Complexité des algorithmes

MPSI - Prytanée National Militaire

Pascal Delahaye

4 avril 2019

L'analyse de la complexité d'un algorithme consiste à évaluer :

1. Le temps d'exécution (complexité temporelle)
2. La place mémoire nécessaire (complexité spatiale)

Ces deux quantités dépendent en partie, de la taille des données à traiter, souvent exprimée à l'aide d'un entier n. Ce nombre n peut être par exemple, le nombre d'éléments d'un tableau, le nombre de bits d'un entier, l'indice du terme u_n d'une suite définie par une relation de récurrence ... etc ...

L'objectif de ce chapitre est de fournir des critères pour comparer les performances respectives de plusieurs algorithmes remplissant la même fonction.

Complexité spatiale : Il fut une époque où la complexité spaciale était le critère principal (en 1970, un lecteur MP3 de 256 Mo occupait plusieurs armoires et coûtait aussi cher qu'une voiture de luxe). De nos jours, la place mémoire est devenue un critère secondaire et nous ne l'aborderons que brièvement dans le cas des programmes récursifs.

Complexité temporelle : Un calcul exact du temps de calcul est très compliqué et dépend de nombreux paramètres (nombre et type des opérations effectuées, ordinateur, compilateur, réseau). Un algorithme étant indépendant de ces considérations, nous évaluerons sa complexité temporelle en comptant le nombre d'exécutions $C(n)$ d'une opération jugée significative, comme par exemple, le nombre de multiplications (les opérations $+$ et $-$ sont en moyenne 100 fois plus rapides que les opérations $*$ et $/$), le nombre de comparaisons pour le tri d'un tableau, le nombre de boucles effectuées. Enfin, puisque la performance des algorithmes est en général équivalente pour des petites valeurs du nombre n, on ne s'intéressera qu'au cas où n est très grand.

Remarque 1. Cependant, selon l'utilisation prévue pour un programme, on ne cherchera pas systématiquement à optimiser la performance de l'algorithme.

1. Un algorithme de tri portant sur quelques dizaines de données sera rapide quelque soit l'algorithme choisi.
2. En revanche, s'il s'agit de trier les clients d'une société d'assurance (plusieurs centaines de milliers), un algorithme performant sera indispensable

Remarque 2. La méthode d'évaluation de la complexité temporelle est assez grossière :

1. D'une part parce que les différentes opérations intervenant dans un algorithme peuvent avoir des durées très variables.

2. D'autre part parce que l'exécution d'un algorithme où les opérations sont très rapides peut être ralentie par les accès à la mémoire (Goulet de Von Neumann).

En fait, selon les besoins de l'utilisateur, on peut définir 3 types de complexité.

On note :
- D_n l'ensemble des arguments de taille $n \in \mathbb{N}$ donnée.
- $C(d_n)$ le *coût* de l'algorithme pour un argument $d_n \in D_n$. (évalué en nombre d'opérations significatives effectuées)
- $p(d_n)$ la probabilité d'apparition de l'argument d parmi tous les arguments de taille n.

Notons alors :

1. La complexité dans le pire des cas : $C_{max}(n) = \max(C(d_n) \mid d_n \in D_n)$ Centrale nucléaire
2. La complexité dans le meilleur des cas : $C_{min}(n) = \min(C(d_n) \mid d_n \in D_n)$ Peu utilisée
3. La complexité moyenne : $C_{moy}(n) = \sum_{d_n \in D_n} p(d_n).C(d_n)$ Moteur de recherche web

Remarque 3. La complexité dans le meilleur des cas présente peu d'intérêt et la complexité moyenne n'est pas au programme. On s'intéressera donc essentiellement à la complexité dans le pire des cas.

1 Les types de complexité

1.1 La notation "O"

DÉFINITION 1 : **Notation "O"**
Soient deux fonctions $f,\ g : \mathbb{N} \mapsto \boxed{\mathbb{R}^{+*}}$.
On dit que "f est dominée par g" lorsque $\exists C > 0$ tel que : $\forall n \in \mathbb{N},\ 0 \leq f(n) \leq C.g(n)$.

Ce revient à dire que la fonction $n \mapsto \dfrac{f(n)}{g(n)}$ est bornée sur $\mathbb{N}$.

