7. Introduction à la complexité

L'objet d'étude de ce chapitre est la *vitesse*, ou encore l'*efficacité* d'un algorithme : on s'intéresse aux ressources mobilisées par l'exécution d'un algorithme pour effectuer ses opérations, en fonction de son entrée et de son contexte d'exécution. Les principales ressources à considérer sont :

— le temps et l'espace mémoire : c'est celles que nous étudierons dans ce chapitre,
— l'énergie consommée : elle prend une importance de plus en plus grande à mesure que des programmes de plus en plus complexes et critiques sont exécutés sur des objets de plus en plus petits et autonomes, en plus des enjeux écologiques liés à la consommation d'électricité,
— la quantité de données échangées sur le réseau, surtout dans des contextes où l'algorithme distribue énormément ses calculs au point de nécessiter beaucoup de machines différentes. Dans le même ordre d'idée, on peut considérer le nombre / la quantité d'accès mémoire nécessaires à l'exécution de l'algorithme.

Ensuite, le parti de ce chapitre (et du programme) est d'étudier de manière *asymptotique* l'évolution de la complexité en fonction de la taille de l'entrée de l'algorithme : on ignore donc à la fois son efficacité sur des *petites* entrées et les constantes multiplicatives en considérant les opérations élémentaires comme ayant la même complexité : un accès tableau équivaut à un appel récursif, équivaut à une somme d'entiers équivaut à une multiplication de flottants…Ainsi, on s'intéresse plutôt à déterminer un *ordre de grandeur* de la complexité à calculer, dans le sens des notations de Landau décrites en Section I.

Cette analyse asymptotique joue un rôle crucial dans l'évaluation des performances des algorithmes en fournissant un aperçu général de leur comportement avec des données d'entrée de plus en plus grandes. Elle permet d'identifier les tendances dominantes, de classer les algorithmes en fonction de leur efficacité relative et de prendre des décisions éclairées lors de la sélection d'une solution algorithmique. Cependant, l'analyse asymptotique présente des limites. Elle néglige les facteurs constants et les seuils critiques qui peuvent avoir un impact significatif sur les performances réelles pour des tailles d'entrée modestes ou pour des architectures matérielles spécifiques. De plus, elle peut ne pas capturer les nuances des comportements non asymptotiques, comme les pires cas concrets, les situations de données réelles et les cas atypiques.

Compétences attendues en fin de chapitre

- Manipuler et exprimer des ordres de grandeur de complexité. *On utilisera assez peu grand-Θ et grand-Ω au profit de grand-$\mathcal{O}$, mais je vous ai donné de l'avance sur le cours de maths.*
- Identifier les opérations élémentaires ou non élémentaires d'un algorithme.
- Déterminer la complexité temporelle d'un algorithme itératif ou récursif.
- Définir les notions de complexité temporelle dans le pire cas, dans le meilleur cas, et l'idée de la complexité moyenne et amortie.
- Connaître les complexités des opérations des annexes A.1 et B.1. *Je pense notamment à* `List.length`, `Array.length`, `Array.make`, `String.length` *en OCaml;* `malloc`, `strlen` *en C.*
- Déterminer la complexité temporelle d'un algorithme en OCaml / C ou en pseudocode.
- Définir et déterminer la complexité spatiale d'un algorithme. *Quand on l'aura vu : en prenant en compte la pile d'appel récursive et l'éventuelle optimisation des appels récursifs terminaux.*

I. Un chouia de maths

Définition 1 – Notations de Landau

Soient $u, v \in \mathbb{R}^{\mathbb{N}}$ deux suites de réels. On note :

- $u = \mathcal{O}(v)$ s'il existe $A \in \mathbb{R}_+^*$ et $n_0 \in \mathbb{N}$ tel que pour tout $n \geq n_0$, $|u_n| \leq A|v_n|$.
  On dit que u est "en grand-$\mathcal{O}$ de v".
- $u = \Omega(v)$ s'il existe $A \in \mathbb{R}_+^*$ et $n_0 \in \mathbb{N}$ tel que pour tout $n \geq n_0$, $|u_n| \geq A|v_n|$.
  On dit que u est "en grand-Ω de v".
- $u = \Theta(v)$ s'il existe $A, B \in \mathbb{R}_+^*$ et $n_0 \in \mathbb{N}$ tel que pour tout $n \geq n_0$, $A|v_n| \leq |u_n| \leq B|v_n|$.
  On dit que u est "en grand-Θ de v".

