Trois questions d'algorithmique

Journées nationales de l'APMEP – 20/10/2013

G. Aldon - J. Germoni - J.-M. Mény (IREM de Lyon)

Trois problèmes basiques

Supposons avoir construit un algorithme pour résoudre un problème.

Est-ce que l'algorithme donne...

1. une réponse ? $\rightsquigarrow$ terminaison
2. la bonne réponse ? $\rightsquigarrow$ correction
3. la bonne réponse en un temps acceptable ? $\rightsquigarrow$ complexité

Terminaison

Preuve de terminaison

Mise en évidence d'un convergent, i.e. une quantité qui diminue à chaque passage, vivant dans un ensemble bien fondé (où il n'existe pas de suites infinies strictement décroissantes).

```
Algorithme PGCD
Entrée : a, b entiers
Sortie : un entier
Variables locales : x, y, r
 x := a ; y := b ;
 tant que y != 0 faire
      r := reste de la division de x par y
      x := y
      y := r  //
 renvoyer x
```

Terminaison

Preuve de terminaison

Mise en évidence d'un convergent, i.e. une quantité qui diminue à chaque passage, vivant dans un ensemble bien fondé (où il n'existe pas de suites infinies strictement décroissantes).

```
Algorithme PGCD
Entrée : a, b entiers
Sortie : un entier
Variables locales : x, y, r
 x := a ; y := b ;
 tant que y != 0 faire
      r := reste de la division de x par y
      x := y
      y := r   //nouvelle valeur de y < ancienne valeur
 renvoyer x
```

Correction - validité

Preuve de correction

Mise en évidence d'un invariant de boucle, i.e. une assertion qui est vraie avant l'entrée dans la boucle et qui, si elle est vraie au début d'un passage, reste vraie en fin de passage. Donc vraie en sortie.

```
Algorithme PGCD
Entrée : a, b entiers
Sortie : un entier
Variables locales : x, y, r
  x := a ; y := b ;          //
  tant que y != 0 faire
      r := reste de la division de x par y
      x := y                 //
      y := r                 //
  renvoyer x                 //
```

Correction - validité

Preuve de correction

Mise en évidence d'un invariant de boucle, i.e. une assertion qui est vraie avant l'entrée dans la boucle et qui, si elle est vraie au début d'un passage, reste vraie en fin de passage. Donc vraie en sortie.

```
Algorithme PGCD
Entrée : a, b entiers
Sortie : un entier
Variables locales : x, y, r
  x := a ; y := b ;          // a ∧ b = x ∧ y
  tant que y != 0 faire
      r := reste de la division de x par y
      x := y                 //
      y := r                 //
  renvoyer x                 //
```

Correction - validité

Preuve de correction

Mise en évidence d'un invariant de boucle, i.e. une assertion qui est vraie avant l'entrée dans la boucle et qui, si elle est vraie au début d'un passage, reste vraie en fin de passage. Donc vraie en sortie.

```
Algorithme PGCD
Entrée : a, b entiers
Sortie : un entier
Variables locales : x, y, r
  x := a ; y := b ;          // a ∧ b = x ∧ y
  tant que y != 0 faire
      r := reste de la division de x par y
      x := y                 // x = qy + r,   0 ≤ r < y
      y := r                 // a ∧ b = x ∧ y = y ∧ (x − qy)
  renvoyer x                 //
```

Correction - validité

Preuve de correction

Mise en évidence d'un invariant de boucle, i.e. une assertion qui est vraie avant l'entrée dans la boucle et qui, si elle est vraie au début d'un passage, reste vraie en fin de passage. Donc vraie en sortie.

```
Algorithme PGCD
Entrée : a, b entiers
Sortie : un entier
Variables locales : x, y, r
  x := a ; y := b ;          // a ∧ b = x ∧ y
  tant que y != 0 faire
      r := reste de la division de x par y
      x := y                 // x = qy + r,   0 ≤ r < y
      y := r                 // a ∧ b = x ∧ y = y ∧ (x − qy)
  renvoyer x                 // a ∧ b = x ∧ 0 = x
```

Fractions égyptiennes

Problème : $a < b$ donnés, trouver $n_1 < \cdots < n_r$ t. q. $\dfrac{a}{b} = \displaystyle\sum_{k=1}^{r} \frac{1}{n_k}$.