On notera alors $f = O(g)$ et on dira que f est un "grand O" de g.

Remarque 4. Attention car l'expression "est dominé par" peut prêter à confusion.
Que pensez-vous de l'affirmation : "$10^{100}n$ est dominé par $10^{-100}n$" ?

DÉFINITION 2 : **Notation "$\sim$"**
Soient deux fonctions $f,\ g : \mathbb{N} \mapsto \boxed{\mathbb{R}^{+*}}$.
On dit que "f est équivalente à g" lorsque $\forall \varepsilon > 0$ tel que $\exists n_0 \in \mathbb{N}$, tel que : $\forall n \geq n_0,\ |f(n) - g(n)| \leq \varepsilon.g(n)$.

Ce revient à dire que la fonction $\dfrac{f(n)}{g(n)} \xrightarrow[n \to +\infty]{} 1$.

On notera alors $f \sim g$.

L'objet de cette partie est de comparer le côut $C(n)$ d'un algorithme aux fonctions usuelles.

Remarque 5. On recherchera $C(n)$ sous la forme d'un équivalent ou plus souvent sous la forme d'un O.

1.2 Les types de complexité

Ce tableau donne les types de complexité usuellement rencontrés :

Type	Définition
Complexité logarithmique	$C(n) = O(\ln n)$ ou $C(n) = O(\ln^k n)$ avec $k > 1$
Complexité quasi-linéaire	$C(n) = O(n \ln n)$
Complexité linéaire	$C(n) = O(n)$
Complexité polynomiale	$C(n) = O(n^k)$ avec $k > 1$
Complexité quadratique	$C(n) = O(n^2)$
Complexité exponentielle	$C(n) = O(a^n)$ avec $a > 1$
Complexité hyperexponentielle	$C(n)/a^n \to +\infty,\ \forall a > 1$ ex : $C(n) = n!$

Remarque 6. **Comparaison des temps de calcul pour** $n = 1000$

	$\ln n$	n	$n \ln n$	n^2
Nombre d'opérations		1000		
Temps de calcul 1		10^{-7} s		
Temps de calcul 2		10^{-2} s		
Temps de calcul 3		4 s		

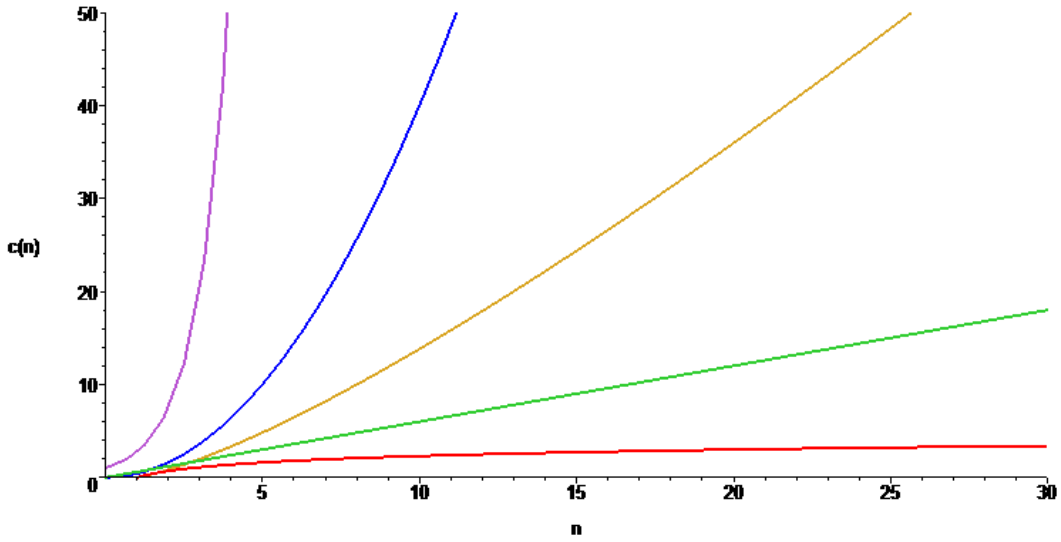


FIGURE 1 – Comparaison des types de complexité

Remarque 7. Attention à ne pas choisir trop vite!!
Supposons que nous disposions de deux algorithmes de coût $\begin{cases} C_1(n) = O(\ln n) \\ C_2(n) = O(n \ln n) \end{cases}$ pour résoudre un même problème.
On aurait tendance à choisir le premier au détriment du second ...

