairie d'arrondissement...
- qui transmet à la mairie centrale...
- qui transmet à la préfecture...
- qui transmet au ministère de l'intérieur...
- qui transmet à l'imprimerie nationale.

Et une fois le passeport imprimé, qu'est-ce qui se passe ?

Récursivité terminale

Qu'est-ce qu'une fonction « récursive » ?

- Pfiou... faudrait demander au jury de l'agreg !
- Bon, proposons $f(x) = \begin{cases} g(x) & \text{si } x \in \mathcal{B} \\ h(x, f(x')) & \text{sinon} \end{cases}$
- Non, plutôt $f(x) = \begin{cases} g(x) & \text{si } x \in \mathcal{B} \\ h(x, f(x'_1), \ldots, f(x'_k)) & \text{sinon} \end{cases}$
- Non, même pas...
- Bon, et pour le passeport ?

$$f(x) = \begin{cases} g(x) & \text{si } x \in \mathcal{B} \\ f(x') & \text{sinon} \end{cases}$$

- Call vs. jump.

Un exemple jouet : calculer $\sum_{k=1}^{n} k$

- En récursif, puisque c'est le thème !

```
def somme1(n):
    if n == 0:
        return 0
    return n + somme1(n-1)
```

- Problème dès que n est de l'ordre de 1000.
- Un autre récursif :

```
def somme_avec_accu(accu, n):
    if n == 0:
        return accu
    return somme_avec_accu(accu+n, n-1)


def somme2(n):
    return somme_avec_accu(0, n)
```

Au delà du jouet

- Dérécursifions `somme2` :

```
def somme3(n):
    accu, k = 0, n
    while k > 0:
        accu, k = accu+k, k-1
    return accu
```

- Calcul de $n!$ et x^n : même chose... quoique !
- Exercice : dérécursifier ceci :

```
def expo(x, n):
    if n == 0:
        return 1
    if n % 2 == 0:
        return expo(x**2, n//2)
    return x * expo(x**2, n//2)
```

Exponentiation rapide

- On rend la récursion terminale.

```
def expo_rap(x, n):
    return expo_rat_term(1, x, n)
def expo_rap_term(accu, x, n):
    if n == 0: return accu
    if n%2 == 0: return expo_rap_term(accu, x*x, n//2)
    else: return expo_rap_term( x*accu , x*x, n//2)
```

- On itère.

```
def expo_rap_iter(x, n):
    accu, y, p = 1, x, n
    while p > 0:
        if p%2 == 1: accu *= y
        y, p = y*y, p//2
    return accu
```

Avant d'aller plus loin...

Les récursivités sont-elles terminales sur ces exemples ?

- Calcul de $pgcd(a,b)$:

  $$pgcd(a,b) = pgcd(b,r),$$

  avec $a = bq + r$.
- Calcul de u et v tel que $au + bv = pgcd(a,b)$
  (fastoche si on a $bu_1 + rv_1 = pgcd(a,b)$).
- Tri fusion.
- Tri rapide.

Algorithme d'Euclide

Comment calculer $pgcd(a,b)$?

$$\text{Si } a = bq + r, \text{ alors } pgcd(a,b) = pgcd(b,r)$$

42	15
15	12
12	3
3	0

$$42 = 2 \times 15 + 12$$
$$15 = 1 \times 12 + 3$$
$$12 = 4 \times 3 + 0$$
$$pgcd(42,15) = 3$$

Récursif (terminal)

```
def pgcd(a, b):
    if b == 0:
        return a
    return pgcd(b, a%b)
```

Itératif

```
def pgcd2(a,b):
    ac, bc = a, b
    while bc > 0:
        ac, bc = bc, ac%bc
    return ac
```

Complexité(s) ?

On suppose : $0 \leq a, b \leq n$

- Nombre d'étapes : $O(\ln n)$.
- Chaque division euclidienne : $O(\ln^2 n)$ opérations élémentaires sur les bits... Donc $O(\ln^3 n)$.
- En fait, c'est $O(\ln^2 n)$!C'est-à-dire $0(1)$ dans beaucoup de contextes.
- Et en espace ? $\ln^2 n$ (récursif) vs. $\ln n$ (itératif).

Euclide étendu

Comment trouver u et v tels que $au + bv = pgcd(a, b)$?

