Chapitre 7 : Eléments d'analyse d'algorithmes

Table des matières

1 Terminaison et correction : illustrations sur des algorithmes d'arithmétiques

1.1 Apprentissage de l'analyse d'un algorithme : division euclidienne

a) **Remarque :** une des difficultés de lecture d'un algorithme en informatique par rapport aux mathématiques est qu'au cours du déroulement d'un algorithme, une variable peut être réaffectée et prendre successivement différentes valeurs. En info., on peut penser à la *variable* comme à un *contenant* et à *la valeur* comme *un contenu*.

Convention : Dans ces notes, on notera `x` un nom de variable informatique et x la *valeur* (ou x_k les valeurs successives) stockée(s) dans la variable qui s'appelle `x`.

Cette difficulté est aussi un avantage en terme de concision... quand on a bien compris... comme on espère le montrer ci-dessous.

b) **Un algorithme pour obtenir le reste de la division euclidienne** On se donne deux entiers a et b avec $b > 0$. Le résultat de la division euclidienne est le couple (q, r) tel que $a = bq+r$ et $0 \leq r < b$. Ici, on se limite d'abord à la recherche du reste : pour cela on va utiliser seulement *deux variables informatiques* `a` et `b`. Au départ dans `a` et `b`, il y a les valeurs a et b pour lesquelles on veut calculer la division euclidienne, mais l'algorithme modifie la valeur de la variable informatique `a` et on va montrer qu'à la fin, dans `a` il y a le reste r que l'on cherche.

Voici l'algo. écrit en PYTHON où les valeurs de a et b seront donc rentrées comme *arguments* d'une fonction appelée *reste*, *volontairement non documentée* :

```
def reste(a,b):
    while (a<0) or (a>=b):
        if a<0:
            a=a+b
        if a>=b:
            a=a-b
    return a
```

Comment juger de la validité de cet algorithme ? Deux problèmes bien distincts :

- *terminaison* : on doit être sûr que l'algorithme s'arrête au bout d'un nombre fini d'étapes,
- *correction de l'algo.* : on doit être sûr que l'algorithme fait ce qu'on veut, donc ici que la valeur renvoyée est bien celle du reste.

c) **Le problème de la terminaison :**

Pour étudier l'algorithme on note $a_0 = a$, et a_k la valeur de la variable informatique `a` après la k-ième étape.

(M1) ici : on se place dans chacun des deux cas des `if` et on fait une analyse semblable à celle de la dém. mathématique.

- Si $a_0 = a \geq b$, alors $a_1 = a_0 - b$ et si on prend k le maximum des entiers tels que $a_0 - kb \geq b$ alors $a_0 - (k+1)b < b$ et $a_0 - (k+1)b \geq 0$ (idem cours de maths). Donc l'algorithme s'arrête après la $(k+1)$-ième étape.
- Si $a_0 < 0$, alors $a_1 = a_0 + b$ et si on prend k le maximum des entiers les $a_0 + kb < 0$, on sait que $a_0 + (k+1)b \geq 0$ et $a_0 + (k+1)b < b$ (solution du cas laissé en exercice dans le cours de maths).

(M2) avec un compteur qui diminue strictement : *variant de boucle*

• Souvent on montre qu'un algorithme s'arrête en montrant qu'un certain nombre diminue strictement à chaque test de la boucle `while`. Ce nombre est appelé en info. un *variant*. Souvent ce nombre est un entier naturel et une suite d'entiers naturels ne peut pas décroître strictement indéfiniment !

Ici par exemple : on peut voir qu'après chaque étape de l'algorithme la distance entre `a` et le centre de l'intervalle $[0, b]$ diminue strictement. Cette distance vaut `abs(a-b/2)` ; pour éviter la fraction on peut considérer le double : `abs(2a-b)`.

d) **Le problème de la correction :**

Pour vérifier la *correction* d'un algorithme, on exhibe souvent des *invariants* c'est-à-dire des objets qui ne *changent pas* pendant toute l'exécution de l'algorithme.

