Cours 1 — Complexité algorithmique

Introduction

Plan du cours

« My point of view is this : I see complexity as the intricate and exquisite interplay between computations (complexity classes) and applications (that is, problems) »

Christos H. Papadimitriou
Computational Complexity, un des livres fondateurs en théorie de la complexité algorithmique,
University of California San Diego (1994)

L'objectif de ce cours est de poser clairement le concept de **complexité en temps** d'un algorithme et d'apprendre, sur des exemples simples, à la calculer. Notamment, seront introduits :
— les **notations de Landau**, dont $\mathcal{O}$, telles qu'elles seront introduites en cours de maths dans les semaines à venir ;
— la notion de **taille** des variables d'entrée ;
— les notions de **complexité du pire cas** et **du meilleur cas** et, rapidement, la notion de **complexité moyenne** ;
— la façon de calculer une complexité soit de manière théorique, soit de manière empirique (par comptage ou par mesure temporelle.

1 Concept de complexité algorithmique

1.1 Position du problème

En algorithmique, comme en programmation, lorsque l'on réalise un algorithme ou que l'on programme un algorithme dans un langage donné sur une plateforme spécifique, plusieurs questions fondamentales se posent :

1. tout d'abord la question de la **validité de l'algorithme**. Il s'agit de :
   — la **preuve de terminaison** (l'exécution s'achève et l'algorithme se termine en un nombre fini d'étape) ;
   — la **preuve de correction** (l'algorithme réalise bien ce pourquoi il est conçu).
2. ensuite, se pose la question de l'**empreinte en mémoire** : il s'agit de déterminer comment évolue la consommation de mémoire entre le début et la fin de l'exécution du programme. Il s'agit d'une question fondamentale si l'on souhaite éviter un dépassement de la mémoire (ou *memory overflow*). Cette question est particulièrement importante dans les calculs réalisés sur des systèmes embarqués (montre, téléphone, balise,...).
3. un autre point tout aussi important est le **coût en temps de calcul** : il s'agit dans ce cas de déterminer le temps nécessaire à l'exécution d'un programme. Cette question est particulièrement importante dans des systèmes en temps réels (télécommunications, réalité virtuelle, *streaming*) ou pour des calculs extrêmes (sécurité informatique, modélisation de systèmes complexes comme les modèles de prévision du climat).

4. on pourrait ajouter à cette liste plusieurs autres questions, comme par exemple :
   — la **consommation énergétique** d'un programme sur une plateforme donnée (cela peut être crucial pour calculer l'*empreinte carbone* d'un système informatique) ;
   — le **rayonnement électromagnétique** d'un programme (cela est par exemple utilisé pour tester la sécurité de cartes bancaires — *cryptanalyse différentielle*) ;
   — la **complexité de conception** : il s'agit du temps humain mis par l'algorithmicien ou le programmeur pour concevoir l'algorithme et choisir les structures de données de la façon la plus pertinente afin de mettre en œuvre le programme (la mesure de ce paramètre se traduit parfois en millions d'euros).

Dans ce cours, nous allons nous intéresser exclusivement à la mesure du coût en temps de calcul. On parlera à ce sujet de **complexité en temps** d'un algorithme. Au fur et à mesure de l'année nous rencontrerons aussi des exemples dans lesquels se poseront les questions d'empreinte mémoire. On parlera dans ces cas-là de **complexité en espace** et nous verrons ultérieurement comment la mesurer.

1.2 Notion de complexité en temps

Concept : *étudier la **complexité en temps** (ou complexité temporelle, ou complexité) d'un algorithme consiste à évaluer le **temps d'exécution** d'un algorithme en fonction de la **taille des données en entrée**.*

❶ Commençons par étudier précisément un exemple

Dans les années 90, suite à l'invention du système de chiffrement RSA (du nom des trois mathématiciens Ronald Rivest, Adi Shamir et Leonard Adleman), l'entreprise RSA a lancé un concours international de factorisation de nombres qui sont les produits de 2 grands nombres premiers distincts — l'objectif étant de garantir la sécurité du système RSA utilisé par la plupart de nos cartes de crédit, sécurité qui repose sur la difficulté à pourvoir factoriser des nombres entiers en produits de nombres premiers.