On dira aussi que u est de l'ordre de v si $u = \mathcal{O}(v)$.

Exemple 2

On a :

- $\ln n = \Omega(\log_2 n)$, puisque $\log_2 n = \frac{\ln n}{\ln 2}$
- $3n^2 + 2n - 56 = \mathcal{O}(n^2)$, ce qui sera d'ailleurs le cas de tous les polynômes de degré au plus 2.
- Les suites en grand-$\mathcal{O}$ de 1 sont exactement les suites bornées (juré).

Propriété 3

On a $u = \Theta(v)$ si et seulement si $u = \mathcal{O}(v)$ et $u = \Omega(v)$, si et seulement si $u = \mathcal{O}(v)$ et $u = \Omega(v)$.

Attention !

Ici, l'utilisation du "=" est problématique. Par exemple, on peut avoir $u = \Theta(w)$ et $v = \Theta(w)$ sans forcément avoir $u - v = \Theta(w)$. Pour être plus rigoureux, on pourrait écrire plutôt $u \in \Theta(v)$. Il est important de faire attention dès que l'on écrit une égalité comportant un $\mathcal{O}, \Theta, \Omega$.

Remarque 4

Vous verrez en maths deux notations supplémentaires : la notation $u = o(v)$ quand pour tout $\epsilon > 0$, il existe $n_0 \in \mathbb{N}$ tel que pour tout $n \geq n_0$, on a $u_n = \epsilon v_n$, et la notation $u \sim v$ quand $u - v = o(v)$.

On autorise aussi l'utilisation d'un des symboles de Landau dans une égalité, par exemple, on peut écrire $f(n) = g(n) + \mathcal{O}(h(n))$ quand $f(n) - g(n) = \mathcal{O}(h(n))$. On se limitera à n'utiliser qu'une seule notation de Landau dans chaque égalité, et on fera attention à bien justifier chacune des égalités les mentionnant. Pour classifier ces suites réelles, on introduit encore un peu de vocabulaire :

Définition 5

On utilise les adjectifs suivants sur les suites réelles (on met en gras les plus importantes à connaître par c) :

Adjectif	$\mathcal{O}(\cdot)$
constant	1
logarithmique	$\ln n$
polylogarithmique	$(\ln n)^k, k \in \mathbb{N}$
linéaire	n
quasi-linéaire	$n^{1+\epsilon}$ pour tout $\epsilon > 0$
quadratique	n^2
polynomial	$n^k, k \in \mathbb{N}$
exponentiel	$x^n, x \in \mathbb{R}_+^*$

Remarque 6

L'ordre des fonctions dans le tableau précédent est l'ordre croissant de croissance des fonctions, c'est-à-dire que si f est au-dessus de g, alors $f = \mathcal{O}(g)$. Par exemple, $n^2 = \mathcal{O}(n^3)$.

Retenir cet ordre est notamment utile pour simplifier des expressions de complexité en utilisant les propriétés suivantes.

Propriété 7

Soient u, v, w des suites réelles. Alors :
— si $u = \mathcal{O}(v)$ et $v = \mathcal{O}(w)$, alors $u = \mathcal{O}(w)$;

— si $u = \mathcal{O}(w)$ et $v = \mathcal{O}(w)$, alors $u + v = \mathcal{O}(w)$ et $u - v = \mathcal{O}(w)$;
— si $u = \mathcal{O}(u')$ et $v = \mathcal{O}(v')$, alors $u \times v = \mathcal{O}(u' \times v')$.

Exercice 26 – Démonstration de la Propriété 7

Le montrer.

