Deug-Mias-2
Institut Galilée

Complexité des Algorithmes

A) Introduction

Ce que l'on entend par complexité des algorithmes est une évaluation du coût d'exécution d'un algorithme en termes de temps (complexité temporelle) ou d'espace mémoire (complexité spatiale).
Ce qui suit traite de la complexité temporelle, mais les mêmes notions permettent de traiter de la complexité spatiale.

Ce coût d'exécution dépend de la machine sur lequel s'exécute l'algorithme, de la traduction de l'algorithme en langage exécutable par la machine. Mais nous ferons ici abstraction de ces deux facteurs, pour nous concentrer sur le coût des actions résultant de l'exécution de l'algorithme, en fonction d'une "taille" *n* des données traitées. Ceci permet en particulier de comparer deux algorithmes traitant le même calcul. Nous verrons également que nous sommes plus intéressés par un comportement asymptotique (que se passe -t'il quand *n* tend vers l'infini?) que par un calcul exact pour n fixé. Enfin, le temps d'exécution dépend de la nature des données (par exemple un algorithme de recherche d'une valeur dans un tableau peut s'arreter des qu'il a trouvé une occurrence de cette valeur. Si la valeur se trouve toujours au début du tableau, le temps d'exécution est plus faible que si elle se trouve toujours à la fin). Nous nous intéresserons d'abord dans ce qui suit à la complexité en "pire des cas" , qui est une manière, pessimiste, d'ignorer cette dépendance (on evaluera le temps d'exécution, dans le cas évoqué ci-dessus, en supposant qu'il faut parcourir tout le tableau pour trouver la valeur cherchée).

Considérons l'exemple-1 suivant:

```
{début}
K <-- 0
I <-- 1
{#1}
TantQue I ≤ N {#2}  Faire
        R <-- R+T[I] {#3}
```

I <-- I+1{#4}

FinTantQue

{fin}

Le temps d'exécution t(n) de cet algorithme en supposant que:

- N=n
- t1 est le temps d'éxécution entre le début et {#1}
- t2 est le temps d'éxécution de la comparaison {#2}
- t3 est le temps d'éxécution de l'action {#3}
- t4 est le temps d'éxécution de l'action {#4}
- t1, t2, t3, t4 sont des constantes (c.à.d. ne dépendent pas de n)

s'écrit:

$$t(n) = t1 + \sum_{i=1}^{n} (t2 + t3 + t4) \quad + t2$$

et en définissant le temps *tit* d'éxécution d'une itération (condition comprise), on obtient:

$$tit = (t2 + t3 + t4)$$

d'où $t(n) = t1 + t2 + n \times tit$

ce qui signifie que le temps d'exécution dépend linéairement (plus précisément est une fonction affine) de la taille *n*.

Nous avons dit que nous nous intéressions au comportement asymptotique: que dire de *t(n)* quand *n* tend vers l'infini?

$$\lim_{n \to \infty} {}^{t(n)}\!/_{tit \times n} \quad = 1$$

autrement dit *t(n)* est équivalent à l'infini à *tit*n, ce qui s'écrit:*

$$t(n) \underset{\infty}{\sim} tit \times n$$

L'algorithme est donc asymptotiquement linéaire en *n*.

Dans cet exemple simple l'évaluation en "pire des cas" est immédiate puisque t(n) ne dépend pas de la nature des données, ce qui n'est pas le cas de l'exemple-2 suivant:

{début}

K <-- 0

I <-- 1

{#1}
TantQue I ≤ N *{#2}* Faire
    R <-- R+T[I] *{#3}*
    Si R>1000 *{#3'}* Alors
        R<-- 2*R *{#3''}*
    FinSi
    I <-- I+1{#4}
FinTantQue
{fin}

Ici, le pire des cas (celui qui conduit au temps d'exécution le plus grand) est celui où la condition *{#3'}* est toujoursVraie. En effet dans ce cas là R<-- 2*R *{#3''} est exécutée à chaque itération.* Ce qui correspond à l'évaluation suivante du temps d'exécution:

$$t(n) = t1 + \sum_{i=1}^{n}(t2 + t3 + t3' + t3'' + t4) \quad + t2$$

Ici encore l'algorithme est asymptotiquement linéaire en *n*..

Un usage courant est d'associer un temps constant à chaque type d'opération ou d'action élémentaire. Ainsi en notant:

- *taff* le temps correspondant à une affectation, *tp, tm, tc* les temps associés respectivement à une addition, une multiplication et une comparaison, on obtient le temps suivant:

$$(n) = 2 \times taff + \sum_{i=1}^{n}(tc + taff + tp + tc + taff + tm + taff + tp) \quad + tc$$

$$= (2 + n \times 3) \times taff \quad + (2n) \times tp \quad + n \times tm \quad + (n+1) \times tc$$