- Si $a = bq + r$ et $bu_1 + rv_1 = pgcd$, alors $av_1 + b(u_1 - qv_1) = pgcd$.
- « Si tu résous le problème pour (b, r) alors je saurai le résoudre pour (a, b) ».
- Super facile à coder !

```
def bezout(a,b):
    if b == 0:
        return (1,0)
    (u1,v1) = bezout(b, a%b)
    return (v1, u1-(a//b)*v1)
```

- Horrible à exécuter (enfin, je trouvais ça difficile en 1989... et encore maintenant).

Même pas peur

42	15
15	12
12	3
3	0

$$42 = 2\times 15 + 12$$
$$15 = 1\times 12 + 3$$
$$12 = 4\times 3 + 0$$
$$pgcd(42,15) = 3$$

$$\begin{aligned} 3 &= \underbrace{3}_{a_3=b_2}\times 1 + \underbrace{0}_{b_3=a_2-4b_2}\times 0 = \underbrace{12}_{a_2=b_1}\times 0 + \underbrace{3}_{b_2=a_1-b_1}\times\underbrace{(1-0\times 4)}_{1} \\ &= \underbrace{15}_{a_1=b_0}\times 1 + \underbrace{12}_{b_1=a_0-2b_0}\times\underbrace{(0-1)}_{-1} = 42\times(-1) + 15\underbrace{(1+2)}_{3} \end{aligned}$$

$$(-1)\times 42 + 3\times 15 = 3$$

Yeah !

Vers l'itératif

- Rappel :
  - Récursivité terminale = « voila le matos pour calculer le résultat ; débrouille toi »
  - Récursivité générale = « calcule moi ça, puis je vais faire ce qu'il faut pour trouver le résultat qu'on m'a demandé »
- Ici : $\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} v_1 \\ u_1 - qv_1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix} \begin{pmatrix} u_1 \\ v_1 \end{pmatrix}$
- Plutôt que « calcule $\begin{pmatrix} u_1 \\ v_1 \end{pmatrix}$ pour (b, r) et je le cognerai contre $\begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix}$ » *un peu comme pour $n!$ en récursif...*
- On pourrait dire : « Quand tu auras les coefficients pour (b, r), tu les cogneras contre telle matrice ».

Un algorithme itératif

- L'algorithme :

  Entrées : $(a, b) \in \mathbb{N}^2 \setminus \{(0,0)\}$

  $(a_c, b_c, M_c) \leftarrow \left(a, b, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\right)$

  tant que $b_c > 0$ **faire**

  $M_c \leftarrow M_c \times \begin{pmatrix} 0 & 1 \\ 1 & -(a_c // b_c) \end{pmatrix}$

  $(a_c, b_c) \leftarrow (b_c, a_c \% b_c)$

  Résultat : Première colonne de M_c
- Terminaison : $a_c + b_c$ décroît strictement (à partir de la deuxième étape !)
- Invariant : « Si $a_c \times u_1 + b_c \times v_1 = pgcd(a, b)$ alors en posant $\begin{pmatrix} u \\ v \end{pmatrix} = M_c \begin{pmatrix} u_1 \\ v_1 \end{pmatrix}$ on aura $a \times u + b \times v = pgcd(a, b)$ »... ce qui prouve la correction.

C'est reparti !

$$\begin{array}{c|c} 42 & 15 \\ 15 & 12 \\ 12 & 3 \\ 3 & 0 \end{array} \qquad \begin{array}{c} 42 = 2 \times 15 + 12 \\ 15 = 1 \times 12 + 3 \\ 12 = 4 \times 3 + 0 \\ pgcd(42, 15) = 3 \end{array}$$

$$\begin{array}{ccccc} & \begin{pmatrix} 0 & 1 \\ 1 & -2 \end{pmatrix} & \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix} & \begin{pmatrix} 0 & 1 \\ 1 & -4 \end{pmatrix} & \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & \begin{pmatrix} 0 & 1 \\ 1 & -2 \end{pmatrix} & \begin{pmatrix} 1 & -1 \\ -2 & 3 \end{pmatrix} & \begin{pmatrix} -1 & \star \\ 3 & \star \end{pmatrix} & \begin{pmatrix} -1 \\ 3 \end{pmatrix} \end{array}$$

$$(-1) \times 42 + 3 \times 15 = 3$$

Fastoche !