Exercice 27

★ **Question 1** Classer les fonctions suivantes par ordre croissant de croissance (pour le grand-$\mathcal{O}$) :

1. $n^2 + 2n^2 \ln n + 1 + 2^n + 2^{2n}$
2. $4n^2 + 3n + 1$
3. $3n \ln n + 1$
4. $n^2 + 3n^2 \ln n$
5. $n^2 + 3n^2 \ln n + 1$
6. $2n^2 + 3n^2 \ln n + 1 + 2^n$
7. $n^2 + n \ln n$

II. Complexité temporelle

La première complexité que l'on calcule sur un algorithme, c'est son temps d'exécution. C'est aussi une complexité qui dépend d'une grande variété de facteurs différents : langage d'implémentation, environnement logiciel, versions des programmes, matériel utilisé, etc. Heureusement, quand on réfléchit en termes de complexité asymptotique, il suffit de réfléchir en termes de *nombre d'instructions élémentaires* exécutées. Ainsi, il nous faut :

— estimer le nombre de fois que chaque instruction est exécuté ;
— déterminer quelles instructions sont élémentaires ou pas ;
— estimer la complexité des instructions non élémentaires en terme d'instructions élémentaires qu'elles-mêmes exécutent ;
— sommer toutes ces quantités pour obtenir une estimation du nombre d'instructions élémentaires exécutées.

Définition 8 – Complexité temporelle

La complexité temporelle d'un algorithme est le nombre d'instructions élémentaires exécutées par l'appel à cet algorithme en fonction de ses entrées.

Ici, on parle de la complexité temporelle d'un algorithme comme dépendant de son entrée. Dans la suite, on s'intéressera surtout à la complexité d'un algorithme en fonction de la *taille* de l'entrée.

Remarque 9 – Quand on dit complexité...

Quand on parle de complexité tout court, on entend complexité temporelle.

II.A. Opération élémentaire

Quand on programme en utilisant un certain langage de programmation, il est essentiel de déterminer quelles opérations fournies par le langage sont élémentaires ou non, ce qui nécessite de lire son manuel. Quelques exemples à connaître :

Opérations qui sont élémentaires
- Créer et initialiser une variable ;
- filtrer une valeur, par exemple récupérer la tête et la queue d'un `list` ;
- accéder et modifier un élément d'un `array`, récupérer sa longueur ;
- exécuter des opérations arithmétiques sur les entiers et les flottants, et les comparer ;
- effectuer un appel de fonction (attention : on compte séparément le coût de l'appel seul et le coût de l'évaluation de ses arguments éventuels) ;
- l'allocation de mémoire par un `malloc` sans l'initialiser et sa libération [i].

Opérations qui ne sont pas élémentaires
- accéder au $i$ème élément d'un `list` (c'est linéaire en i, puisqu'il faut parcourir les i premiers éléments de la liste)
- concaténer deux listes `u @ v` (linéaire en la taille de `u`)
- obtenir la longueur d'une liste
- initialiser un tableau

Ce qu'on va calculer Sauf à fixer exactement quelles opérations élémentaires on utilise, on n'exprimera la complexité temporelle qu'en terme d'ordres de grandeur au sens de la Section I.

Remarque 10 – On essaye de trouver la complexité la plus fine possible

Le but du calcul de la complexité est d'évaluer le mieux possible le temps de calcul d'un algorithme. Ainsi, il ne sert à rien de montrer qu'un algorithme a une complexité en grand-$\mathcal{O}$ de $2^{n!}$ alors qu'il est clairement linéaire : on essaiera toujours de calculer la complexité *la plus fine* possible, idéalement un petit-o proche asymptotiquement de la complexité ou un grand-Θ.

II.B. Complexité dans le pire cas

La première évaluation de la complexité temporelle d'un algorithme est celle *dans le pire des cas* :

Définition 11 – Complexité temporelle dans le pire cas

La complexité temporelle dans le pire cas en $n \in \mathbb{N}$ d'un algorithme est la complexité temporelle maximale de cet algorithme sur toutes les entrées de taille n.

on majore le nombre d'opérations élémentaires nécessaires à l'exécution d'un programme sur une entrée d'une taille n, pour tout $n \in \mathbb{N}$. Par exemple, réimplémentons la fonction `List.mem` :

i. Oui, c'est un mensonge, mais la vraie réponse est que ça dépend de l'humeur de votre machine ce qui n'est pas très satisfaisant.

OCaml
```
let rec appartient l x = match l with
  | [] -> false
  | h :: t -> h = x || appartient t x
```

Sur une entrée `l`, `x` telle que `x` est la tête de `l`, la complexité est constante alors que si `x` n'apparaît pas dans `l`, la complexité temporelle de `appartient` est en $\mathcal{O}(|l|)$. La complexité au pire s'intéresse justement à ce pire cas.