Sur notre exemple : la correction se voit directement sur le raisonnement mathématique fait au c) (M1) et la recherche d'un invariant n'est pas si évidente... *elle le devient si on regarde l'algorithme plus complet qui donne* (q, r) comme on va le faire maintenant.

e) **L'algorithme complet de division euclidienne, et l'invariant de boucle**

On se donne toujours des valeurs initiales (a, b). On pourrait n'utiliser que *trois variables* informatiques `a,b,q` pour définir la fonction PYTHON suivante, néanmoins, pour plus de lisibilité, on va créer une quatrième variable `r`

```
def quo_reste(a,b):
    """renvoie le quotient et le reste de la div. eucl. de a par b"""
    q=0
    r=a # on aurait pu travailler avec a directement
    while (r<0) or (r>=b):
        if r<0:
            r=r+b
            q=q-1
        if r>=b:
            r=r-b
            q=q+1
    return q,r
```

Comme dans l'algo. précédent, la var. `b` n'est pas modifiée, et garde toujours la valeur initiale b, ici `a` n'est pas modifiée non plus, alors que la variable locale `r` prend comme valeur finale la valeur du reste comme on l' a montré plus haut. La variable locale `q` est initialisée à 0 et à la fin la fonction retourne la valeur finale de `q` dont on va montrer que c'est la valeur q du quotient de la division euclidienne de a par b.

Ici la valeur de `bq+r` est un invariant de la boucle `while`

En effet : à chaque étape on transforme `bq+r` en `b(q-1)+(r+b)` ou bien en `b(q+1)+(r-b)`.
Quel intérêt d'un tel invariant ? A l'initialisation l'égalité `bq+r==a` est vraie avec les valeurs : $b \times 0 + a = a$. Donc à l'arrêt de l'algorithme, comme `a` et `b` n'ont pas changé de valeur et comme on a déjà vu qu'à la fin de l'algo. `r` contenait bien la valeur voulue pour le reste de la div. euclidienne, la variable `q` contient bien la valeur q telle que $a = bq + r$.

1.2 L'algorithme d'Euclide pour le p.g.c.d.

a) **Description mathématique** (donnée dans le cours de maths) : on se donne deux nombres (a, b) et on définit une suite finie strictement décroissante d'entiers naturels $(r_k)_{k\in[\![0,N]\!]}$ par $r_0 = a$, $r_1 = b$ et pour tout $k \geq 1$, tant que $r_k \neq 0$, on définit r_{k+1} comme le reste de la division euclidienne de r_{k-1} par r_k.
La terminaison de l'algorithme est claire : la suite (r_k) est une suite d'entiers naturels strictement décroissante jusqu'à 0. Par déf. r_N est le *dernier reste non nul*.

b) **Algorithme informatique :**

Maths : La suite (r_k) est une suite réc. d'ordre 2 : r_{k+1} est défini à partir de r_k et r_{k-1}.
Traduction en info. : on a besoin d'une boucle qui modifie *deux variables*.

En Python en utilisant l'opérateur `%` pour le reste de la div. eucl. :

```
def Euclide(a,b):
    b=abs(b)# pour se ramener à un b positif, ce qui ne change pas le pgcd
    while b>0:
        a,b=b,a%b
    return a
```

Remarque : Dans un langage qui ne permet pas l'affectation en couple, on aura besoin d'une variable locale tampon pour faire l'échange.

c) **La correction de l'algorithme :** *pourquoi l'algo. renvoie-t-il pgcd(a,b)* ?
La justification a été donnée en cours de maths, et en fait, sans le dire, on a introduit un :

Invariant de boucle : à chaque étape de la boucle le pgcd`(a,b)` est inchangé.

Au départ il vaut $\text{pgcd}(a, b)$ et à la fin, en notation maths, il vaut $\text{pgcd}(r_N, 0)$ avec r_N le dernier reste non nul.
Rappel : pour tout entier α, $\text{pgcd}(\alpha, 0) = \alpha$.