Pour simplifier encore, on confondra les temps associés à plusieurs opérations différentes (par exemple additions et multiplications seront associées au même temps *to*).
Enfin puisque ce qui nous intéresse est l'ordre de grandeur asymptotique de ce temps, on simplifiera encore en s'intéressant à une catégorie d'opérations ou d'actions. Par exemple, dans le cas ci-dessus on comptera, plutot que le temps, le nombre de telles opérations, ce qui donne, en notant *no*, le nombre d'additions et multiplications:

$$no = 3n$$

Ce procédé est surtout intéressant si on compte ainsi des opérations caractéristiques au sens intuitif suivant : le nombre de ces opérations est asymptotiquement proportionnel au temps d'exécution. Dans le cas ci-dessus, par exemple, compter les divisions n'aurait pas de sens (il n'y en a pas), alors que le nombre d'additions et multiplications est asymptotiquement linéaire en *n*,

ce qui est également le cas du temps d'exécution.
C'est un point important, en effet un algorithme linéaire est bien preférable à un algorithme quadratique (c.à.d dont le temps d'exécution est asymptotiquement proportionnel au carré de *n*) même si pour des valeurs petites de *n*, le temps réel d'exécution serait plus faible pour ce dernier. En effet, à partir d'une certaine valeur de n, l'algorithme quadratique devient beaucoup plus lent que l'algorithme linéaire. C'est cette évaluation de la tendance de l'algorithme qui nous interesse plus qu'une évaluation précise du temps d'exécution. Comparons ainsi deux algorithmes dont les temps d'exécution *ta* et *tb* seraient les suivants:
$ta(n) = 100xn$

$tb(n) = 2 \times n^2$

pour n = 50 les deux temps sont identiques, mais pour

n=100, ta = 10.000, tb =20.000

n=1000, ta = 100.000, tb = 2.000.000

n = 10.000, ta =1.000.000, tb = 200.000.000

B) Les notations *O* et Θ

Définition_1

Soient *f* et *g* deux fonctions de *N* dans $\Re^+$*,

Si il existe un réel $c > 0$ et un entier positif (un rang) n_0 tel que

Pour tout $n > n_0$, $f(n) \leq c.g(n)$

on dit que *f* est en *O(g) (f est asymptotiquement dominée par g)*

Exemple-1:

f(n) = 3n +1, g(n) = n

3n+1 est en *O(n)*

En effet pour $n_0 = 2$, et c = 4 on a bien pour $n > n_0$, l'inégalité $3n + 1 \leq 4n$

Définition_2

Soient *f* et *g* deux fonctions de *N* dans $\Re^+$*,

Si *f* est en *O(g)* et *g* est en *O(f)* alors

on dit que *f* est en Θ *(g) (f et g sont de même ordre de grandeur asymptotique).*

Exemple-2:

f(n) = 3n +1, g(n) = n
3n+1 est en Θ *(n)*
En effet d'une part, *3n+1* est en *O(n),* d'autre part pour $n_0 = 2$, et c = 2 on a bien, pour n> n_0, l'inégalité $n \leq 2(3n+1)$ et donc *n* est en *O(3n+1)*

En pratique *f* représente une quantité à étudier (temps, nombre d'opérations) et *g* fait partie d'une échelle de fonctions simples (n, $n\log_2(n)$, n^2, etc...)destinée à informer sur le comportement asymptotique de *f.*

Remarquons que, si nous reprenons l'exemple-2, il est correct d'affirmer que *3n+1* est en $O(n^2)$ (à vérifier par le lecteur*)* cependant, implicitement, on s'interesse à dominer *(3n+1)* par la plus petite fonction possible. C'est ce que permet, intuitivement, d'affirmer la notation Θ. Dans cet exemple on remarque facilement que (n^2) n'est pas en *O(3n+1),* et donc que *(3n+1)* n'est pas en Θ (n^2). En effet quelle que soit la valeur de *c,* il n'existe pas de rang à partir duquel on aurait $(n^2) \leq c(3n+1)$*:* il suffit d'essayer de résoudre $(n^2) - c3n - 1 = 0$ et d'observer que le discriminant $((c3)^2 + 4)$ est toujours positif, et donc qu'à l'extérieur des racines (c'est à dire de rangs particuliers) le polynome est > 0, ce qui exclut naturellement de trouver un rang à partir duquel on aurait $(n^2) \leq c(3n+1)$.

Propriété_1

f est en Θ *(g)* <u>si et seulement si</u>:

Il existe *c, d* réels >0 et un rang n_0 tels que, pour tout n > n_0 , on a

$d.\ g(n) \leq f(n) \leq c.g(n)$

<u>Preuve</u>: (cf Cours)

La notation Θ se ramène donc à un encadrement (à partir d'un certain rang) de la quantité f étudiée.

Propriété_2

1) si $\lim_{n\to\infty} f(n)/g(n) = a \neq 0 \quad \Rightarrow f$ est en $\Theta(g)$

2) si $\lim_{n\to\infty} f(n)/g(n) = 0 \quad \Rightarrow f$ est en $O(g)$ mais f n' est pas en $\Theta(g)$

3) Si $\lim_{n\to\infty} {f(n)}/{g(n)} = \infty \quad \Rightarrow f$ n' est pas en $O(g)$ et donc f n' est pas en $\Theta(g)$

Preuve: (cf Cours)

Cette propriété permet dans la pluspart des cas d'évaluer *f*.
En pratique on cherchera des équivalents à l'infini de f.