Remarque 12

On n'a pas vraiment défini précisément la notion de "taille" d'une entrée : c'est tout simplement que ça dépendra de la situation. Par exemple, pour une fonction prenant un entier en entrée, on pourra considérer que la taille de cet entier est l'entier lui-même, la longueur de sa représentation…Et le même genre de considération se pose pour d'autres types de données, comme on le verra dans le Chapitre 10.

II.C. Complexité dans le meilleur cas

Définition 13 – Complexité temporelle dans le meilleur cas

La complexité temporelle dans le meilleur cas en $n \in \mathbb{N}$ d'un algorithme est la complexité temporelle minimale de cet algorithme sur toutes les entrées de taille n.

La complexité dans le meilleur cas s'intéresse au contraire au meilleur cas, c'est-à-dire à l'entrée de taille n tel que le programme utilise le moins d'opérations élémentaires possible. Cette complexité est intéressante dans des cas précis, par exemple pour décrire l'intérêt de certains algorithmes ayant une mauvaise complexité dans le pire cas, mais ayant une excellente complexité dans des cas précis. C'est, par exemple, le cas de l'algorithme de tri par insertion. Dans le pire cas, il a une complexité en $\mathcal{O}(n^2)$, mais dans le cas des tableaux triés ou presque déjà triés, sa complexité est en fait en $\mathcal{O}(n)$, ce qui est plus efficace de beaucoup d'algorithmes de tri dit "efficaces".

Exercice 28

Quelle est la complexité dans le meilleur cas de la fonction `appartient` ?

II.D. Complexité en moyenne

C'est en même temps la complexité qui s'approche le plus de l'idée intuitive de complexité ("en gros, combien de temps ça prend") et la plus dure à définir et calculer formellement. Elle est notamment utile dans le cas ou certains cas *pathologiques* amènent à une mauvaise complexité au pire, alors que ces cas n'arrivent que très rarement.

Définition 14 – Complexité temporelle en moyenne

La complexité en moyenne d'un algorithme exprime pour $n \in \mathbb{N}$ la moyenne de la complexité de l'exécution de cet algorithme sur toutes les entrées de taille n.

Cette définition amène à se poser des questions non triviales :

— Quelle est la distribution de probabilité des entrées de l'algorithme ? Parfois elle est difficile à exprimer, déterminer ou évaluer. Souvent, elle n'existe tout simplement pas.
— Ensuite, le calcul de la moyenne entraîne souvent des calculs mathématiques difficiles à mener à terme pour obtenir une évaluation simple de la complexité.
— Enfin, quand on a calculé la complexité en moyenne d'un algorithme, quelle est la probabilité de tomber sur une entrée pathologique où la complexité de l'algorithme est bien pire que prévue ?

Un exemple classique de calculs de complexité en moyenne : pour l'algorithme de tri rapide, sa complexité au pire est quadratique, mais sa complexité en moyenne est quasi-linéaire (ce que l'on montrera en TD / TP).

II.E. Complexité amortie

Dans le cas où la complexité au pire est mauvaise et que la complexité en moyenne est meilleure, on peut vouloir raffiner notre complexité en considérant non pas la *moyenne* de la complexité d'un algorithme, mais sa complexité *amortie* : l'idée est de compter le nombre d'opérations élémentaires engendrées par n appels successifs à l'algorithme, puis de diviser par n le résultat. L'exemple le plus répandu de ce genre de calcul est la complexité amortie de l'opération `t.append(x)` en Python, avec `t` un tableau redimensionnable (qui correspond au type `list` de Python) : la plupart du temps, cette opération a une complexité constante, et rarement une complexité linéaire en la taille de `t`. On implémentera en C cette structure de données.

Cette notion de complexité amortie est pertinente quand on analyse la complexité d'un algorithme contenant beaucoup d'appels à la même fonction auxiliaire. Cela pourra arriver d'utiliser cette notion dans des algorithmes utilisant des structures de données complexes.

Remarque 15 – Quand on dit complexité...

Quand on parle de complexité sans préciser, on parle de la complexité dans le pire cas.

III. Comment estimer la complexité temporelle ?

III.A. Sur un algorithme récursif

Dans un algorithme récursif, le calcul de la complexité temporelle d'un algorithme s'exprime naturellement comme une relation de récurrence.