Voici le résultat actuel du challenge RSA :

TABLE I. RSA CHALLENGE NUMBERS AND THEIR RESPECTIVE FACTORIZATION DETAILS

SL	Challenge number	Decimal digits	Year	Factoring team	Method	Compute time
1	RSA-120	120	1993	Lenstra et al	MPQS	830 MIPS-years
2	RSA-129	129	1994	Atkins et al	MPQS	5000 MIPS-years
3	RSA-130	130	1996	Lenstra et al	GNFS	1000 MIPS-years
4	RSA-140	140	1999	Montgomery et al	GNFS	2000 MIPS-years
5	RSA-155	155	1999	Montgomery et al	GNFS	8000 MIPS-years
6	RSA-160	160	2003	Franke et al	GNFS	2.7 1-GHz Pentium-years
7	RSA-576	174	2003	Franke et al	GNFS	13 1-GHz Pentium-years
8	RSA-640	193	2005	Bahr et al	GNFS	30 2.2-GHz Opteron-years
9	RSA-200	200	2005	Kleinjung et al	GNFS	75 2.2-GHz Opteron-years
10	RSA-704	212	2012	Shi Bai et al	CADO-NFS	*
11	RSA-220	220	2016	Shi Bai et al	CADO-NFS	*
12	RSA-768	232	NA	not factored	NA	NA
13	RSA-896	270	NA	not factored	NA	NA
14	RSA-1024	309	NA	not factored	NA	NA
15	RSA-1536	463	NA	not factored	NA	NA
16	RSA-2048	617	NA	not factored	NA	NA

Pour fixer les idées, le nombre RSA-120 vaut :

$$RSA-120 = 227010481295437363334259960947493668895875336466084780038173258247009162675779735389791151574049166747880487470296548479$$

et il se factorise en :

$$RSA-120 = 327414555693498015751146303749141488063642403240171463406883 \times 693342667110830181197325401899700641361965863127336680673013$$

L'obtention de la factorisation de $RSA-120$ a été obtenue en 830 millions d'instructions par seconde envoyées à un microprocesseur durant l'équivalent d'un an (MIPS-Year). Pour donner un ordre d'idée 1 MIPS-Year est l'équivalent en nombre d'opérations, du nombre d'opérations effectuées par un ordinateur de cadence 0.365 GHz durant un jour. Vu qu'un simple téléphone portable actuel contient un microprocesseur cadencé à 3GHz, il réalise donc de l'ordre de 9 MIPS-Year opérations en une journée.

Ce simple exemple montre à quel point la mesure du temps de calcul d'un algorithme, aussi difficile soit-il, dépend donc très fortement de la plateforme utilisée, de ses caractéristiques et notamment de la puissance de son micro-processeur.

Regardons cela sur un exemple. Il est possible de mesurer très concrètement en ms la durée d'exécution d'un algorithme en python. Il s'agit d'une première mesure naïve de la complexité. On utilise pour cela le package `time` qui permet d'afficher l'horloge système (en s). Mesurons par exemple le temps mis par une recherche naïve de facteur premier sur un entier n de la forme du challenge RSA précédent, soit $n = pq$ avec $p < q$ deux nombres premiers :

```
def facto(n):
    N=int(n**(1/2)) # les facteurs sont plus petit que la racine carrée de n
    p=2 # premier nombre premier
    while not(n\%p==0): # continuer tant que p ne divise pas n
        p+=1
    return p

n=<ma valeur>
t0=time.time()
facto(n)
print(time.time()-t0) # affiche le temps d'éxécution de facto(n)
```

Listing 1 – Mesure de temps de calcul de factorisation d'un entier RSA

Sur un ordinateur portable de 5 ans d'âge muni d'un microprocesseur de cadence 3GHz, j'obtiens les temps suivants :

Entrées	Temps
$n = p \times q$ où p et q sont de taille 10^5	$\approx 15ms$
$n = p \times q$ où p et q sont de taille 10^6	$\approx 200ms$
$n = p \times q$ où p et q sont de taille 10^7	$\approx 1,835s$
$n = p \times q$ où p et q sont de taille 10^8	$\approx 18.908s$

Remarque :

1. On pourra utiliser `nextprime(N)` du package `sympy` pour générer le plus petit nombre premier supérieur à N.
2. On observe que, si l'on réalise à nouveau une mesure de temps sur exactement la même entrée (deux premiers de taille 10^8), la valeur temporelle peut changer fortement (passer de moins de $20s$ à $30s$). En effet, si l'ordinateur exécute une tâche à ce moment-là, la valeur peut être entièrement changée.

Conclusion : on observe donc que le calcul du temps réel d'exécution d'un algorithme donne, certes, une indication de la complexité d'un algorithme mais, est :
- fortement **dépendant** de la machine utilisée (plateforme, OS, quantité de mémoire, langage de programmation, microprocesseur, etc) ;
- n'est **pas reproductible** entre différentes machines ni même sur la même machine à différents moments.