Exemple-3:

f(n) = 3n +1

$3n+1 \underset{\infty}{\sim} 3n$ et donc $\lim_{n\to\infty} {3n+1}/{n} = 3 \neq 0 \quad \Rightarrow (3n+1)$ est en $\Theta(n)$

Exemple-4 (deux boucles imbriquées sans dépendance des indices).

```
B<--0
I <--1
Pour I <-- 1 à N Faire
        B<--B+2
        Pour J <-- 1 à N Faire
                T[I,J] <-- (1+T[J,I] )*B
        FinPour
FinPour
```

Soit *Op(n)* le nombre d'additions et multiplications avec N=n,

$$Op(n) = \sum_{i=1}^{n} 1 + (\sum_{j=1}^{n} 2) = \sum_{i=1}^{n} 1 + 2n = n(1+2n) = 2n^2 + n$$

$Op(n) \underset{\infty}{\sim} 2n^2$

$\lim_{n\to\infty} {(2n^2+n)}/{n^2} = 2 \neq 0 \quad \Rightarrow Op(n)$ est en $\Theta(n^2)$

Exemple-5 (deux boucles imbriquées avec dépendance des indices).

```
B<--0
I <--1
Pour I <-- 1 à N Faire
        B<--B+2
        Pour J <-- 1 à i Faire
                T[I,J] <-- (1+T[J,I] )*B
        FinPour
FinPour
```

$$Op(n) = \sum_{i=1}^{n}(1+(\sum_{j=1}^{i}2)) \quad = \sum_{i=1}^{n}(1+2i) \quad = n + 2\sum_{i=1}^{n} i \quad = n + 2(\frac{n(n+1)}{2}) = n^2 + 2n$$

$$Op(n) \underset{\infty}{\sim} n^2$$

$$\lim_{n\to\infty} {(n^2+2n)}/{n^2} \quad = 1 \neq 0 \qquad \Rightarrow Op(n) \text{ est en } \Theta(n^2)$$

Remarque importante:

l'affectation, la lecture, l'écriture d'une variable de type elementaire se fait en un temps constant (indépendant de n) et est compté comme une opération. Il en va différemment de l'affectation (qui est une copie de la valeur d'une expression dans une variable) d'une variable de type non élémentaire, lorsque ce type dépend de n. Ainsi soit un tableau T de taille n, alors:

T2 <-- T correspond à n affectations élémentaires, et si on note *Aff(n)* le nombre d'affectations et d'opérations dans l'algorithme suivant:

T[1] <--T[N] -T[N-1]
TAMP <--T
T <--T2
T2 <-- TAMP

On obtient *Aff(n) =2 + 3n*

C) Règles du calcul de complexité « en pire des cas »

Nous essayons ici de fixer des règles pour aider à l'évaluation de la complexité en temps ou en nombre d'opérations d'un algorithme.

Notons $T_A(n)$ le temps ou le nombre d'opérations, « en pire des cas » correspondant à la suite d'actions A, ou au calcul de l'expression A. Une suite d'actions est considérée ici comme une action non élémentaire.

Règle-1: Enchaînement

Soient deux suites d'actions A1 et A2, et A1+A2, la suite « A1 suivi de A2 ».

Alors:

$$T_{A1+A2}(n) = T_{A1}(n) + T_{A2}(n)$$

Règle-2: Conditionnelle

Soit une action *A* de la forme « Si *C* Alors *A1* Sinon *A2* FinSi »

Alors:

$$T_A(n) = T_C(n) + Max(T_{A1}(n), T_{A2}(n))$$

En effet, dans le pire des cas, c'est toujours la plus coûteuse des deux actions qui s'exécute.

Exemple-6:

```
Pour i <--1 à N Faire
        Res<--X+Y+Z+Res
        Si T[I] +K < B Alors
                {Action1}
                Pour j <--1 à N Faire
                        Res <-- Res +T[J]
                FinPour
        Sinon
                {Action2}
                Res<--Res+T[I]
        FinSi
FinPour
```

Soit *Op(n)* le nombre d'additions « en pire des cas », avec N=n.
Notons *Opc(i,n)* le nombre d'additions dans la structure « Si.....Sinon...FinSi » à la ième itération de la boucle externe, *Op1(i,n)* le nombre d'additions dans *Action1* et *Op2(i,n)* le nombre d'additions dans *Action2* :

$$Op(n) = \sum_{i=1}^{n} 3 + Opc(i,n)$$

$$Opc(i,n) = 1 + Max(Op1(i,n), Op2(i,n)) = 1 + Max(n,1) = 1 + n$$

$$Op(n) = \sum_{i=1}^{n} 4 + n = n^2 + 4n$$

$$Op(n) \underset{\infty}{\sim} n^2 \text{ donc } Op(n) \text{ est en } \Theta(n^2)$$