Exemple 16 – Retournement de liste en OCaml

Considérons le problème suivant : pour une liste `l`, calculer la liste retournée. On en a vu une version naïve en TP :

OCaml
```
let rec retourne_naif l = match l1 with
  | [] -> []
  | h :: t -> (retourne_naif t) @ [h]
```

Sachant que la concaténation `@` a une complexité linéaire, on peut évaluer la complexité $C(n)$ de `retourne_naif` sur une entrée de taille n comme suis :

$$\begin{cases} C(0) = B \\ \text{pour } n \in \mathbb{N},\ C(n+1) = An + C(n) \end{cases}$$

Avec $A, B > 0$ des constantes, c'est-à-dire pour tout $n \in \mathbb{N}$,

$$\begin{aligned} C(n) &= C(0) + 0 + A + \cdots + A(n-1) \\ &= B + A\frac{n(n-1)}{2} \\ &= \Theta(n^2) \end{aligned}$$

Ensuite, on avait écrit une version plus efficace de cette fonction :

OCaml
```
let retourne_efficace l =
  let rec aux l_aux acc = match l_aux with
    | [] -> acc
    | h :: t -> aux t (h :: acc)
  in aux l []
```

Pour cette fonction, la complexité de aux dépend en théorie de n la taille de `l_aux` et m la taille de acc. On a :

$$\begin{cases} C'(0, m) = B \\ C'(n+1, m) = C'(n, m+1) + A \end{cases}$$

avec $A, B > 0$ des constantes. On peut montrer qu'une expression explicite est $C'(n, m) = An + B$ (et qu'elle ne dépend pas de m). Cela nous donne une complexité linéaire de la fonction `retourne_efficace` en fonction de la longueur de la liste.

Exercice 29 – Un premier calcul de complexité

Considérons le problème suivant : pour une liste de la forme $[\ell_0; \ldots; \ell_{n-1}]$, renvoyer la liste $[\ell_0 + n - 1; \ell_1 + n - 2; \ldots; \ell_{n-1} + 0]$. Un algorithme naïf résolvant ce problème est le suivant, en OCaml :

OCaml
```
let rec f l = match l with
  | [] -> []
  | h :: t -> h + List.length t :: f t
```

▶ **Question 1** Calculer sa complexité temporelle.

▶ **Question 2** En proposer une nouvelle version de complexité linéaire.

Exercice 30 – Complexité de l'exponentiation rapide

On a déjà montré dans l'Exemple 9 que la fonction suivante calculait x^n pour $x \in \mathbb{Z}$ et $n \in \mathbb{N}$:

OCaml
```
let rec expo_rapide x n =
  if n = 0 then 1
  else if n mod 2 = 0 then expo (x * x) (n / 2)
  else x * expo (x * x) (n / 2)
```

▶ **Question 1** Notons $M(n)$ le nombre de multiplications effectuées pendant l'appel `expo_rapide x n`. Exprimer $M(2n+1)$ et $M(2n)$ en fonction de $M(n)$.

▶ **Question 2** Montrer que $M(n) = \Theta(\log_2 n)$. *On pourra montrer que pour tout* $n \in \mathbb{N}^*$, $M(n) \leq \log_2 n + 1$.

▶ **Question 3** En déduire la complexité de la fonction `expo_rapide` en fonction de `n`.

III.B. Sur un algorithme itératif

Pour les algorithmes itératifs, pour chaque boucle on procède en deux temps : d'abord, on identifie la complexité du corps de la boucle, éventuellement en fonction du contexte de la boucle, puis on somme ces complexités pour chaque tour de boucle. Le cas le plus simple et le plus courant est que la complexité d'un seul tour de boucle est constante : alors la complexité de l'algorithme est de l'ordre du nombre de tours de boucle.