1.3 La version soustractive de l'algorithme d'Euclide

Considérons l'algorithme implémenté dans la fonction Python suivante :

```
def EuclideS(a,b):
    """calcul du pgcd par la méthode des soustractions successives"""
    a=abs(a)
    b=abs(b)
    while (a!=0) and (b!=0):
        if a<=b:
            b=b-a
        elif b<a:
            a=a-b
    if a==0:
        return b
    if b==0:
        return a
```

Analysons cet algorithme pour le comprendre du point de vue de la *terminaison* et de la *correction*. On entre dans la fonction des valeurs a et b pour les variables `a` et `b`.

a) **Terminaison :** La fonction prend en entrée des nombres supposés entiers, mais se ramène tout de suite à des nombre positifs. Notons (a_k) et (b_k) la suite des valeurs prises par les variables informatiques `a` et `b` au fil de l'algorithme. A chaque étape de l'algorithme, tant que $\min(a_k, b_k) \neq 0$, on a : $a_{k+1} + b_{k+1} = (a_k + b_k) - \min(a_k, b_k)$. Ainsi $0 \leq a_{k+1} + b_{k+1} < a_k + b_k$. La suite $(a_k + b_k)$ est bien une suite d'entiers naturels strictement décroissante tant que la boucle `while` se répète. Donc la boucle `while` s'arrête.

b) **Correction :** *pourquoi l'algorithme est-il correct, c'est-à-dire pourquoi fait-il ce qui est annoncé dans la documentation*? Comme précédemment, on exhibe un *invariant de boucle.*

Ici encore la valeur de pgcd`(a,b)` est un invariant de la boucle.

En effet, pour tout $(u, v) \in \mathbb{Z}^2$ pgcd(u, v) =pgcd$(u - v, v)$ =pgcd$(u, v - u)$.
Or au départ de l'algorithme pgcd`(a,b)` vaut pgcd(a, b) et à la fin de la l'algo l'une des deux variables vaut 0 et par exemple si `b=0`, pgcd`(a,0)=a` donc à la fin de l'algo. la variable `a` contient le pgcd cherché.

2 Introduction aux problèmes de coût d'algorithme : complexité

Une fois qu'on a étudié la *terminaison* et la *correction* d'un algorithme, on peut aussi étudier son *coût* (on dit aussi la complexité) i.e. en combien d'étapes il se termine et le nombre d'opérations *élémentaires* qu'il nécessite.

2.1 Un concept mathématique important : la notation $O()$

a) **Définition (maths)** Si (u_n) et (v_n) sont deux suites réelles, on dit que $u_n = O(v_n)$ (sous-entendu quand $n \to +\infty$) ssi il existe un rang $n_0 \in \mathbb{N}$ et une constante $A > 0$ tels que : $\forall\, n \geq n_0$, $|u_n| \leq A|v_n|$.

Terminologie : Si $u_n = O(v_n)$, on dit que la suite (u_n) est *dominée* par la suite (v_n).

Remarque : Si $u_n = o(v_n)$ alors a fortiori $u_n = O(v_n)$ mais la récip. est fausse.

Exemple : Si $u_n \leq 4n^2 + 120n + 14566$ A.P.C.R. alors $u_n = O(n^2)$.
D'une manière générale, $\boxed{\text{si } v_n \sim w_n \text{ et } u_n = O(v_n) \text{ alors } u_n = O(w_n).}$
Dans l'exemple précédente, $4n^2 + 120n + 14566 \sim 4n^2$ d'où la conclusion $u_n = O(n^2)$.

b) **Intérêt en informatique :** *Souvent (mais pas toujours !) en informatique, ces constantes (comme le* 4 *devant le* $4n^2$ *ci-dessus-) , n'ont pas beaucoup d'intérêt.*
Bien sûr on pourrait dire qu'un algo. en $O(n)$ est aussi en $O(n^2)$ *mais on essaiera toujours de donner la réponse la plus précise, la plus pertinente : il ne sert à rien de dire à quelqu'un qu'on va venir le voir dans moins d'un mois si on sait qu'on va venir demain..*