Pré-algorithme :

- choisir n *aussi petit que possible* tel que $a/b \geq 1/n$,
- remplacer $\dfrac{a}{b}$ par $\dfrac{a}{b} - \dfrac{1}{n} = \dfrac{na - b}{nb}$,
- recommencer autant que nécessaire.

Exemple : $\dfrac{17}{31}$

Fractions égyptiennes

Problème : $a < b$ donnés, trouver $n_1 < \cdots < n_r$ t. q. $\dfrac{a}{b} = \displaystyle\sum_{k=1}^{r} \frac{1}{n_k}$.

Pré-algorithme :

- choisir n *aussi petit que possible* tel que $a/b \geq 1/n$,
- remplacer $\dfrac{a}{b}$ par $\dfrac{a}{b} - \dfrac{1}{n} = \dfrac{na - b}{nb}$,
- recommencer autant que nécessaire.

Exemple : $\dfrac{17}{31} = \dfrac{1}{2} + \dfrac{3}{62}, \quad \dfrac{3}{62}$

Fractions égyptiennes

Problème : $a < b$ donnés, trouver $n_1 < \cdots < n_r$ t. q. $\dfrac{a}{b} = \displaystyle\sum_{k=1}^{r} \frac{1}{n_k}$.

Pré-algorithme :

- choisir n *aussi petit que possible* tel que $a/b \geq 1/n$,
- remplacer $\dfrac{a}{b}$ par $\dfrac{a}{b} - \dfrac{1}{n} = \dfrac{na - b}{nb}$,
- recommencer autant que nécessaire.

Exemple : $\dfrac{17}{31} = \dfrac{1}{2} + \dfrac{3}{62}$, $\quad \dfrac{3}{62} = \dfrac{1}{21} + \dfrac{1}{62 \times 21}$.

Fractions égyptiennes

Problème : $a < b$ donnés, trouver $n_1 < \cdots < n_r$ t. q. $\dfrac{a}{b} = \displaystyle\sum_{k=1}^{r} \frac{1}{n_k}$.

Pré-algorithme :

- choisir n *aussi petit que possible* tel que $a/b \geq 1/n$,
  c'est-à-dire $n - 1 < b/a \leq n$,
  ou encore $n = \mathrm{ceil}(b/a) = -E(-b/a)$,
- remplacer $\dfrac{a}{b}$ par $\dfrac{a}{b} - \dfrac{1}{n} = \dfrac{na - b}{nb}$,
- recommencer autant que nécessaire.

Exemple : $\dfrac{17}{31} = \dfrac{1}{2} + \dfrac{3}{62}$, $\quad \dfrac{3}{62} = \dfrac{1}{21} + \dfrac{1}{62 \times 21}$.

Fractions égyptiennes : algorithme

Problème : $a < b$ donnés, trouver $n_1 < \cdots < n_r$ t. q. $\frac{a}{b} = \sum_{k=1}^{r} \frac{1}{n_k}$.

```
Algorithme Egypte
Entrée : a, b entiers
Sortie : une liste d'entiers
Variables locales : x, y, n: entiers; L: liste;
  x := a ; y := b ; L := []
  tant que x != 0 faire
      n := ceil(y/x)
      L := append(L,n)
      x := n*x-y
      y := n*y
  renvoyer L
```

Fractions égyptiennes : terminaison

Problème : $a < b$ donnés, trouver $n_1 < \cdots < n_r$ t. q. $\frac{a}{b} = \sum_{k=1}^{r} \frac{1}{n_k}$.

```
Algorithme Egypte
Entrée : a, b entiers naturels
Sortie : une liste d'entiers
Variables locales : x, y, n: ent; L: liste;
  x := a ; y := b ; L := []
  tant que x != 0 faire
      n := ceil(y/x)
      L := append(L,x)
      x := n*x-y
      y := n*y
  renvoyer L
```

La variable x est un convergent : $n-1 < \frac{y}{x} \leq n \Leftrightarrow 0 \leq nx - y < x$.