❷ Comment mesurer une information significative de la complexité d'un algorithme en s'affranchissant des données spécifiques à la machine ?

On a vu au paragraphe précédent que le simple calcul du temps d'exécution n'est pas une indication suffisamment précise et pratique pour mesurer la complexité d'un algorithme.

Une autre piste consiste à compter toutes les opérations réalisées par l'algorithme étudié. Prenons pour ce faire un exemple que nous avons déjà étudié : le calcul du maximum d'une liste.

```python
def mymax(L):
    res=L[0]
    n=len(L)
    for i in range(1,n):
        if L[i]>res:
            res=L[i]
    return(res)
```

Listing 2 – Maximum d'une liste

Pour chaque ligne de code i, notons c_i le coût d'éxécution en temps des opérations de la ligne :
- c_1 est le coût de l'entrée dans la procédure ;
- c_2 et c_3 sont des coûts d'affectations de variable, on peut supposer qu'ils sont égaux ($c_2 = c_3$) ;
- c_4 est le coût de la construction d'un processus itératif ;
- c_5 est le coût d'un test ;
- $c_6 = c_2 = c_3$;
- c_7 est le coût de la sortie de la procédure.

Selon que l'on considère que certaines opérations sont élémentaires ou non, on influe sur la granularité et l'estimation du coût. Ici, on peut supposer que les coûts c_1 à c_7 sont tous fixes indépendamment de l'entrée `L`. Nous pouvons alors effectuer notre calcul du coût d'exécution :

$$c_1 + 2c_2 + c_4 + n(c_5 + c_2) + c_7.$$

Autrement dit, ce coût est de la forme $An + B$ avec A et B deux constantes qui dépendent du système utilisé.

On observe donc que ce coût ne dépend que de n et de variables dépendantes du système. Nous noterons donc ce coût $C(n)$ où n est la taille de notre liste :

$$C(n) = An + B.$$

Vu que nous ne connaissons pas les paramètres A et B et que nous ne cherchons pas à les connaître, et vu que l'information qui nous intéresse et d'ordre asymptotique (quand $n \to +\infty$), nous retiendrons donc que **ce coût est affine en** n.

> Le fait que le coût total de notre algorithme soit affine en n est l'information qui nous intéresse :
> - elle est de nature asymptotique quand $n \to +\infty$;
> - elle est indépendante de la nature du système utilisé.
>
> Nous souhaitons donc retenir cette information comme indicateur de la mesure de la **complexité de notre algorithme**.

Il nous reste à préciser l'écriture de cette complexité. Pour ce faire, nous avons besoin du formalisme des notations de Landau, ce qui est l'objet du paragraphe suivant.

1.3 Précisions sur les ordres de grandeur

Les notations de Landau sont au programme du cours de mathématiques de PCSI. On va introduire ici uniquement les définitions de $\mathcal{O}$ et Θ :

Considérons (u_n) et (v_n) deux suites réelles.
- *Nous dirons que $u_n = \mathcal{O}(v_n)$ s'il il existe $M > 0$ et un rang N à partir duquel :*

$$|u_n| \leqslant M|v_n|.$$

  Autrement dit : $u_n = \mathcal{O}(v_n)$ si et seulement si (u_n) est bornée par (v_n) à multiple près pour n assez grand.
- *Nous dirons que $u_n = \Theta(v_n)$ s'il existe $M_1 > 0$ et $M_2 > 0$ et un rang N à partir duquel :*

$$M_1 v_n \leqslant u_n \leqslant M_2 v_n.$$

  Autrement dit : $u_n = \Theta(v_n)$ si et seulement si (u_n) est du même ordre que (v_n) pour n assez grand.

On se sert de ces symboles pour construire des comparaisons asymptotiques des suites. Ainsi par exemple :

$$(\ln(n))^k = \mathcal{O}(n^l) \quad \text{et} \quad n^l = \mathcal{O}(a^n) \quad \text{et} \quad a^n = \mathcal{O}(n!) \quad \text{et} \quad n! = \mathcal{O}(n^n)$$

pour tout $(k,l) \in \mathbb{N}^2$, pour tout $a > 1$. Et de même :

$$4\ln(n)^2 + 5\ln(n)^3 = \Theta(\ln(n)^3) \quad \text{et} \quad 4n^2 + n + 1 = \Theta(n^2).$$

Tout cela sera amplement revu en cours de mathématiques avec bien plus de détail qu'ici et avec toutes les démonstrations.
Remarque : on peut facilement voir que si $u_n = \Theta(v_n)$ alors $u_n = \mathcal{O}(v_n)$ mais la réciproque est fausse.