Confirmerez-vous votre choix si vous apprenez que $\begin{cases} C_1(n) = 10^6.\ln n \\ C_2(n) = 10^{-3} n \ln n \end{cases}$?

Remarque 8. En pratique, un algorithme de complexité quasi-linéaire a un comportement très proche d'un algorithme de complexité linéaire. En effet, la taille des données n ne dépassera jamais 10^{15} et on aura donc $n \leq n \ln n \leq 35.n$.

Plan d'étude de la complexité d'un algorithme :

1. Il faut définir l'entier n représentant la taille des arguments.
   On prendra par exemple :
   1. $n = n$ si l'argument est un entier n
   2. $n =$ la longueur de L si l'argument est un tableau de longueur n
   3. $n =$ ce que propose l'énoncé si l'argument est autre chose ... (voir exemples ...)
2. Il faut définir l'unité de coût :
   On prendra par exemple :
   1. le nombre de multiplications si la multiplication est l'opération dominante
   2. le nombre d'itérations si le nombre d'opérations et de tests sont constants par itération
   3. le nombre de comparaisons si aucune autre opération importante n'est effectuée
3. Enfin, le plus difficile reste à faire : calculer $C(n)$! !...
   Dans les cas difficiles, vous serez guidés par l'énoncé.

2 Calculs de complexité

Nous allons voir dans ce chapitre des exemples illustrant les différents types de complexité vus précédemment.

2.1 Complexité linéaire

`Exemple 1.` **La fonction factorielle**
Déterminer la complexité des fonctions récursives et itératives donnant la valeur de $n!$.

2.2 Complexité polynomiale

`Exemple 2.` Etudier la complexité de la résolution d'un système de cramer $n \times n$ par la méthode du pivot de Gauss.

2.3 Complexité exponentielle

`Exemple 3.` **La suite de fibonacci**

1. Algorithme 1 :
   Déterminer la complexité de la fonction récursive donnant la valeur du terme u_n de la suite de Fibonacci.

2. Algorithme 2 :
   Déterminer la complexité de la fonction itérative donnant la valeur du terme u_n de la suite de Fibonacci.

3. Algorithme 3 :
   Déterminer la complexité de la fonction itérative donnant la valeur du terme u_n de la suite de Fibonacci et utilisant l'expression de $u(n)$ en fonction de n.

4. Algorithme 4 :
   Que dire de la complexité de l'algorithme suivant, donnant lui aussi le terme u_n de la suite de fibonacci ?

OCaml
```
let fib2 n = let rec f = function
                         |0 -> (1,0)
                         |z -> let (a,b) = f(z-1) in (a + b, a)
             in fst (f (n-1));;
```

5. Algorithme 5 :
   On admettra que le terme général de la suite de fibonacci vérifie la relation suivante :

$$u_{p+q} = u_{p+1}u_q + u_p u_{q-1} \quad \forall (p,\ q) \in \mathbb{N}^* \times \mathbb{N}$$

   (a) Exprimez u_{2m}, u_{2m+1} et u_{2m+2} en fonction de u_m et u_{m+1}.

(b) Déduisez-en une procédure récursive `fibolog` telle que `fibolog n` calcule le couple $(u_n,\ u_{n+1})$.
Vous pourrez distinguer deux cas dans votre fonction suivant la parité de n.

(c) Evaluez l'ordre de grandeur du nombre d'appels récursifs effectués par cette fonction lorsque $n = 2^m$.
Pour certaines valeurs de n, Caml donne un résultat négatif, pourquoi ?

2.4 Complexité logarithmique

`Exemple 4.` $(*)$ Déterminer la complexité de l'algorithme permettant de déterminer le nombre de chiffres d'un entier naturel à l'aide de divisions par 10 successives.

`Exemple 5.` **L'algorithme d'Euclide**

Le but de cet exercice est d'évaluer la complexité de la fonction récursive donnant la valeur du PGCD de a et b (avec $a \leq b$) par l'algorithme d'Euclide. On considère le nombre de division euclidienne comme indicateur de la complexité.
On considére a et b strictement positifs avec $b \leq a$ et on prend $n = a$ comme mesure de la taille des données.