Fractions égyptiennes : correction

Problème : $a < b$ donnés, trouver $n_1 < \cdots < n_r$ t. q. $\frac{a}{b} = \sum_{k=1}^{r} \frac{1}{n_k}$.

```
Algorithme Egypte
Entrée : a, b entiers naturels
Sortie : une liste d'entiers
Variables locales : x, y, n: ent; L: liste;
  x := a ; y := b ; L := [] //
  tant que x != 0 faire
      n := ceil(y/x)
      L := append(L,x)
      x := n*x-y                    //
      y := n*y                      //
  renvoyer L                        //
```

Invariant de boucle : $\frac{a}{b} = \frac{x}{y} + \sum_{r \in L} \frac{1}{r}$.

Fractions égyptiennes : correction

Problème : $a < b$ donnés, trouver $n_1 < \cdots < n_r$ t. q. $\frac{a}{b} = \sum_{k=1}^{r} \frac{1}{n_k}$.

```
Algorithme Egypte
Entrée : a, b entiers naturels
Sortie : une liste d'entiers
Variables locales : x, y, n: ent; L: liste;
  x := a ; y := b ; L := [] //∑_{r∈L} 1/r = 0
  tant que x != 0 faire
      n := ceil(y/x)
      L := append(L,x)
      x := n*x-y                    //
      y := n*y                      //
  renvoyer L                        //
```

Invariant de boucle : $\frac{a}{b} = \frac{x}{y} + \sum_{r \in L} \frac{1}{r}$.

Fractions égyptiennes : correction

Problème : $a < b$ donnés, trouver $n_1 < \cdots < n_r$ t. q. $\frac{a}{b} = \sum_{k=1}^{r} \frac{1}{n_k}$.

```
Algorithme Egypte
Entrée : a, b entiers naturels
Sortie : une liste d'entiers
Variables locales : x, y, n: ent; L: liste;
  x := a ; y := b ; L := [] //
  tant que x != 0 faire
      n := ceil(y/x)
      L := append(L,x)
      x := n*x-y                    // x/y ~> (nx-y)/(ny) = x/y - 1/n
      y := n*y                      // sum_{r in L} 1/r ~> sum_{r in L} 1/r + 1/n
  renvoyer L                        //
```

Invariant de boucle : $\frac{a}{b} = \frac{x}{y} + \sum_{r \in L} \frac{1}{r}$.

Fractions égyptiennes : correction

Problème : $a < b$ donnés, trouver $n_1 < \cdots < n_r$ t. q. $\frac{a}{b} = \sum_{k=1}^{r} \frac{1}{n_k}$.

```
Algorithme Egypte
Entrée : a, b entiers naturels
Sortie : une liste d'entiers
Variables locales : x, y, n: ent; L: liste;
  x := a ; y := b ; L := [] //
  tant que x != 0 faire
      n := ceil(y/x)
      L := append(L,x)
      x := n*x-y                    //
      y := n*y                      //
  renvoyer L                        // x/y = 0 : gagné ! (*)
```

Invariant de boucle : $\frac{a}{b} = \frac{x}{y} + \sum_{r \in L} \frac{1}{r}$. (*) Reste à voir : r diminue

Complexité

Idée : faire sentir la nécessité d'évaluer *a priori* le temps mis par un algorithme pour s'exécuter.

Calcul des nombres de Fibonacci $(f_n)_{n\in\mathbb{N}}$ définis par

$$f_0 = f_1 = 1, \quad \forall n \in \mathbb{N} \setminus \{0,1\},\ f_n = f_{n-1} + f_{n-2} :$$

Trois algorithmes :

- algorithme récursif,
- algorithme itératif,
- calcul de puissances.

Complexité de l'algorithme récursif

```
Algorithme fib_rec(n: entier)
    si n<2 alors renvoyer 1
    sinon renvoyer fib_rec(n-1)+fib_rec(n-2)
```

Opérations élémentaires :

- appels à la fonction `fib_rec` (et tests),
- sommes.

Complexité de l'algorithme récursif

```
Algorithme fib_rec(n: entier)
    si n<2 alors renvoyer 1
    sinon renvoyer fib_rec(n-1)+fib_rec(n-2)
```

Opérations élémentaires :

- a_n appels à la fonction `fib_rec` (et tests),
- s_n sommes.