1.4 Méthodes d'écriture et de calcul de la complexité

A l'aide des notations de Landau introduites au paragraphe précédent nous allons pouvoir préciser la définition de la complexité d'un algorithme.

Considérons un algorithme dont les entrées ont une taille donnée par un paramètre entier. On notera $\mathcal{D}_n$ les entrée de taille n pour $n \in \mathbb{N}$.

- *Si le coût d'exécution de l'algorithme pour une entrée $d \in \mathcal{D}_n$ ne dépend que de n alors on appellera **complexité de notre algorithme** ce coût, noté $C(n)$.*
- *Si le coût d'exécution de l'algorithme pour une entrée $d \in \mathcal{D}_n$ est variable alors on appellera **complexité du pire cas de notre algorithme** le maximum de ces coûts pour une entrée d parcourant $\mathcal{D}_n$, noté $C_{max}(n)$.*
- *Si le coût d'exécution de l'algorithme pour une entrée $d \in \mathcal{D}_n$ est variable alors on appellera **complexité du meilleur cas de notre algorithme** le minimum de ces coûts pour une entrée d parcourant $\mathcal{D}_n$, noté $C_{min}(n)$.*
- *Si le coût d'exécution de l'algorithme pour une entrée $d \in \mathcal{D}_n$ est variable alors on appellera **complexité moyenne** la moyenne de ces coûts pour une entrée d parcourant $\mathcal{D}_n$, noté $C_{moy}(n)$.*

Ces complexités seront exprimées à l'aide des notations de Landau $\mathcal{O}(n)$ ou $\Theta(n)$. Ainsi dans l'exemple étudié précédemment la complexité vaut $C(n) = \mathcal{O}(n)$ ou de même $C(n) = \Theta(n)$.

Considérons plusieurs exemple :

1. Si la complexité est $\mathcal{O}(n)$ on dira que la complexité est **linéaire** en n. La complexité linéaire est celle nécessaire au fait de « lire » l'entrée. Par exemple

```
def lecture(L):
     n=len(L)
     res=[]
     for i in range(n):
         res=res.append(L[i])
     return(res)

```

Listing 3 – Complexité linéaire

   est de complexité linéaire.

2. Si la complexité est $\mathcal{O}(1)$ on dira que la complexité est **constante**. Par exemple

```
def constant(L):
    return(L[0])

```

Listing 4 – Complexité constante

est de complexité constante. Un algorithme de complexité constante est donc plus rapide que le simple fait de « lire » la variable d'entrée. En général, il s'agit d'un algorithme « d'oubli » (i.e. où l'on abandonne l'information sur une partie de l'entrée — ici on abandonne l'information sur toutes les entrées de L sauf la première, on ne prend donc même pas le temps de « lire » la liste L).

3. Si la complexité est $\mathcal{O}(n^2)$ on dira que la complexité est **quadratique** en n, si elle est $\mathcal{O}(n^k)$ pour $k \in \mathbb{N}$ et $k \geqslant 3$ alors on dira qu'elle est **polynomiale** en n et si elle est $\mathcal{O}(a^n)$ pour $a > 1$ alors on dira qu'elle est **exponentielle** en n.

Remarque : attention, quand nous étudierons les représentations des entiers, nous verrons que la taille utiliser pour représenter un entier est son logarithme, soit $\ln(n)$. Ainsi un algorithme de complexité $\mathcal{O}(n)$ sera un algorithme exponentiel (en la taille $\ln(n)$) car $n = e^{\ln(n)}$. Il est donc capital d'identifier avec précision la taille des variables avant de se lancer dans un calcul de complexité. Nous reviendrons sur ce point plus tard.

2 Exemples de calculs de complexité

2.1 Calculs de sommes

Considérons un exemple de calcul de somme comme par exemple :

$$\sum_{i=1}^{n} i.$$

Nous avons déjà vu qu'une telle somme peut se calculer par le code suivant :

```
def somme1(n):
     res=0
     for i in range(n):
         res=res+i
     return(res)

```

Listing 5 – Somme d'entiers 1

Ici on va considérer comme taille de la variable d'entrée, elle-même à savoir n, et dans ce cas la complexité est linéaire $O(n)$.

Mais on notera que si on avait décidé de calculer cette somme via la formule

$$\frac{n(n+1)}{2}$$

alors le code aurait été :

```
def somme2(n):
     return(n*(n+1)/2)

```

Listing 6 – Somme d'entiers 2

Dans ce cas, la complexité aurait été constante $O(1)$.

