1. **Introduction**

**Trois questions essentielles**

Un algorithme a pour objectif la résolution d'un problème.

Est-ce que l'algorithme donne...

* 1 Une réponse ? terminaison
* 2 La bonne réponse ? correction
* 3 La bonne réponse en un temps acceptable ? complexité

**1.1 Terminaison**

**1.1.1 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).

**Exemple :** Calcul de PGCD de deux entiers.

**Algorithme** PGCD

**Entree:** a , b e n t i e r s

**Sort i e :** un e n t i e r

**Variables** l o c a l e s : x , y , r

x := a ; y := b ;

tant que y != 0 f a i r e

r := r e s t e de l a d i v i s i o n de x par y

x := y

y := r //// **nouvelle valeur de y < ancienne valeur**

renvoyer x

**1.2 Correction – validité**

**1.2.1 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 enfin de passage. Donc vraie en sortie.

**Exemple**

**Algorithme PGCD**

Entree : a , b e n t i e r s

Sortie : un e n t i e r // D(a; b) = diviseurs communs

Variables l o c a l e s : x , y , r

x := a ; y := b ; // D(a; b) = D(x; y)

tant que y != 0 f a i r e

r := r e s t e de l a d i v i s i o n 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

* 1. **Complexité**

**1.3.1 Définition de la complexité d’algorithme**

La complexité (temporelle) d'un algorithme est le nombre d'opérations élémentaires (affectations, comparaisons, opérations arithmétiques) effectuées par un algorithme. Ce nombre s'exprime en fonction de la taille *n* des données. On s'intéresse au coût exact quand c'est possible, mais également au coût moyen (que se passe-t-il si on moyenne sur toutes les exécutions du programme sur des données de taille *n*), au cas le plus favorable, ou bien au cas le pire. On dit que la complexité de l'algorithme est *O*(*f*(*n*)) où *f* est d'habitude une combinaison de polynômes, logarithmes ou exponentielles. Ceci reprend la notation mathématique classique, et signifie que le nombre d'opérations effectuées est borné par *cf*(*n*), où *c* est une constante, lorsque *n* tend vers l'infini.

Classement des algorithmes selon la complexité :

Les algorithmes usuels peuvent être classés en un certain nombre de grandes classes de complexité.

* Les algorithmes sous-linéaires, dont la complexité est en général en *O*(log*n*). C'est le cas de la recherche d'un élément dans un ensemble ordonné fini de cardinal *n*.
* Les algorithmes linéaires en complexité *O*(*n*) ou en *O*(*n*log *n*) sont considérés comme rapides, comme l'évaluation de la valeur d'une expression composée de *n* symboles ou les algorithmes optimaux de tri.
* Plus lents sont les algorithmes de complexité située entre *O*(*n*2) et *O*(*n*3), c'est le cas de la multiplication des matrices et du parcours dans les graphes.
* Au delà, les algorithmes polynomiaux en *O*(*nk*) pour *k* > 3 sont considérés comme lents, sans parler des algorithmes exponentiels (dont la complexité est supérieure à tout polynôme en *n*) que l'on s'accorde à dire impraticables dès que la taille des données est supérieure à quelques dizaines d'unités.

La recherche de l'algorithme ayant la plus faible complexité, pour résoudre un problème donné, fait partie du travail régulier de l'informaticien.

**1.3.2 Calculs élémentaires de complexité**

Donnons quelques règles simples concernant ces calculs. Tout d'abord, le coût d'une suite de deux instructions est la somme des deux coûts :

*T*(*P*; *Q*) = *T*(*P*) + *T*(*Q*).

Plus généralement, si l'on réalise une itération, on somme les différents coûts :

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *T*(for(i = 0; i < n; i++) P(i); | ) =  |

|  |
| --- |
| *n*-1 |
| ∑ |
| *i*=0 |

 | *T*(*P*(*i*)). |

Si *f* et *g* sont deux fonctions positives réelles, on écrit

*f* = *O*(*g*)

**Exemple**

**Recherche du plus petit élément**

Reprenons l'exemple suivant:

Fonction  plusPetit (x : tableau d’entier) : entier

Début
   k: entier; k = 0, n : entier ; // n c’est la taille maximale du tableau x.

Pour i de 1 à n faire
          // invariant : k est l'indice du plus petit
          // élément de x[0..i-1]
        si (x[i] < x[k])
              k = i;
retourner k;
Fin .

Dans cette fonction, on exécute *n*-1 tests de comparaison. La complexité est donc *n*-1 = *O*(*n*).