Pour une boucle `for` Pour une boucle `for`, il suffit de sommer le nombre d'opérations élémentaires de chaque itération de la boucle entre l'indice de départ et d'arrivée. Par exemple, pour une fonction similaire à `List.map` pour les tableaux en C tel que `vectoriser_f(t)` renvoie un tableau des images des éléments de `t` par `f` :

C
```
// on suppose que l'on a déjà défini int f(int x);
int* vectoriser_f(int t[], int n) {
    int* res = (int*) malloc(n * sizeof(int));
    for (int k = 0; k < n; k++) {
        res[k] = f(t[k]);
    }
    return res;
}
```

Alors, la complexité de la fonction `vectoriser_f` dépend de la complexité de `f` : si on a simplement `f(x) = x + 1`, alors chaque tour de boucle prend un temps constant et on a :

$$C(n) = \Theta\left(\sum_{i=0}^{n-1} 1\right) = \Theta(n)$$

Parfois, le calcul est plus complexe quand la complexité d'une itération dépend aussi de n. Par exemple, si l'on a :

C
```c
// renvoie le nombre d'éléments de t strictement inférieurs à t[k]
int f(int t[], int n, int k) {
    int x = 0;
    for (int j = 0; j < n; j++) {
        if (t[j] < t[k]) { x = x + 1; }
    }
    return x;
}
```

Dans cet exemple, la complexité d'un tour de boucle dépend de `n` (il est en fait en $\Theta(n)$, puisque la complexité de `f` est elle-même linéaire en `n` et ne dépend pas de ses autres entrées). La complexité de `exemple` est donc :

$$C(n) = \Theta\left(\sum_{j=0}^{n-1} n\right) = \Theta(n^2)$$

Exercice 31 – Mais qu'est ce qu'on vient d'écrire...

▶ **Question 1** On vient d'écrire qu'une somme de n grand-Θ de n était égal à un grand-Θ. C'est vrai ça ?

▶ **Question 2** Reprendre le calcul de la complexité dans le cas où dans `f`, on remplace la ligne 5 par `for j = 0 to k-1 do`.

Remarque 17 – Sur les sommations des relations d'équivalence

On prendra garde aux sommations brutales d'équivalents : par exemple, si l'on fait une boucle `for j = 0 to n-1 do` contenant des appels prenant un appel de complexité $\Theta(2^j)$, une majoration grossière donne une complexité en $\mathcal{O}(n2^n)$ alors qu'on a $\Theta(2^n)$. On prendra garde à vérifier que la complexité que l'on calcule n'est pas largement surestimée par ce genre de "brutalité mathématique".

Exercice 32 – Tri par sélection

Une variante du tri par insertion vu en TP est le tri par sélection : le principe est de déterminer le plus petit élément d'un tableau, puis le second le plus petit, puis le troisième le plus petit, etc. Voici son code en Python, on l'a déjà écrit en C.

Python
```python
def swap(t, i, j):
    temp = t[i]
    t[i] = t[j]
    t[j] = temp

def selection_sort(t):
    n = len(t)
    for i in range(n-1):
        j_min = i
        for j in range(i+1, n):
            if t[j] < t[j_min]:
                j_min = j
        swap(t, i, j_min)
```

▶ **Question 1** Déterminer sa complexité temporelle dans le pire et le meilleur cas.

Pour une boucle `while` C'est le même principe que la boucle `for`, sauf que l'on ne sait pas à l'avance combien de tours de boucle seront utilisés. Il faut aussi faire attention à prendre en compte la complexité de l'évaluation de la condition de boucle. Un exemple en Python qui a ces deux soucis :

Python
```python
## Cherche le premier indice tel que la somme dépasse la borne
def f(t, borne):
    j = 0
    while j < len(t) and sum(t[:j]) < borne:
        j += 1
    return j
```

Ici, le nombre d'itérations du `while` est difficile à prédire exactement : on sait cependant qu'il est borné par `len(t)`, et que la condition prend au plus un temps linéaire en `len(t)` à être évalué. En notant $n =$ `len(t)`, on a :

$$C(n) \leq \sum_{i=0}^{n-1} An \text{ avec } A > 0 \text{ fixé}$$
$$C(n) = \mathcal{O}(n^2)$$

Remarque 18 – Variant de boucle et complexité

La valeur initiale d'un variant de boucle `while` donne un majorant du nombre d'itérations : après une preuve de terminaison/correction, c'est un bon point de départ pour évaluer la complexité d'une fonction. Attention, alors qu'un variant de boucle n'a pas besoin d'être "optimisé" pour montrer la terminaison (il suffit qu'il décroisse strictement et soit minoré), on fera attention à ce qu'il ne donne pas une majoration trop grossière de la complexité.