2.2 Avec des listes

Considérons l'algorithme de la recherche de deux valeurs les plus proches dans un tableau que nous avons déjà travaillé en TD :

```
def rechercheproche( L ):
  N =len( L )
  i , j = 0 , 1
  for p in range (N -1):
     for q in range ( p +1 , N ):
```

```
        if abs ( L [ p ] - L [ q ]) < abs ( L [ i ] - L [ j ]):
            i , j = p , q
    return [i , j ]
```

Listing 7 – Valeurs les plus proches

Ici on va considérer comme taille de la variable d'entrée, la longueur N de la liste. La complexité est donc donnée par le nombre et la longueur des boucles effectuées, que nous noterons $C(N)$. La quantité $C(N)$ s'obtient en ajoutant 1 à chaque itération de la fonction — ce qui explique qu'elle ne dépende que de `N`. Autrement dit $C(N)$ s'écrit :

$$C(N) = \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} 1.$$

On notera que, quel que soit deux entiers $k \leq l$, on a $\sum_{j=k}^{l} 1 = l - k + 1$, de plus on connait la formule, démontrée en cours de maths :

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2} \quad \forall n \in \mathbb{N}.$$

Ainsi :

$$\begin{aligned} C(N) &= \sum_{i=0}^{N-2} N - 1 - i \\ &= (N-1)^2 - \sum_{i=1}^{N-2} i \\ &= (N-1)^2 - \frac{(N-2)(N-1)}{2} \end{aligned}$$

Il s'agit bien d'un polynôme de degré 2 en N. En conclusion :

$$C(N) = \mathcal{O}(N^2.)$$

2.3 Avec des chaines de caractères

Considérons l'exemple que nous avons déjà traité en TD de la recherche d'un motif dans une chaine de caractère.

```
def recherchemotif(text,motif):
    T=len(text) # longueur de text
    M=len(motif) # longueur de motif
    res=[] # conteneur des résultats
    p=0 # compteur du balayge sur text
    while p+M<=T: # le compteur p + la longueur de motif ne doit pas excéder la
    longueur de text
        i=0
        while (i<M and motif[i] == text[p+i]): # balayage des M caractères de motif
    après la position p
            i+=1
        if i==M: # les M caractères en partant de p coincident entre text et motif
            res.append(p) # on stocke le compteur
        p+=1
    return res
```

Listing 8 – Recherche de motif

Ici on va considérer comme taille de la variable d'entrée, la longueur N de la chaine de caractère `text` (celle de `motif` étant inférieure). La complexité est donc donnée par le nombre et la longueur des boucles effectuées. Il y a exactement deux boucles mais la deuxième n'est pas de longueur constante. Le coût d'exécution n'est donc pas constant pour un `text` quelconque de longueur N. Nous allons donc évaluer la complexité du pire cas. Celui-ci consiste en ce que les deux boucles

soient de longueur maximale. Cela a lieu pour un `text` de longueur $2N$ et un `motif` de longueur N et les deux chaines de caractère ne possédant que la même lettre. Dans ce cas :

$$C_{\max}(2N) = \sum_{p=0}^{N-1} \sum_{i=0}^{N-1} 1 = N^2.$$

Donc le pire cas est de complexité quadratique.
On notera que le meilleur cas est de complexité linéaire.

3 Calcul de complexité explicite en python sur des exemples

Enfin, on notera qu'il existe une méthode très pratique pour calculer la complexité sur des exemples directement en python. Il s'agit de fixer un compteur qui est une variable globale et qui s'incrémente en temps voulu, par exemple à chaque passage de boucle ou à chaque test réalisé.

Pour appliquer cette méthode à l'algorithme précédent, on procèdera comme suit :

```
def recherchemotifC(text,motif):
    global C
    T=len(text)
    M=len(motif)
    res=[]
    p=0
    while p+M<=T:
        i=0
        C+=1
        while (i<M and motif[i] == text[p+i]):
            C+=1
            i+=1
        if i==M:
            res.append(p)
        p+=1
    return res

C=0
recherchemotifC('Bonjour','o')
print(C)
```

Listing 9 – Recherche de motif

On obtient $C = 9$. Ainsi dans ce cas, le coût d'éxécution significatif vaut 9. Cette méthode ne fournit évidemment pas le caractère asymptotique et ne fournit pas la complexité mais donne une information plus précise que le simple calcul du temps d'exécution de l'algorithme.