1. Montrer que l'on a toujours $a \mod b \leq \frac{a}{2}$. (on traitera deux cas !)

2. Montrer que $c(a) = 2 + c(a \mod b)$.
En faisant l'hypothèse réaliste que la fonction "c" est croissante, on en déduit que $c(n) \leq\leq c(\frac{n}{2}) + 2$.
En considérant que c vérifie $c(n) = c(\frac{n}{2})+2$*, on obtient alors une estimation majorée de la complexité souhaitée.*

3. En déduire que $c(n) = O(\log_2 n)$.

3 Exemple 1 : Recherche linéaire dans un tableau (Complexité Moyenne)

Le principe de la recherche linéaire est le suivant :
On teste un tableau (liste ou tableau), composante par composante pour savoir si un élément e en fait partie.

Exercice : 1

Soit l'algorithme suivant permettant de déterminer si une valeur e se trouve dans un tableau t de longueur n.

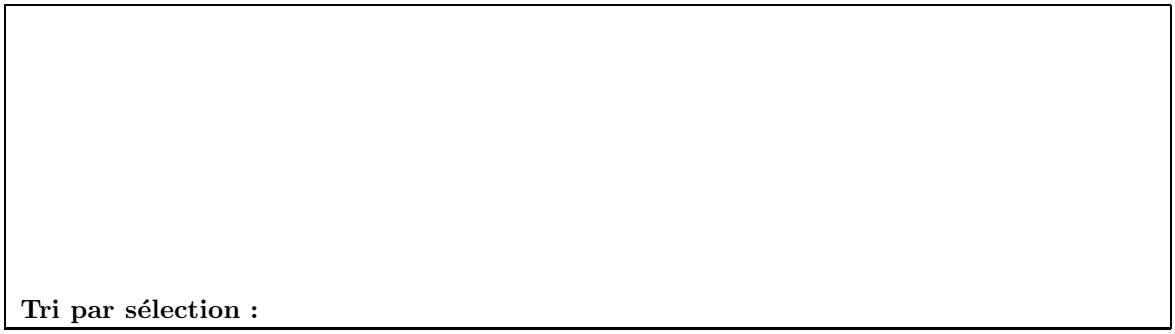
OCaml
```
let cherche e t = let trouve = ref 0 and compteur = ref 0 in
                  while (!trouve = 0) & (!compteur < Array.length t) do
                        if t.(!compteur) = e then trouve := 1
                                             else compteur := !compteur + 1
                                                              done;
                  if !trouve = 1 then true else false;;
```

Nous prendrons comme taille des données la longueur n du tableau V et comme unité de coût la comparaison d'une composante du tableau V à l'élément e et nous supposerons que chaque composante prend ses valeurs de façon équiprobable dans l'intervalle $[\![1, p]\!]$.

1. Quelle est la complexité de l'algorithme dans le pire des cas ?
2. Estimation de la complexité moyenne :
   (a) Calculer le nombre de tableaux dont le coût est égal à k avec $k \in [\![1, n]\!]$.
   (b) En déduire que le coût moyen de l'algorithme est donné par la formule $C_{moy}(n) = p - \dfrac{(p-1)^n}{p^{n-1}}$.
   (c) Que pouvez-vous en déduire ?

4 Exemple 2 : le tri par sélection

Il s'agit sans doute de la méthode de tri la plus élémentaire.
Le tri par sélection d'une liste L à n éléments consiste à échanger le plus petit élément du tableau avec son premier élément, puis à recommencer avec le sous-tableau formé des éléments d'indice $k \in [\![2, n]\!]$.



Partie I : Approche récursive

1. Déterminer une fonction récursive `minimum_et_reste` donnant à partir d'une liste L le couple formé par le plus petit élément de cette liste et la liste des éléments restants.
   Déterminer le type de complexité temporelle de cette fonction.
2. En déduire une fonction `tri_selection` permettant de trier une liste L.
3. Etudier le type de complexité temporelle de cette fonction de tri.