En ce qui concerne les structures itératives, une règle générale pourrait être la suivante:

Règle-3: Itération (TantQue)

Soit une action *A* de la forme « TantQue *C* Faire *A1* FinTantQue»
En notant *niter(n)* le nombre d'itérations, on a:

$$T_A(n) = \sum_{i=1}^{niter(n)} (T_C(i,n) + T_{A1}(i,n)) + T_C(niter(n)+1, n)$$

Cependant cette règle est trop abstraite et ce qu'il faut en retenir est surtout le lien entre les structures répétitives et la sommation, comme nous l'avons vu precédemment. En particulier il

est bon de lier la variable indice de la sommation avec une variable « compteur » de la structure répétive lorsqu'elle existe. Il est cependant intéressant de noter que:

- La condition *C* est exécutée dans le « TantQue » une fois de plus que le « corps » A1 de la boucle.
- T_C n'est pas toujours une constante, mais peut dépendre de *n* et du rang *i* de l'itération, qui est souvent lié à une variable.
-

Exemple-7:

{Nous nous interessons ici au nombre d'opérations (+,-,), Op(n), avec N=n}*
{Nous supposons ici que Truc(l,n)<n et que Truc(l,n) nécessite Tt(n)= (n-l) opérations}
{on suppose aussi que Tab est de dimension Nmax ≥ N+1}

```
Res<--0
L<--2
TantQue  L ≤ Truc(L, N) Faire
        Res<--Res+2*Tab[L+1]+Truc(L,N)
        L<--L+2
FinTantQue
```

Remarquons dabord que L augmente de 2 à chaque itération et que dans le pire des cas la condition testée est L≤ N, pour toute valeur de L (puisque *Truc(L,N) ≤N*).

Nous pouvons alors écrire *Op(n)* de la manière suivante:

$$Op(n) = \sum_{l=2}^{n,\ \text{par pas de } 2} (n-l)+4+(n-l)+1$$

Pour se ramener à des pas de 1 on pose 2k=l, ce qui permet d'écrire:

$$Op(n) = \sum_{k=1}^{n/2} (n-2k)+4+(n-2k)+1$$

$$Op(n) = 2n^2/2 - 4\sum_{k=1}^{n/2} k + 5n/2$$

$$Op(n) = 2n^2/2 - 4\left(\frac{(n/2)(n/2+1)}{2}\right) + 5n/2$$

$$Op(n) = n^2 - (n^2+2n)/2 + 5n/2$$

$Op(n) \underset{\infty}{\sim} n^2/2$ donc $Op(n)$ est en $\Theta(n^2)$

Remarquons que, sans faire tous les calculs, on pouvait anticiper que *Op(n)* serait en $\Theta(n^2)$ en remarquant que le terme de plus haut degré serait de degré 2.

En ce qui concerne la structure « Pour ..... Faire ..... FinPour » on procèdera de la manière suivante: on considère la boucle TantQue équivalente:

```
Pour I <--ideb à ifin
        Action1
FinPour
```

est considéré équivalent à

```
I <-- ideb -1
TantQue I < ifin Faire
        I<--I+1
        Action1
FinTantQue
```

C'est à dire que l'on compte en plus des (ifin-ideb+1) itérations, (ifin-ideb+2) affectations, additions, comparaisons.

Remarquons qu'une pratique courante consiste à négliger (lorsque cela ne change pas la complexité) ces opérations implicites dans le « Pour..... », comme nous l'avons fait ci_dessus.

Exemple 8

```
Pour I <--1 à N Faire
        Res <--Res+I
FinPour
```

le nombre d'additions est ici N si on néglige ces opérations implicites, et 2N+1 si on les compte (ici ifin-ideb+1 = N).

Règle-4 Fonctions et Procédures non récursives

On évalue dabord les fonctions et procédures qui ne contiennent pas d'appels à d'autres fonctions et procédures, puis celles qui contiennent des appels aux précedentes, etc....

Exemple-9:

```
Algorithme Truc
Var     C: Booléen
                N, R0, R1,I: entier*
Procédure A (Var R:entier*)
        Var I:entier*
```

```
        debut
      Pour I <--1 à N Faire
              R <--R*I
        FinPour
      fin

Procédure B (Var R:entier*)
        Var I,J:entier*
        debut
      J <--1
        Pour I <--1 à N Faire
              A(J)
              R <--R*J
        FinPour
      fin
debut
lire(C)
lire(N)
R0 <--1
R1<--1
{debut #1}
Si C ='#' Alors
        {début ##1}
        Pour I <--1 à N Faire
              B(N)
        FinPour
        {fin ##1}
FinSi
{Fin #1}
{début #2}
A(N)
{fin #2}
fin
```

Nous calculons ici le nombre de multiplications Op(n) pour N=n.

Nous observons dabord que l'algorithme, dans le pire des cas (ici C='#')

$$Op(n) = \sum_{i=1}^{n} 1 + OpB(n) \quad + OpA(n)$$

où *OpA(n)* et *OpB(n*) représentent le le nombre de multiplications correspondant à l'exécution de

A et de B.