2.2 Illustrations simples

a) Quelle est la complexité des fonctions suivantes, en terme de nombres de multiplications `*` ? Formuler aussi le résultat avec la notation $O()$.

```
def table1(n):
    for i in range(11):
        print(i*n)
def table2(n):
    for i in range(n):
        print(i*n)
def table3(n):
    for i in range(n):
```

```
        for j in range(n):
            print(i*j,end=" ")
        print()
```

b) Mêmes questions : en terme de nombres d'additions +

```
def sommeSimple(L):
    S=0
    for i in range(len(L)):
       S=S+L[i]
    return S

 def  sommeDouble(n):
    S=0
    for i in range(1,n+1):
        for j in range(i,n+1):
            S=S+i+j
    return S
```

Essayons `sommeDouble1(10000)`. Commentaire ?

2.3 Illustration sur le problème de l'évaluation d'un polynôme

Pour se donner une fonction polynomiale $f : x \mapsto \sum_{k=0}^{n-1} a_k x^k$, il suffit de se donner la liste des a_k. On considère donc notre fonction polynomiale donnée par une telle liste L.

a) La méthode la plus naïve pour évaluer f en un point x est alors certainement :

```
def evaluer(L,x):
    S=0
    for i in range(len(L)):
        S=S+L[i]*x**i
    return S
```

Quelle est la complexité de cette fonction, par rapport à la taille `n=len(L)` des données ?

b) Méthode incontournable du point de vue informatique :

ne pas recalculer les x^i à chaque fois, mais réaffecter en multipliant par x [1]

```
def evaluer2(L,x):
    S=0
    monome=1
    for i in range(len(L)):
        S=S+L[i]*monome # A l'étape i monôme vaut x**i
        monome=monome*x # là il vaut x**(i+1)
    return S
```

Quelle est la complexité de cette fonction, par rapport à la taille `n=len(L)` des données ?

2.4 Exemples d'algorithmes déjà rencontrés à coût logarithmique

2.4.1 L'exponentiation rapide

Complexité de l'exponentiation naive ? Complexité de l'exp. rapide ?

1. pour chasser cette mauvaise habitude !

2.4.2 La recherche dichotomique

Cf. T.P. et C.R. du T.P. en question.

Les algorithmiques où l'on divise la taille de l'entrée par un certain facteur à chaque étape, ici par deux, sont logarithmiques

2.4.3 Cas de l'algorithme d'Euclide

Exercice à faire : On considère deux entiers $(a, b) \in \mathbb{N}^2$, avec $a > b$, on note $a = bq + r$ la division euclidienne de a par b. On note $r_0 = a$, $r_1 = b$ et on réécrit cette division euclidienne $r_0 = r_1 q_1 + r_2$: étape 1 de l'algo. d'Euclide.

Pour tout $k \geq 1$, tant que $r_k \neq 0$, on note r_{k+1} le reste de la division euclidienne de r_{k-1} par r_k, qu'on écrit $r_{k-1} = q_k r_k + r_{k+1}$. On note, comme dans le cours, r_N le dernier reste non nul, de sorte que la dernière étape de l'algorithme d'Euclide s'écrit : $r_{N-1} = q_N r_N + 0$.

Avec ces notations l'algorithme a exactement N étapes (numérotées par les q_i).

a) Montrer que dans la division euclidienne $a = bq + r$, on a $br \leq \frac{1}{2}ab$ et que pour tout k, $r_{k+1} r_k \leq \frac{1}{2} r_k r_{k-1}$.

Autrement dit, le produit "dividende-diviseur" est au moins divisé par 2 *à chaque étape de l'algo. d'Euclide*

b) En déduire que le nombre N d'étapes de l'algorithme d'Euclide appliqué à a et b est majoré par $\log_2(a) + \log_2(b)$.