Chapitre 4

Analyse des algorithmes.

Chapitre 4

Analyse des algorithmes.

I. Terminaison et correction.

I.1/ Variants et invariants de boucles.

Lors d'une boucle, il faut s'assurer de deux choses :

1. La boucle s'arrête.
2. La boucle réalise ce qu'on lui demande.

Pour vérifier cela on a besoin des variants de boucles pour la terminaison et des invariants de boucles pour la correction.

Définition - variant de boucle. Pour montrer qu'une boucle while (for ce n'est pas nécessaire) s'arrête, on associe au $i^{\text{ème}}$ passage dans la boucle un entier naturel k_i déduit des variables manipulées dans la boucle de telle sorte que la suite (k_i) soit strictement décroissante. Cette suite est appelée un variant de boucle. Ainsi pour montrer que (k_i) est un variant de boucle, il faut :

1. montrer que k_i est un entier **positif**,
2. montrer que la suite (k_i) est strictement décroissante.

Définition - invariant/constante de boucle. Pour prouver qu'une boucle produit ce qu'on lui demande, on a besoin d'une constante de boucle. C'est une proposition formée à l'aide des variables manipulées dans la boucle :

1. qui est vraie avant l'entrée dans la boucle,
2. qui est vraie à chaque passage dans la boucle,
3. qui permet à l'aide de la condition d'arrêt de la boucle de vérifier l'algorithme.

Pour prouver qu'une proposition est un invariant de boucle, on raisonne comme pour la récurrence en mathématique :

1. On montre que la proposition est vraie avant la boucle : l'initialisation
2. On suppose que la propriété est vraie au i^{eme} passage de la boucle, et on la montre au $(i+1)^{\text{eme}}$ passage de la boucle. Ce qui montre qu'elle est constante à chaque passage dans la boucle.
3. On utilise cette constante pour montrer le résultat.

Remarque. Lorsqu'on a recourt à de la récursivité, il faut également montrer que l'on n'appelle pas la fonction une infinité de fois et que l'algorithme réalise ce qu'on lui demande. On a aussi besoin d'un variant et d'un invariant de boucle. Dans ce cas, on remplace dans les définitions précédentes, "passage dans la boucle" par "appel de la fonction".

I.2/ Exemples.

Exercice.[1]

1. Écrire une fonction qui à partir d'un entier n calcule $n!$. On l'écrira avec une boucle puis avec de la récursivité.
2. Prouver dans chaque cas que le programme s'arrête.
3. A l'aide d'un invariant de boucle bien choisi, montrer que l'algorithme calcule ce qu'on lui demande.

Exercice.[2] Écrire une fonction qui à partir d'un réel a et d'un entier n calcule a^n. On l'écrira avec une boucle puis avec de la récursivité, puis on testera que l'algorithme s'arrête et qu'il réalise ce qu'on lui demande.

Exercice.[3] Écrire une fonction qui à partir de deux entiers a et b ($b \neq 0$) renvoie le quotient et le reste de la division euclidienne de a par b. On testera ensuite que l'algorithme s'arrête et qu'il réalise ce qu'on lui demande.

Exercice.[4] Réaliser un programme qui calcule les n premiers termes de la suite de Fibonacci définie par $F_0 = 0$ et $F_1 = 1$.

$$\forall n \in \mathbb{N}, \quad F_{n+2} = F_{n+1} + F_n$$

Prouver que votre programme se termine bien et que votre algorithme produit bien le résultat attendu à l'aide d'un invariant de boucle

---

II. Complexité.

II.1/ Différents type de complexité.

But. Créer un outil qui permet d'estimer/de comparer les besoins en temps et en espace des programmes en fonction de leurs paramètres. Étant donné la capacité toujours plus impressionnante des unités de stockage, le besoin en espace mémoire n'est un problème que dans de très rares cas. Nous nous intéresserons donc essentiellement à la complexité en temps.

Méthode.

1. On estime en général la taille des paramètres c'est-à-dire des données nécessaires à l'exécution de l'algorithme par un entier n. Par exemple :

Algorithme manipulant	Entier n choisi
Tableau, liste	longueur du tableau
Entier	nombre de chiffre de cet entier
Chaine de caractère	longueur de la chaine.