De plus on a:

$$OpB(n) = \sum_{i=1}^{n} 1 + OpA(n)$$

et finalement

$$OpA(n) = \sum_{i=1}^{n} 1 = n$$

le calcul se déroule alors ainsi:

$$OpB(n) = \sum_{i=1}^{n} 1 + n = n + n^2$$

$$Op(n) = \sum_{i=1}^{n} 1 + (n + n^2) \quad + n \quad = n^3 + n^2 + 2n \underset{\infty}{\sim} n^3$$ donc $Op(n)$ est en $\Theta(n^3)$

Remarque importante: lorsqu'on fait un appel, il faut en toute rigueur compter l'appel lui-même comme une opération particulière, mais aussi compter les opérations correspondant au passage de l'argument.

Plus précisément: lors d'un passage par valeur quelles sont les opérations mises en jeu? Pour chaque argument passé il faut évaluer l'argument (par exemple 1 'addition pour factorielle(n+1), et affecter cette valeur à une nouvelle variable (locale à la fonction). On néglige souvent cette dernière opération, cependant si l'argument passé est un tableau de taille N, alors l'affectation correspond à N affectations élémentaires, et ce coût n'est plus négligeable. C'est en particulier une des raisons pour lesquelles on évite souvent de passer par valeur un tableau même si sa valeur ne doit pas être modifiée par la procédure (ou fonction). En effet un passage par adresse ne correspond pas à N affectations élémentaires puisque seule l'adresse en mémoire du tableau est fournie à la procédure lors de l'appel.

Comparaison de deux algorithmes.

Considérons ici le problème suivant: nous disposons d'un tableau T de N entiers (N pair) ayant la propriété suivante: les N/2 premiers éléments de T se retrouvent dans la seconde moitié du tableau mais en ordre inverse (par exemple T=(1,3,5,7,7,5,3,1)). Nous donnons ci-dessous deux fonctions calculant la somme des éléments du tableau:

```
{Const N=  }
{Type Tabentier = Tableau [1..N] de entier}
Fonction Somme1( T:Tabentier):entier*
        Var I, R:       entier*
        debut
      I<--1
```

```
    R <--0
    TantQue I ≤ N Faire
            R <-- R+T[I]
            I <-- I+1
    FinTantQue
    retourner (R)
Fin

Fonction Somme2(T:Tabentier) : entier*
    Var I, R,M:     entier*
    debut
I<--1
    R <--0
    M<--N/2
    TantQue I ≤ M Faire
            R <-- R+T[I]
            I <-- I+1
    FinTantQue
     retourner (R+R)
Fin
```

Comptons les nombres d'additions *Op1(n)* et *Op2(n)* dans les deux fonctions:
Op1(n) = 2n, Op2(n) = 2 (n/2) +1= n+1
Si nous comparons ces deux fonctions, les nombres d'additions sont de même ordre de grandeur asymptotique (tous deux sont en $\Theta(n)$) mais on remarque également que:

$Op1(n) \underset{\infty}{\sim} n$ et $Op2(n) \underset{\infty}{\sim} n/2$

et plus précisément :

$$\lim_{n\to\infty} {op2(n)}/{op1(n)} = 1/2$$

c'est à dire qu'asymptotiquement *Somme2* nécessite deux fois moins d'opérations que *Somme1*.

Plus généralement lorsqu'on compare deux algorithmes effectuant la même tâche, on comparera surtout les ordres de grandeur asymptotique (l'un est-il quadratique et l'autre linéaire?) mais aussi plus finement, comme dans le cas ci-dessus, le rapport asymptotique des nombre d'opérations ou des temps d'exécution.

Cas des procédures et fonctions récursives.

Dans ce cas on obtient, lorsqu'on calcule un temps d'exécution ou un nombre d'opérations, des équations de récurence.

Exemple-1

Considérons le cas de n!, pour lequel nous avons écrit précedemment la fonction *Fact*, en utilisant les propriétés suivantes:

Fact(0) =1

Fact(n) = n*fact(n-1)

Si nous intéressons au temps d'exécution, nous obtenons l'équation suivante (où t0 et t1 sont des constantes):

Op(0) = t0

Op(n) = Op(n-1) + t1

Nous résolvons cette équation par substitutions successives. Une méthode pour présenter les substitutions consiste à écrire les équations de manière à ce qu'en sommant celles-ci on ait à gauche Op(n) et à droite une expression non recurrente (les termes en gras se simplifient):

{1} Op(n) = **Op(n-1**) + t1
{2} **Op(n-1)** = **Op(n-2**) + t1
{3} **Op(n-2)** = **Op(n-3)** + t1
.....
{k} **Op(n-k+1)**= **Op(n-k)** + t1
.....
{n-1} **Op(2)** =**Op(1)** + t1
{n} **Op(1)** = t0 +t1

Op(n) = t0+ n.t1 est en $\Theta(n)$.