LE CODE – LE CODE

```
def bezout2(a, b):
    ac, bc = a, b
    mc = np.identity(2, dtype = 'int64')
    while bc != 0:
        mc = mc.dot(np.array([[0, 1], [1, -(ac//bc)]]))
        ac, bc = bc, ac%bc
    return list(mc[:, 0])
```

```
>>> bezout2(42, 15)
[-1, 3]
>>> bezout2(15, 42)
[3, -1]
```

Bon, et donc ?

- En temps, les deux algorithmes sont en $O(\ln^2 n)$ opérations élémentaires ($0 \leq a, b \leq n$).

  À vérifier (pour ceux qui s'ennuient et n'ont pas osé partir)
- Pour la version récursive :
  - C'est celle que j'écri(vai)s naturellement !
  - Les élèves reconstituent l'algorithme (plus) facilement.
  - Fonctionnement limpide.
  - La version itérative est (un peu) magique ! Enfin, l'invariant est un tout petit peu élaboré.
- Pour la version itérative :
  - C'est (un peu) magique, justement !
  - C'est beaucoup plus facile à tracer à la main que la version récursive.
  - En espace, on passe de $\ln^2 n$ à $\ln n$... Yeah !

Pour trier une zone de tableau...

- On scinde (en place) cette zone ; on place le pivot.
- On trie la première partie.
- On trie la seconde partie.

Entrées : T, d, f

$i_1, i_2 \leftarrow d, d$ # Si, vraiment !

tant que $i_2 < f$ **faire**

  si $T[i_2 + 1] \leq p$ **alors**

    $T[i_1 + 1] \leftrightarrow T[i_2 + 1]$

    $i_1 \leftarrow i_1 + 1$

  $i_2 \leftarrow i_2 + 1$

$T[d] \leftrightarrow T[i_1]$

Résultat : i_1

Code de la préparation de la zone

```
def preparation(t, debut, fin):
    pivot = t[debut]
    while i2 < fin:
        if t[i2+1] < pivot:
            t[i1+1], t[i2+1] = t[i2+1], t[i1+1]
            i1 += 1
        i2 += 1
    t[debut], t[i1] = t[i1], t[debut]
                # placer le pivot dans le sandwich
    return i1
```

Code du tri rapide

- ```
  def tri_rapide_zone(t, debut, fin):
      if debut<fin:
          p = preparation(t, debut, fin)
          tri_rapide_zone(t, debut, p-1)
          tri_rapide_zone(t, p+1, fin)

  def tri_rapide(t):
      tri_rapide_zone(t, 0, len(t)-1)
  ```
- Complexité : quadratique si « ça se passe mal » (...)
- En plus : n appels récursifs imbriqués !
  - la pile de récursion explose ;
  - le tri n'est plus (vraiment) en place ;
  - en fait, il ne va pas marcher dès que n dépasse quelques milliers/millions !
  ```

Solution théorique

- Seul le premier appel récursif est non terminal.
- Il n'y a qu'à faire la *plus petite* zone en premier !

```
def tri_rapide_zone(t, debut, fin):
    if debut<fin:
        p = preparation(t, debut, fin)
        if p-debut <= fin-p:
            tri_rapide_zone(t, debut, p-1)
            tri_rapide_zone(t, p+1, fin)
        else:
            tri_rapide_zone(t, p+1, fin)
            tri_rapide_zone(t, debut, p-1)
```

- Pile de récursion (non terminale) de hauteur $O(\ln n)$.
- Ça ne marche pas ! Récursion terminale non détectée.

Mise en place itérative

- Idée : dérécursifier avec une pile `TODO`.
- Algorithme :

  Entrées : *T*, *d* et *f*

  ```
  TODO←PileVide()
  Empiler( (d,f), TODO )
  ```

  tant que *NOT (EstVide(TODO))* **faire**

  (d_1, f_1) ←`Depiler( TOTO )`

  si $d_1 < f_1$ **alors**

  $p \leftarrow preparation(d_1, f_1)$

  ```
  Empiler( (d1, p-1), TODO)
  Empiler( (p+1, f1), TODO)
  ```

- Rha, j'ai oublié ces histoires de plus petite zone à traiter en premier !

LE CODE – LE CODE

