Complexité en algorithmique

Gilles Aldon, Jérôme Germoni, Jean-Manuel Mény

IREM de Lyon

Mars 2012

Trois questions essentielles

Un algorithme a pour objectif la résolution d'un problème.
Est-ce que l'algorithme donne...

Trois questions essentielles

Un algorithme a pour objectif la résolution d'un problème.
Est-ce que l'algorithme donne...

1. une réponse ? $\rightsquigarrow$ terminaison

Trois questions essentielles

Un algorithme a pour objectif la résolution d'un problème.
Est-ce que l'algorithme donne...

1. une réponse ? $\rightsquigarrow$ terminaison
2. la bonne réponse ? $\rightsquigarrow$ correction

Trois questions essentielles

Un algorithme a pour objectif la résolution d'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
Entree : 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
Entree : 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
Entree : 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
Entree : a, b entiers
Sortie : un entier                    // D(a,b) = diviseurs communs
Variables locales : x, y, r
  x := a ; y := b ;                   // D(a,b) = D(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
Entree : a, b entiers
Sortie : un entier                    //
Variables locales : x, y, r
  x := a ; y := b ;                   // D(a,b) = D(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                           // D(a,b) = D(x,y) = D(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
Entree : a, b entiers
Sortie : un entier                    //
Variables locales : x, y, r
  x := a ; y := b ;                   // D(a,b) = D(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                           // D(a,b) = D(x,y) = D(y, x - qy)
  renvoyer x                          // D(a,b) = D(x,0) : PGCD = x
```

Complexité : la suite de Fibonacci

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.

Temps de calcul avec l'algorithme récursif

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

Mise en place et mesure des temps de calcul

Temps de calcul avec l'algorithme récursif

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

Mise en place et mesure des temps de calcul

Instructions utiles

	Xcas	Sage
dessin de listes	`listplot(L)`	`list_plot(L)`

mesure du temps	`time(`*calcul*`)`	`t = cputime()`
		calculs
		`t = cputime()-t`

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 $\phi = \frac{1+\sqrt{5}}{2}$, $C, C' > 0$.

Aparté : calcul de puissances

Problème : calculer x^n (n entier) en minimisant le nombre de multiplications.

```
Algorithme Naive(x, n: entier)
    si n=0 alors renvoyer 1
    sinon renvoyer Naive(x,n-1)*x
```

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

Nombre d'opérations dans un calcul de puissances

Problème : calculer x^n (n entier) en minimisant le nombre de multiplications.

```
Algorithme nbPuissanceNaive(n: entier)
    si n=0 alors renvoyer 0
    sinon renvoyer nbPuissanceNaive(n-1)+1
```

```
Algorithme nbPuissanceRec(n: entier)
    si n=0 alors renvoyer 0
    sinon si n pair
              renvoyer puissRec(n/2)+1
          sinon
              renvoyer puissRec((n-1)/2)+2
```

Nombre d'opérations dans un calcul de puissances

Comptage des opérations

Soit n un entier non nul, $[a_r, a_{r-1}, \ldots, a_1, a_0]$ ses chiffres en base 2. Le nombre de produits faits par l'algorithme récursif est :

Nombre d'opérations dans un calcul de puissances

Comptage des opérations

Soit n un entier non nul, $[a_r, a_{r-1}, \ldots, a_1, a_0]$ ses chiffres en base 2. Le nombre de produits faits par l'algorithme récursif est :

$$M(n) = r - 1 + a_r + a_{r-1} + \cdots + a_1 + a_0.$$

Il est donc compris entre $r = \log(n)$ et $2\log(n)$.

Par récurrence sur r.

- Si n est pair : $a_0 = 0$ et $n/2 = [a_r, \ldots, a_1]$, d'où :

$$M(n) = M(\frac{n}{2}) + 1 = r - 2 + a_r + \cdots + a_1 + 1 = r - 1 + a_r + \cdots + a_1 + a_0.$$

- Si n est impair : $a_0 = 1$ et $(n-1)/2 = [a_r, \ldots, a_1]$, d'où :

$$M(n) = M(\frac{n-1}{2}) + 2 = r - 2 + a_r + \cdots + a_1 + 2 = r - 1 + a_r + \cdots + a_1 + a_0.$$

Application : suite de Fibonacci par les puissances

Pose, pour tout n entier :

$$F_n = \begin{pmatrix} f_n & f_{n+1} \end{pmatrix}.$$

Alors :

$$F_{n+1} = \begin{pmatrix} f_{n+1} & f_n + f_{n+1} \end{pmatrix} = F_n A \quad \text{où } A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}.$$

Application : suite de Fibonacci par les puissances

Pose, pour tout n entier :

$$F_n = \begin{pmatrix} f_n & f_{n+1} \end{pmatrix}.$$

Alors :

$$F_{n+1} = \begin{pmatrix} f_{n+1} & f_n + f_{n+1} \end{pmatrix} = F_n A \quad \text{où } A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}.$$

D'où :

$$F_n = F_0 A^n = F_{-1} A^{n+1} \quad \text{où } F_{-1} = \begin{pmatrix} f_{-1} & f_0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \end{pmatrix}.$$

Ainsi, F_n est la deuxième ligne de A^{n+1}.

Intérêt

Calcul de A^{n+1} en $O(\log n)$ opérations arithmétiques.
NB : Formule de Binet en diagonalisant A.

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 floor((p-a)/2)
      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

Complexité : les triangles entiers

Évaluation de la complexité

Première version :
Nombre de tests : p^3.

Complexité : les triangles entiers

Évaluation de la complexité

Première version :
Nombre de tests : p^3.

Seconde version :
Nombre de tests :

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

Complexité et ISN : tri par sélection

Le principe du tri par sélection d'une liste $T = (T[1], T[2], \ldots, T[n])$:

Pour chaque entier j $(1 \leqslant j \leqslant n-1)$:

- parcourir les éléments $T[j]$, $T[j+1]$, …, $T[n]$, retenir l'indice k du plus petit.
- placer au rang j le plus petit des éléments $T[j]$, $T[j+1]$, …, $T[n]$ (en échangeant $T[j]$ et $T[k]$).

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 1e : évaluation d'un polynôme en une valeur

Entrée : un polynôme p (liste de ses coefficients) et une valeur réelle x.
Sortie : $p(x)$

Exemples avec SAGE.

Quelques remarques sur la complexité

- Notion d'opération élémentaire dépendante 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.