Le cas où il y a plusieurs appels, est souvent plus difficile. Par exemple l'équation de récurrence correspondant à Fib(n) est la suivante:

Op(0)= Op(1) = t0

Op(n) = Op(n-1) + Op(n-2) + t1

Cette équation est assez difficile, mais nous reviendrons plus tard sur une majoration.

Un cas plus simple serait le suivant:

Fonction Mib(N:entier; P: entier): entier

Var Res : entier

{N≥0, P≤N}

```
debut
Si N = 0 Alors
        Res <--P
Sinon
        Res <-- Mib(N-1,P-1) + Mib(N-1,P) +P
FinSi
Mib<--Res
Fin
```

Ici nous obtenons les équations de récurrence suivantes, en remarquant que le temps d'exécution ne dépend pas de la valeur de P:

$Op(0) = t0$

$Op(n) = 2Op(n-1) + t1$

On peut utiliser la méthode précédente, mais pour obtenir les simplifications voulues on double chaque nouvelle équation:

{1} $Op(n) = 2\mathbf{Op(n-1)} + t1$

{2} $\mathbf{2\ Op(n-1)} = \mathbf{2^2\ Op(n-2)} + 2t1$

{3} $\mathbf{2^2\ Op(n-2)} = \mathbf{2^3\ Op(n-3)} + 2^2 t1$

.....

{k} $\mathbf{2^{k-1}\ Op(n-(k-1))} = \mathbf{2^k\ Op(n-k)} + 2^{k-1} t1$

.....

{n-1} $\mathbf{2^{n-2}\ Op(2)} = \mathbf{2^{n-1}\ Op(1)} + 2^{n-2} t1$

{n} $\mathbf{2^{n-1}\ Op(1)} = 2^n t0 + 2^{n-1} t1$

$\mathbf{Op(n)} = 2^n t0 + (1+2 + 2^2 + ... + 2^{n-1})t1$

$= 2^n t0 + (2^n - 1 / (2-1))\ t1$

est en $\Theta(2^n)$

C) Complexité « en moyenne».

En ce qui concerne la complexité moyenne, le point de vue adopté est probabiliste : plutôt que calculer une quantité en considérant la pire situation, c'est a dire la pire configuration des données d'entrée, on considère l' *univers* de toutes les configurations possibles, chacune associée à une probabilité, et on fait une somme pondérée, par ces probabilités, des valeurs prise par cette quantité dans les différentes configurations. C'est ce qu'on appelle l'*espérance mathématique*, ou plus communément, la *moyenne* de cette quantité selon ce modèle de probabilité. Une quantité, considérée dans le cadre probabiliste, s'appelle une *variable aléatoire.*

Dans ce qui suit nous cherchons donc à calculer un nombre d'opérations, ou un temps d'exécution, *moyen* selon un modèle de probabilité. Il faut donc préciser ce modèle lorsque l'on calcule une complexité moyenne.

Considérons par exemple un algorithme travaillant sur un tableau de *n* éléments, et tel que le nombre d'opérations effectuées *T(n)* peut être *n, 2n, 3n, 4n, 5n, 6n* selon le résultat d'un lancer de dé, et que de plus nous sommes dans un cas d'équiprobabilité, c'est à dire que chaque configuration a la même probabilité, (1 chance sur 6) , de se produire . Le nombre moyen d'opérations $T_{moy}(n)$ s'écrit alors :

$T_{moy}(n)$ = (1/6 * n)+ (1/6 * 2n)+ (1/6 * 3n) + (1/6 * 4n) + (1/6 * 5n)+(1/6 * 6n)
= 1/6 *n (1+2+3+4+5+6) = 3.5 *n

Remarquons que la somme des probabilités des configurations est égale à 1.

Plus formellement, soit un univers dénombrable *U*, une probabilité *p* définie sur *U* est une fonction des sous-ensembles de *U* (aussi appelés évènements*)* dans *{0,1}*, telle que :

1) $p(\{u\}) \geq 0$ pour tout élément *u* de l'univers.
2) $p(e_1 \cup e_2) = p(e_1)+p(e_2)$ pour toute paire d'évènements e_1, e_2 disjoints
3) $\sum_{u \in U} p(u) = 1$

On remarquera en particulier que la probabilité d'un évènement est la somme des probabilités de ses évènements élémentaires :

$$p(e) = \sum_{u \in e} p(u).$$

Une *variable aléatoire* réelle *X* est une application de *U* dans R .

L'*espérance mathématique* E(X), ou *moyenne* X_{moy} d'une variable aléatoire est définie par :

$$E(X) = \sum_{u \in U} p(u).X(u)$$

En réalité les configurations sont souvent nombreuses, mais l'important est de pouvoir les regrouper en sous-ensembles disjoints, chacun associé à la somme des probabilités des configurations qui le constituent. La réunion de ces sous-ensembles (ou évènements) doit constituer l'ensemble des configurations, et donc la somme des probabilités de ces évènements est également égale à 1.