```
def tri_rapide_iter(t):
    TODO = creer_pile()
    empiler((0, len(t)-1), TODO)
    while not(est_vide(TODO)):
        (d,f) = depiler(TODO)
        if d < f:
            p = preparation_zone(t, d, f)
            if p-d <= f-p:
                empiler((p+1,f), TODO)
                empiler ((d,p-1), TODO)
            else:
                empiler ((d,p-1), TODO)
                empiler((p+1,f), TODO)
```

Pour terminer avec le tri rapide

- Temps d'exécution dégradé ? (temps en secondes ; moyenne sur dix tableaux aléatoires)

n	Tri récursif	Tri avec pile
10^4	0.071	0.086
10^5	0.90	1.07
10^6	13.1	14.5

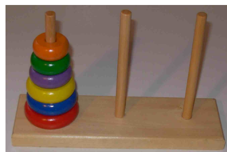
  And the winner is ?

- Pile de hauteur $O(\ln n)$?

(d_k, f_k)
$\dots$
$\dots$
(d_1, f_1)

$$f_k - d_k \leq \frac{n}{2^{k-1}}$$

Hanoï

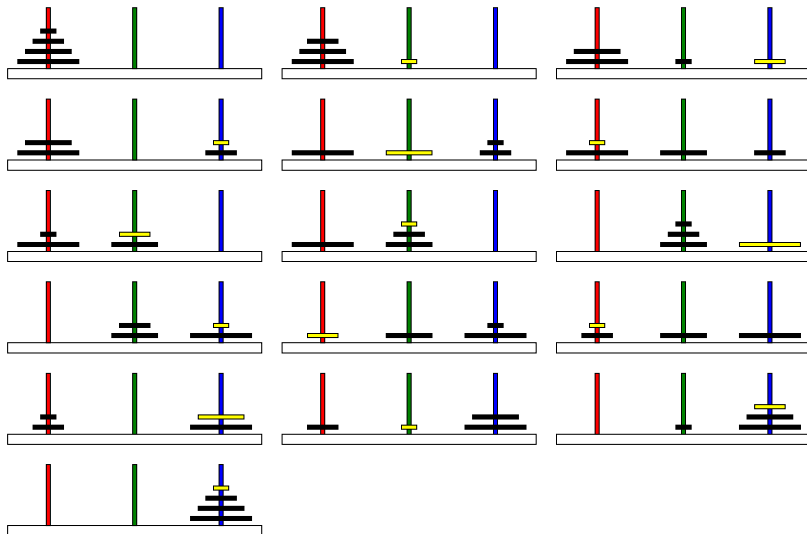


- Une solution récursive facile à programmer...

```
def hanoi(n, a, b):
    """ n disques à transporter de a vers b"""
    c = 6-(a+b) # le piquet intermédiaire
    if n > 0:
        hanoi(n-1, a, c)
        print("Déplacer une pièce de %i vers %i"%(a,b))
        hanoi(n-1, c, b)
```

- et horrible à exécuter.
- Exercice : dérécursifier !

Histoire de frimer



Fibonacci (encore et toujours)

$$f_{n+2} = f_n + f_{n+1}$$

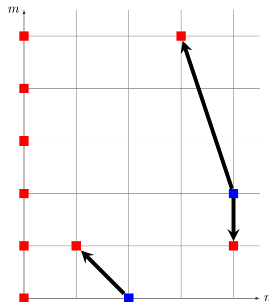
- Récursion naïve : exponentielle (en n) en temps ; pas en espace (quelle complexité, au fait ?)
- « La bonne façon de faire » : $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n$
  - $O(n)$ opérations élémentaires sur les bits...
  - ou $O(\ln n)$ en modulaire.

  Oublions !
- Itératif+tableau... ou bien récursif+dictionnaire (`remember`) ? Qu'est-ce qui est le plus simple, le moins coûteux ?
- Mêmes idées en programmation dynamique !

Last but not least : Ackerman

$$\mathrm{Ack}(n,m) = \begin{cases} m+1 & \text{si } n=0 \\ \mathrm{Ack}(n-1,1) & \text{si } n>0 \text{ et } m=0 \\ \mathrm{Ack}(n-1,\mathrm{Ack}(n,m-1)) & \text{sinon} \end{cases}$$

Gni ?



C’est fini

Des questions ?

Merci de votre attention !

PAGE 3

DEPARTMENT	COURSE	DESCRIPTION	PREREQS
COMPUTER SCIENCE	CPSC 432	INTERMEDIATE COMPILER DESIGN, WITH A FOCUS ON DEPENDENCY RESOLUTION.	CPSC 432