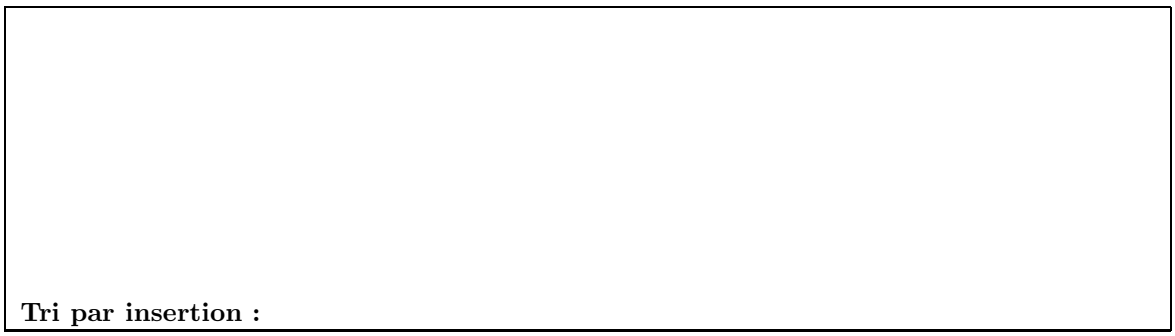
Partie II : Approche itérative

Proposer une programmation itérative du tri par sélection sur les composantes d'un tableau.
On pourra écrire les fonctions auxiliaires suivantes :

1. `echange i j v` qui échange les composantes i et j d'un tableau v
2. `minimum a b v` qui détermine l'indice du minimum d'un tableau v entre les indices a et b.

5 Exemple 3 : le tri par insertion

C'est le tri du joueur de cartes.
On considère que les éléments de la liste à trier sont donnés l'un après l'autre.
Le premier élément constitue une liste triée de longueur 1. On insère alors à sa bonne place le deuxième élément, puis le troisième ... etc ...



Partie I : Approche récursive

1. Construction de la fonction :
   (a) Commencez par écrire la fonction `insere` qui insère un élément "e" à sa place dans une liste déjà triée "L".
   (b) En déduire une programmation récursive du tri par insertion.

   Pour l'analyse de complexité, nous noterons n la longueur de la liste à traiter et nous prendrons comme unité de coût la *comparaison* menant à un appel récursif.

2. Donner la complexité de cet algorithme dans le meilleur et dans le pire des cas.

3. Partie hors-Programme : Analyse de la complexité en moyenne :
   Nous supposerons que les éléments qui composent la liste de longueur n sont les entiers de 1 à n.
   A une telle liste L on peut associer l'unique permutation σ de $[\![1, n]\!]$ définie par $\sigma(i) = j$ si l'élément j est à la i ième position. On note S_n l'ensemble des permutations de $[\![1, n]\!]$.

   (a) Pour une permutation σ donnée (et donc pour sa liste associée), une inversion est un couple $(i_1,\ i_2)$ tel que $i_1 < i_2$ et $\sigma(i_1) > \sigma(i_2)$. Combien d'inversions comporte la liste $[8; 1; 5; 10; 4; 2; 6; 7; 9; 3]$?

   (b) En raisonnant en partant de la queue de la liste, montrer que pour une liste L_σ associée à une permutation $\sigma \in \mathfrak{S}_n$, on a :

   $$c(L_\sigma) = \text{nbr inversions}(\sigma) + (n-1) \quad \text{puis} \quad c_{\text{moy}(n)} = \text{nbr moyen d'inversions par permutation} + (n-1)$$

   (c) Estimation du nombre moyen d'inversions d'une liste de longueur n.
   A une permutation $\sigma = [\sigma(1); \sigma(2); \ldots; \sigma(n)]$ on peut associer la permutation miroir $\bar{\sigma} = [\sigma(n); \sigma(n-1); \ldots; \sigma(1)]$. En remarquant que $(i_1,\ i_2)$ est une inversion de σ ssi $(n-(i_2-1),\ n-(i_1-1))$ n'est pas une inversion de la permutation miroir :
      i. Justifier que $N(\sigma) + N(\bar{\sigma}) = \dfrac{n(n-1)}{2}$, où $N(\sigma)$ désigne le nombre de permutation de σ et $\bar{\sigma}$ désigne la permutation miroir de σ.
      ii. En déduire le nombre moyen d'inversions d'une permutation.