Par exemple supposons que le nombre d'opérations n'est plus déterminé par un dé mais par le reste de la division par 6 de la somme des éléments du tableau. Chaque tableau possible correspond à une configuration, mais les évènements intéressants sont ceux de la forme « le reste

de la division par 6 est *r* » car ce sont eux qui déterminent le nombre d'opérations, ils forment une partition et sont chacun associé à une probabilité de 1/6. Le nombre moyen d'opérations est donc là encore 3.5 *n.

Plus formellement si $U = e_1 \cup e_{2\ldots} \cup e_n$ et pour toute paire d'évènements $(e_i \cap e_j)=\varnothing$, et en supposant que *X(u)* est constant dans chaque e_i :

$$E(X) = \sum_{i=1,n} p(e_i).X(e_i)$$

ou encore en notant plus simplement $x_i = X(e_i)$ et $p_i = p(e_i)$ on obtient:

$$E(X) = \sum_{i=1,n} p_i.x_i$$

Enfin lorsqu'on essaie de calculer une moyenne, on est parfois, pour poser le calcul, amené à partitionner un événement en sous-événements. Dans ce cas une propriété extrêmement importante est que la moyenne est aussi la « moyenne des moyennes ». Reprenons l'exemple précédent et supposons que cette fois le nombre d'opérations ne dépend non plus seulement de la valeur modulo 6 de la somme des éléments du tableau mais aussi de la couleur du tableau (bleu ou rouge) : lorsque le tableau est rouge il faut deux fois plus d'opération que lorsqu'il est bleu. On a alors les cas suivants : (où « (R= *valeur*) et (*C=couleur)* » est un évenement*)*

(R =1) et (C=bleu) ⇶ T(n) = n (R =1) et (C=rouge) ⇶ T(n) = 2n
(R =2) et (C=bleu) ⇶ T(n) = 2n (R =2) et (C=rouge) ⇶ T(n) = 4n
(R =3) et (C=bleu) ⇶ T(n) = 3n (R =3) et (C=rouge) ⇶ T(n) = 6n
(R =4) et (C=bleu) ⇶ T(n) = 4n (R =4) et (C=rouge) ⇶ T(n) = 8n
(R =5) et (C=bleu) ⇶ T(n) = 5n (R =5) et (C=rouge) ⇶ T(n) = 10n
(R =6) et (C=bleu) ⇶ T(n) = 6n (R =6) et (C=rouge) ⇶ T(n) = 12n

On a ci-dessus l'ensemble des cas : ces évènements sont disjoints et recouvrent l'univers des configurations (les tableaux) : ils forment une partition de *U*.
On a alors :

$$\begin{aligned} T_{moy}(n) = {} & (p((R=1) \text{ et } (C=\text{bleu})) * n) + (p((R=1) \text{ et } (C=\text{rouge})) * 2n) + \\ & (p((R=2) \text{ et } (C=\text{bleu})) * 2n) + (p((R=2) \text{ et } (C=\text{rouge})) * 4n) + \\ & (p((R=3) \text{ et } (C=\text{bleu})) * 3n) + (p((R=3) \text{ et } (C=\text{rouge})) * 6n) + \\ & (p((R=4) \text{ et } (C=\text{bleu})) * 4n) + (p((R=4) \text{ et } (C=\text{rouge})) * 8n) + \\ & (p((R=5) \text{ et } (C=\text{bleu})) * 5n) + (p((R=5) \text{ et } (C=\text{rouge})) * 10n) + \\ & (p((R=6) \text{ et } (C=\text{bleu})) * 6n) + (p((R=6) \text{ et } (C=\text{rouge})) * 12n) \end{aligned}$$

En notant *T(r,c,n)* le nombre d'opérations lorsque *R=r et C=c* (*T(n)* est conditionné par les valeurs de *R* et de *C*) chaque ligne dans l'expression ci-dessus correspond à un terme *t* de la forme :

$$p((R=r) \text{ et } (C=\text{bleu})) * T(r,\text{bleu},n) + p((R=r) \text{ et } (C=\text{rouge})) * T(r,\text{rouge},n)$$

Or ce terme peut être factorisé en définissant une probabilité *conditionnelle* :

$p((C=c)/R=r)) = p((R=r) \text{ et } (C=c)) \ / \ p(R=r)$

Ici $p((C=c)/R=r)$ est la probabilité (conditionnelle) de $(C=c)$ sachant que $(R=r)$ et correspond à la probabilité de $(C=c)$ en se restreignant à la partie de l'univers pour laquelle $R=r$ est vraie. La division par $p(R=r)$ permet alors de rétablir la propriété fondamentale des probabilités :

$$\sum_{u \in (R=r)} p(u / R=r) = 1$$

Un cas particulier est le cas *d'indépendance* des évènements $(R=r)$ et $(C=c)$ qui est précisément défini par :

$p((R=r) \text{ et } (C=c)) = \ p(R=r) * p(C=c)$

En conséquence:

$p((C=rouge)/(R=r)) = p(C=rouge)$

(le fait de se restreindre aux configurations telles que R=r ne change pas la probabilité que le tableau soit rouge, autrement dit la probabilité qu'un tableau soit rouge ne dépend pas de la somme des éléments du tableau)

et de même

$p((R=r)/(C=rouge)) = p(R=r)$

(le fait de se restreindre aux configurations telles que C=rouge ne change pas la probabilité que la somme modulo 6 des éléments du tableau soit égale à r)

Le terme *t* défini plus tôt s'écrit alors (on ne suppose pas ici l'indépendance):

$p(R=r)*p((C=bleu)/R=r)*T(r,bleu,n) + \ p(R=r)*p((C=rouge)/R=r)*T(r,rouge,n)$

$= \quad p(R=r)*(p((C=bleu)/R=r)*T(r,bleu,n) + p((C=rouge)/R=r)*T(r,rouge,n))$

Or $p(\ldots/R=r)$ représente ici une probabilité et donc le deuxième terme du produit représente le nombre d'opérations moyen lorsque $R=r$, noté ici $T_{moy}(r,n)$ (on vérifiera que ce terme correspond à la somme pondérée par les probabilités de couleur des nombres d'opérations pour $R=r$ et $C=$ *couleur*. Ici la notion de moyenne porte donc sur la couleur.

Si on revient alors à $T_{moy}(n)$ on obtient *:*

$T_{moy}(n)= p(R=1)*T_{moy}(1,n)+ p(R=2)*T_{moy}(2,n)+ \ldots + p(R=6)*T_{moy}(6,n)$

Autremement dit la moyenne de T(n) est la moyenne des « moyennes de T(n) selon la couleur ».

Ce résultat est très général : on est souvent amené à découper les évènements jusqu'à obtenir des événements (ici « *(R=r) et (C=c)* ») pour lesquels la quantité calculée prend une valeur déterminée (ici *T(n,r,c)).* Puis pour calculer les moyennes on commence par calculer les moyennes selon un découpage (ici la couleur) pour remonter ensuite en utilisant ces moyennes dans le calcul de la moyenne générale. Ce résultat est connu sous le nom de *théorème de Blackwell.*

Pour revenir à notre exemple, nous supposerons l'indépendance de la somme et de la couleur des tableaux et nous supposerons de plus que

p(C=rouge) =1/3 et *p(C=bleu)=2/3*

On obtient alors

$T_{moy}(r,n)$= *1/3 *T(r,bleu,n) + 2/3 *T(r,rouge,n))*

= *1/3*r*n + 2/3 * 2 r*n*

= *5/3 *r*n*

Et donc

$T_{moy1}(n)$ = 1/6 * 5/3 * n + 1/6 * 5/3 *2 n + … + 1/6 * 5/3 *6 n

= 5/18 * n * (1 + 2 + … +6)

= 75/18 * n = 4.1666 *n

En ce qui concerne la complexité moyenne donc, lorsqu'on exécute un algorithme on ignore la configuration des données. On a cherché précédemment a calculer cette complexité « dans le pire des cas », ici on s'intéresse à la valeur moyenne de cette complexité étant donné un certain modèle de probabilité (qui dépend de la situation rencontrée). Nous verrons ainsi que certains algorithmes dont le pire des cas est défavorable (le quicksort est en pire des cas en $O(n^2)$) sont néanmoins intéressants car leur cas moyen est favorable (quicksort est en O(nlog(n)) en moyenne) et donc si on l'exécute souvent on aura un temps d'exécution favorable.

Exemple : complexité moyenne de la recherche d'un élément dans un tableau de n éléments.

Soit l'algorithme suivant où t est un tableau de n entiers :

i**fl**n

tantQue i >=1 Et t[i] ≠ val faire

i**fl**i-1

ftq

(i vaut 0 si val n'y est pas et sinon vaut la dernière occurrence de val dans t *)*

(

On suppose que *probabilité(val est dans t) = pres,* que *val* n'est présent qu'une fois dans le tableau, s'il est présent, et que *probabilité(position de val dans t= i)* est constante si *val* est dans *t.*

Quel est dans ce modèle, le nombre moyen de comparaisons d'un élément de *t* avec *val* ?

Solution :

Nmoy(n) =

p(val n'est pas dans t) N_{moy}(n,absent) + p(val est dans t)* N_{moy}(n,présent)*

La somme des probabilités de la position de val dans t (lorsqu'il y est) = 1, donc :

p(position de val dans t= i / val est dans t) = 1/n

De même :

p(val n'est pas dans t) = 1- pres

Lorsque val n'est pas dans t, le nombre de comparaisons est toujours n :

$N_{moy}(n,absent) = n.$

Lorsque val est dans *t*, $N_{moy}(n,présent)$ s'écrit :

$$\sum_{i=1}^{n} p(\text{position de val dans } t = i \ / \text{ val est dans } t).i \quad = \sum_{i=1}^{n} 1/n \,.\, i \ = (n+1)/2$$

On obtient finalement:

Nmoy(n) = (1-pres).n + pres. (n+1)/2