2. On se met d'accord sur une liste d'opérations élémentaires. Souvent les additions, soustractions, multiplication, division et les affectations.

3. On compte le nombre d'opérations élémentaires $T(n)$ réalisée par l'algorithme. Ainsi $T(n)$ est globalement proportionnel au temps de calcul. De plus, on prendra ici toujours le pire des cas. Il y en a d'autres comme le meilleur des cas, cas moyen). Par convention, cela signifie que :

	si... alors :	while ... :
$\boxed{T_1(n)}$ $\boxed{T_2(n)}$	$\boxed{T_1(n)}$ sinon $\boxed{T_2(n)}$	$\boxed{T_i(n)}$
$T(n) = T_1(n)+T_2(n)$	$T(n) = Max(T_1(n),T_2(n))$	$T(n) = T_1(n)+T_2(n)+...$

II.2/ Pertinence d'une estimation en "O".

La comparaison de l'efficacité des algorithmes se fait grâce à la notation "grand O" ("O" signifiant "dominé" mais ici plutôt "de l'ordre de") On regroupe les algorithmes selon leur vitesse :

Temps	Type de complexité	Ordre de grandeur des temps de calcul pour :		
		$n = 10$	$n = 1000$	$n = 10^6$
$O(1)$	constante	10ns	10ns	10ns
$O(\ln(n))$	logarithmique	10ns	30ns	60ns
$O(n)$	linéaire	100ns	$10\mu s$	10 ms
$O(n^2)$	quadratique	$1\mu s$	1ms	2.8h
$O(n^3)$	cubique	$10\mu s$	10s	316 ans
$O(n!)$	factorielle	$36\mu s$	/	/

III. Exemples de calculs de complexité.

Le but de ce paragraphe est de comparer différents algorithme pour déterminer le plus rapide lorsque n devient grand. Dans ces exemples, on se placera toujours dans le pire des cas et on comptera le nombre d'addition, de multiplication et d'affectation. On comptera un return ou un append comme une affectation.

III.1/ Coefficients binomiaux.

Comparons les complexités de ces deux programmes calculant les coefficients binomiaux. On prendra comme paramètre pour la complexité la valeur du paramètre n de l'algorithme.

```
def CoeffBin1(n,k) :
    if n==0 and k==0 :
        return(1)
    elif n==0 :
        return(0)
    else :
        return(CoeffBin1(n-1,k)+CoeffBin1(n-1,k-1))

def CoeffBin2(n,k) :
    if k<0 or k>n :
        return(0)
    else :
     Ligne = [1]
        for i in range(0, n) :
            Ligne2 = [1]
            for j in range(1, len(Ligne)) :
                Ligne2.append(Ligne[j]+Ligne[j-1])
            Ligne2.append(1)
            Ligne=Ligne2
        return(Ligne[k])
```

Complexité de `CoeffBin1`. Comme $T(n)$ est clairement plus grand que 1, on a :

$$T(n) \;=\; Max\,(1,1,T(n-1)+T(n-1)+2) \;=\; 2T(n-1)+2$$

Par récurrence immédiate, on trouve :

$$T(n) \;=\; 2^nT(0)+2+2^2+\ldots+2^n \;=\; 2^n\times 1+2\frac{2^n-1}{2-1} \;=\; 3.2^n-2$$

Ainsi

$$\boxed{T(n) \;=\; O(2^n)}$$

Complexité de `CoeffBin2`. Ici aussi $T(n)$ est plus grand que 1. On ne tient donc pas compte du premier `if`. De plus la longueur de `Ligne` augmente de 1 à chaque tour de la boucle en i. On a donc de manière évidente que dans la $2^{\text{ième}}$ boucle, `len(Ligne)` vaut $i+1$. Ainsi la complexité de la boucle en j vaut $2i$ et :

$$T(n) = (3+2\times 0)+(3+2\times 0)+\ldots+(3+2\times(n-1)) = 3n+2\frac{n(n-1)}{2}$$

Ainsi

$$\boxed{T(n) = O(n^2)}$$

Conclusion. L'algorithme 2 est meilleur car : $n^2 \underset{+\infty}{<<} 2^n$

III.2/ Recherche dans une liste triée.