Évaluation de (a_n)

Complexité de l'algorithme récursif

```
Algorithme fib_rec(n: entier)
    si n<2 alors renvoyer 1
    sinon renvoyer fib_rec(n-1)+fib_rec(n-2)
```

Opérations élémentaires :

- a_n appels à la fonction `fib_rec` (et tests),
- s_n sommes.

Évaluation de (a_n)

$$a_0 = a_1 = 0, \quad \forall n \geq 2,\ a_n = 1 + a_{n-1} + 1 + a_{n-2},$$

La suite $(a_n + 2)$ satisfait à une relation de récurrence linéaire.
Conséquence : $a_n = C\phi^n + C'\phi'^n \sim C\phi^n$, où $\phi = \frac{1+\sqrt{5}}{2}$, $C, C' > 0$.

Calcul de puissances

```
Algorithme puiss_rec(x,n: entier)
    si n=0 alors renvoyer 1
    si n pair alors
        y := puiss_rec(x,n/2)
        renvoyer y*y
    sinon
        y := puiss_rec(x,(n-1)/2)
        renvoyer x*y*y
```

Évaluation du nombre de multiplications : m_n ?

Calcul de puissances

```
Algorithme puiss_rec(x,n: entier)
    si n=0 alors renvoyer 1
    si n pair alors
        y := puiss_rec(x,n/2)
        renvoyer y*y
    sinon
        y := puiss_rec(x,(n-1)/2)
        renvoyer x*y*y
```

Évaluation du nombre de multiplications : m_n ?

Écrit n en base 2 : $n = [a_r, a_{r-1}, \ldots, a_1, a_0]$.

Si n pair : $m_n = m_{n/2} + 1$; si n impair, $m_n = m_{(n-1)/2} + 2$.

Calcul de puissances

```
Algorithme puiss_rec(x,n: entier)
    si n=0 alors renvoyer 1
    si n pair alors
        y := puiss_rec(x,n/2)
        renvoyer y*y
    sinon
        y := puiss_rec(x,(n-1)/2)
        renvoyer x*y*y
```

Évaluation du nombre de multiplications : m_n ?

Écrit n en base 2 : $n = [a_r, a_{r-1}, \ldots, a_1, a_0]$.

Si n pair : $m_n = m_{n/2} + 1$; si n impair, $m_n = m_{(n-1)/2} + 2$.

Or : si n pair : $\frac{n}{2} = [a_r, \ldots, a_1]$; sinon : $\frac{n-1}{2} = [a_r, \ldots, a_1] + 1$.

Dans tous les cas : $m([a_r, \ldots, a_1, a_0]) = m([a_r, \ldots, a_1]) + a_0 + 1$.

Calcul de puissances

```
Algorithme puiss_rec(x,n: entier)
    si n=0 alors renvoyer 1
    si n pair alors
        y := puiss_rec(x,n/2)
        renvoyer y*y
    sinon
        y := puiss_rec(x,(n-1)/2)
        renvoyer x*y*y
```

Évaluation du nombre de multiplications : m_n ?

Écrit n en base 2 : $n = [a_r, a_{r-1}, \ldots, a_1, a_0]$.

Si n pair : $m_n = m_{n/2} + 1$; si n impair, $m_n = m_{(n-1)/2} + 2$.

Or : si n pair : $\frac{n}{2} = [a_r, \ldots, a_1]$; sinon : $\frac{n-1}{2} = [a_r, \ldots, a_1] + 1$.

Dans tous les cas : $m([a_r, \ldots, a_1, a_0]) = m([a_r, \ldots, a_1]) + a_0 + 1$.