   (d) En déduire un équivalent de la complexité moyenne $c(n)$ (en terme de nombre de comparaisons) du tri par insertion. Comparer la valeur obtenue avec la complexité du tri par sélection

Partie II : Approche itérative

Proposer une programmation itérative du tri par insertion sur les composantes d'un tableau.
Pour cela, vous pourrez commencer par construire une fonction `insere v i` qui insère la composante $v.(i)$ du tableau v dans le tableau trié $[|v.(i+1); \ldots; v.(N-1)|]$ en décalant les composantes de une unité vers la gauche (N correspond à la longueur du tableau v).

6 Exemple 4 : le tri à bulle

Tri à bulle :

Le principe, sur le modèle d'une bulle d'air remontant à la surface de l'eau, est le suivant :
On parcourt le tableau et on échange deux éléments consécutifs s'ils ne sont pas dans le bon ordre. Ainsi, à la fin du

parcours, le plus grand élément est en dernière position. Il s'agit alors de ré-itérer l'opération sur le tableau privé du dernier élément.

1. **Programme itératif :**
   Construire un programme itératif effectuant le tri à bulle.

2. **Programme récursif :**
   Pour des raisons pratiques liées à la manipulation des listes, nous procèderons ici à l'envers, en ramenant en première position le plus petit élément.
   (a) Construire une fonction récursive `bulle : int list -> int list` qui place le plus petit élément d'une liste en première position.
   (b) En déduire une implémentation récursive du tri à bulle.

3. **Etude de la complexité :**
   (a) En prenant le nombre d'échanges comme opération significative :
      i. Déterminer la complexité dans le pire des cas.
      ii. A l'aide du modèle des permutations (voir l'exemple du tri par insertion) déterminer la complexité moyenne.

   (b) En prenant le nombre de comparaisons comme opération significative :

   (c) Commentaires...

7 Etude théorique des récurrences linéaires

Il s'agit ici d'étudier les suites $(c(n))$ vérifiant une relation de la forme :

$$c(n+1) = a.c(n) + f(n) \quad \text{où} \quad \begin{cases} a \in \mathbb{N}^* \\ f : \mathbb{N}^* \mapsto \mathbb{N}^* \end{cases}$$

On rencontre cette situation en particulier dans l'étude de la complexité des algorithmes suivants :

1. Le tri par sélection d'un tableau à n éléments : $c(n+1) = c(n) + \lambda.n + \mu$
2. Les tours de Hanoï où le nombre minimal de manipulations pour déplacer une tour de n rondelles vérifie : $c(n+1) = 2.c(n) + 1$
3. Plus généralement, dans les algorithmes récursifs comprenant $''a''$ appels récursifs portant sur un argument de taille $n-1$.

PROPOSITION 1 : La suite $(c(n))$ est strictement croissante.

Preuve 1 : On commence par justifier que $c(n) \geq 0$ par une récurrence simple ...

PROPOSITION 2 : **Cas où :** $a = 1$
Dans ce cas, on a :

$$c(n) = c(1) + \sum_{k=1}^{n-1} f(k)$$

Preuve 2 : Principe des sommes télescopiques ...

Remarque 9. Dans la plupart des cas, on aura $f(n) = O(n^\alpha)$, et donc $c(n) = O(n^{\alpha+1})$.

PROPOSITION 3 : **Cas où :** $a \geq 2$
En effectuant le changement de variables $u_n = \dfrac{c(n)}{a^n}$, on obtient :

$$\frac{c(n)}{a^n} = \frac{c(1)}{a} + \frac{1}{a} \sum_{k=1}^{n-1} \frac{f(k)}{a^k}$$

Preuve 3 : On est ramené au cas précédent.

Remarque 10. Dans le cas où $a \geq 2$, le comportement de la suite $(c(n))$ dépend donc de la nature de la série $\sum \frac{f(k)}{a^k}$. En tout état de cause, la complexité sera au moins exponentielle. Plus précisément :

1. Si la série converge, alors $c(n) = O(a^n)$ (tours de hanoï)
2. Si $f(n) = O(a^n)$, alors $c(n) = O(na^n)$
3. Si $f(n) = O(b^n)$ avec $b > a$, alors $c(n) = O(b^n)$

`Exemple 6.` Que pensez-vous de l'algorithme suivant ?

OCaml
```
let rec minim = function
                | [x] -> x
                | x::q -> if x < minim q then x
                                         else minim q;;
```