On effectue la même démarche sur des algorithmes de recherche d'un élément dans un tableau trié. Le programme devra renvoyer la position dans la liste de l'élément cherché et -1 si l'élément n'est pas dans la liste. On prendra `n=len(Liste)` comme paramètre pour la complexité.

```
def Recherche1(Elt, Liste) :
    for i in range(0, len(Liste)) :
        if Elt==Liste[i] :
            return(i)
    return(-1)

def Recherche2(Elt, Liste) :
    Min = 0
    Max = len(Liste)-1
    Mil = (Min + Max)//2
    while Liste[Mil]!=Elt and Max-Min>=2 :
        if Liste[Mil]>Elt :
            Max = Mil-1
        else :
            Min = Mil+1
        Mil = (Min + Max)//2
    if Liste[Min]==Elt :
        return(Min)
    elif Liste[Max]==Elt :
        return(Max)
    else :
        return(-1)
```

Complexité de `Recherche1`. De manière immédiate, on a :

$$\boxed{T(n) = n\times 1 = O(n)}$$

Complexité de `Recherche2`. Commençons par chercher ce que devient $A_k = Max_k - Min_k$ à chaque tour de boucle. Il y a 2 cas :

$$\begin{cases} A_{k+1} & = & Max_{k+1} - Min_{k+1} & = & Mil_k - 1 - Min_k \\ A_{k+1} & = & Max_{k+1} - Min_{k+1} & = & Max_k - Mil_k - 1 \end{cases}$$

Comme le milieu est arrondi à l'inférieur, la plus grande valeur est obtenue dans le cas numéro 2. On a ainsi :

$$A_{k+1} \leq Max_k - \left[\frac{Min_k + Max_k}{2}\right] - 1 \leq Max_k - \frac{Min_k + Max_k}{2} \leq \frac{A_k}{2}$$

On obtient par récurrence immédiate :

$$A_k \leq \frac{A_0}{2^k} = \frac{n-1}{2^k} \leq \frac{n}{2^k}$$

Or la boucle s'arrête lorsque $A_k \leq 1$. Ainsi la boucle est terminée si :

$$\frac{n}{2^k} \leq 1 \iff k \geq \log_2(n)$$

On peut donc maintenant calculer la complexité :

$$\boxed{T(n) = 6 + \log_2(n) \times 2 + 1 = O(\log_2(n)) = O(\ln(n))}$$

Conclusion. L'algorithme 2 est plus rapide car : $\ln(n) \underset{+\infty}{<<} n$

III.3/ Puissance.

Enfin déterminons la complexité de 2 algorithmes calculant un réel a puissance un entier n. On prendra `n` comme paramètre pour la complexité.

```
def Puiss1(a,n):
    p=1
    for i in range(0,n) :
        p=p*a
    return(p)


def Puiss2(a,n) :
    if n==0 :
        return(1)
    elif (n%2==0) :
        return(Puiss2(a*a,n//2))
    else :
        return(Puiss2(a*a,n//2)*a)
```

Complexité de `Puiss1`.

$$\boxed{T(n) \;=\; 1 + n \times 2 + 1 \;=\; O(n)}$$

Complexité de `Puiss2`.

$$T(n) \;=\; Max\left(1, T\left(\left[\frac{n}{2}\right]\right) + 3, T\left(\left[\frac{n}{2}\right]\right) + 4\right) \;=\; T\left(\left[\frac{n}{2}\right]\right) + 4 \qquad (*)$$

Ainsi pour k dans $\mathbb{N}$, on a :

$$T\left(2^k\right) \;=\; T\left(2^{k-1}\right) + 4 \;=\; T\left(1\right) + 4k \;=\; 1 + 4k$$

On peut montrer facilement par récurrence forte, grâce à la relation $(*)$ que T est une fonction croissante. On choisit donc k tel que $2^k \leq n \leq 2^{k+1}$, on a donc $k \leq \log_2(n) \leq k+1$ et :

$$4k + 1 \;=\; T(2^k) \;\leq\; T(n) \;\leq\; T(2^{k+1}) \;=\; 4k + 5$$

Soit encore :

$$4(\log_2(n) - 1) + 1 \;\leq\; T(n) \;\leq\; 4\log_2(n)$$

Enfin

$$\boxed{T(n) \;=\; O(\ln(n))}$$

IV. Application aux algorithmes de tri.