$$m_n = r + \sum_{k=0}^{r} a_k, \quad \log_2 n \leq m_n \leq 2 \log_2 n.$$

Tri par sélection : illustration

2	1	5	0	9	4

Tri par sélection : illustration

2	1	5	0	9	4
0	1	5	2	9	4

Tri par sélection : illustration

2	1	5	0	9	4
0	1	5	2	9	4

Tri par sélection : illustration

2	1	5	0	9	4
0	1	5	2	9	4
0	1	5	2	9	4

Tri par sélection : illustration

2	1	5	0	9	4
0	1	5	2	9	4
0	1	5	2	9	4
0	1	2	5	9	4

Tri par sélection : illustration

2	1	5	0	9	4
0	1	5	2	9	4
0	1	5	2	9	4
0	1	2	5	9	4
0	1	2	4	9	5

Tri par sélection : illustration

2	1	5	0	9	4
0	1	5	2	9	4
0	1	5	2	9	4
0	1	2	5	9	4
0	1	2	4	9	5
0	1	2	4	5	9

Tri par sélection

Entrée : T liste de nombres de taille n
Sortie : liste T triée
Traitement :
Pour j de 1 à $n-1$
indiceMin :=j
  Pour k de $j+1$ à n
    si $T[k] < T[j]$ alors indiceMin:= k finSi
  finPour
Echange de $T[j]$ et T[indiceMin] si $j \neq$ indiceMin
finPour

Complexité : tri par sélection

Complexité expérimentale : fichier SAGE ou fichier XCAS...

Complexité : tri par sélection

Complexité expérimentale : second degré

Complexité : tri par sélection

Complexité expérimentale : second degré

Nombre de comparaisons :

$$\sum_{j=1}^{n-1}\left(\sum_{k=j+1}^{n} 1\right) = \sum_{j=1}^{n-1}(n-j) = \frac{1}{2}n(n-1)$$

Nombre d'échanges : au plus le nombre de comparaisons.

Complexité en seconde ou première : un exemple simple pour le lycée

Écrire « le » programme suivant :
Entrée : un entier naturel $p > 0$.
Sortie : les triangles à côtés entiers, rectangles, de périmètre p.

Complexité en seconde ou première : un exemple simple pour le lycée

Écrire « le » programme suivant :
Entrée : un entier naturel $p > 0$.
Sortie : les triangles à côtés entiers, rectangles, de périmètre p.

Première version :

```
Algorithme triangles_entiers_v0(p: entier)
    pour a de 1 jusque p:
        pour b de 1 jusque p:
            pour c de 1 jusque p :
                tester le triplet (a,b,c)
                stocker (a,b,c) si satisfaisant
    renvoyer liste des triplets
```

Première approche expérimentale

Sur une calculatrice Ti82, plus de 5 secondes pour un périmètre $p = 10$.

Première approche expérimentale

Sur une calculatrice Ti82, plus de 5 secondes pour un périmètre $p = 10$.

Hypothèse de proportionnalité temps – nombre de boucles.
Quel temps pour un périmètre $p = 1000$?

$$\frac{5 \times 100^3}{3600 \times 24} \approx 58 \text{ jours}$$

Complexité : les triangles entiers

Amélioration de l'algorithme :

```
Algorithme triangles_entiers_v1(p: entier)
  pour a de 1 jusque p/3:
    pour b de a jusque p/2: // mieux : b de a à floor((p-a)/2)
      si p-a-b > b-1 alors: // ce test devient inutile
        tester le triplet (a,b,p-a-b)
        stocker (a,b,p-a-b) si satisfaisant
  renvoyer liste_des_triplets
```

Comparer les temps de calcul expérimentalement et expliquer.
FICHIER SAGE

Première version :
Nombre de tests : p^3.

Première version :
Nombre de tests : p^3.

Seconde version :
Nombre de tests :

$$\sum_{a=1}^{\lceil p/3 \rceil} \left(\frac{p}{2} - a + 1 \right) \leqslant \frac{1}{9} p(p+3)$$

Quelques remarques pour finir sur la complexité

- La notion d'opération élémentaire dépend du contexte.
- Principe : évaluer (majorer) le nombre d'opérations en fonction de la taille des données (valeur d'un argument, nombre de mots, nombre d'inconnues, nombre de chiffres...).
- Exemples :
  - Cryptographie RSA et factorisation : pour un nombre à 100 chiffres, 10^{50} tests naïfs = trop !
  - Algorithme de Gauss : complexité en $O(n^3)$ si les nombres ont une taille fixe (flottants, corps finis ; pas rationnels...). Exemple : 10^6 inconnues $\rightsquigarrow$ 10^{18} opérations $> 10^9$ secondes !
  - P = NP.