IV. Complexité en espace

Pour estimer la quantité de mémoire nécessaire à l'exécution d'un algorithme, il faut repérer quelles sont les données stockées en mémoire par un algorithme et estimer la place qu'ils prennent en mémoire. On verra plus en détail l'implémentation de ces modèles mémoires.

Définition 19 – Complexité spatiale

La complexité spatiale d'un algorithme est la quantité maximale de mémoire utilisée à un instant donné au cours de l'exécution, exprimé en fonction des entrées.

Pour l'instant, on se fixe les conventions suivantes :
— Chaque variable déclarée compte pour 1.
— Chaque structure déclarée compte pour une constante, plus la taille de chacune de ses composantes. Par exemple, pour une liste `l = h :: t`, sa taille est de 1 plus la taille de `h` plus la taille de `t`.
— La taille d'un tableau de longueur n est n fois la taille de chacun de ses éléments.
— La taille des entrées de l'algorithme et de sa sortie ne sont pas comptées : si un algorithme renvoie un tableau de taille n mais n'utilise que des variables auxiliaires de taille constante, la complexité spatiale est constante.

Exemple 20 – Tri par insertion

Rappelons le tri par insertion vu en TP :

OCaml
```
let insere_tableau t i =
  let x = t.(i) in
  let j = ref i in
  while !j > 0 && t.(!j - 1) > x do
    t.(!j) <- t.(!j - 1);
    j := !j - 1
  done;
  t.(!j) <- x

let tri_insertion_tableau t =
  for i = 1 to Array.length t - 1 do insere_tableau t i done
```

La fonction `insere_tableau` n'utilise que deux variables locales `x` et `j`. Sa complexité spatiale est donc constante. De même, la fonction `tri_insertion_tableau` n'utilise qu'une variable locale `i` : ensuite, au cours de la boucle, la seule mémoire utilisée est celle utilisée pendant l'appel à `insere_tableau`, qui est constante. Ici, on n'a pas besoin de sommer ces complexités spatiales puisqu'on ne s'intéresse qu'au maximum de quantité de mémoire utilisée au cours du temps : la complexité spatiale de `tri_insertion_tableau` est donc constante.

Exemple 21 – Tri fusion

On a vu en TP le tri fusion sur les listes. On peut aussi en écrire une version pour tableau :

Pseudo-code

```
Requiert : T, T_1, T_2 un tableau
1 : Fonction FUSION(T_1, T_2)
2 :     T ← un tableau de taille |T_1| + |T_2|
3 :     i, j ← 0, 0
4 :     Pour tout k ∈ [[0, |T|]] , faire
5 :         Si T_1(i) ≤ T_2(j) alors
6 :             T(k) ← T_1(i)
7 :             i ← i + 1
8 :         Sinon
9 :             T(k) ← T_2(j)
10 :            j ← j + 1
11 :    retourne T

12 : Fonction TRIFUSION(T)
13 :    n ← |T|
14 :    Si n ≤ 1 alors
15 :        retourne T
16 :    Sinon
17 :        retourne FUSION( TRIFUSION(T [: n/2]), TRIFUSION(T [n/2 :]) )
```

Notons $C(n)$ la complexité spatiale de FUSION(s)ur une entrée de taille $n = |T_1|+|T_2|$: on crée un tableau de taille n que l'on renvoie et on utilise deux variables locales i et j. La complexité spatiale de FUSION(e)st donc $\Theta(1)$ sans compter la sortie.
Notons $C'(n)$ la complexité spatiale de TRIFUSION(s)ur un tableau de taille n. Pour $n > 1$, cet algorithme crée deux sous-tableaux de taille $\frac{n}{2}$, les tries successivement par deux appels récursifs à TRIFUSION(e)t renvoie le résultat de leur FUSION(.) Créer les deux sous-tableaux prend $\Theta(n)$ en mémoire, les appels récursifs prennent chacun $C'(\frac{n}{2})$ de mémoire, la fusion est en grand-Θ, ce qui donne une complexité spatiale totale en $C'(n) = \max(\Theta(n), C'(\frac{n}{2}))$ [a]. On peut montrer que la complexité spatiale obtenue est en $\Theta(n)$.

a. Cette expression n'est pas jolie, mais on comprend l'idée.

Remarque 22 – Sur la complexité des appels récursifs

Le fait d'utiliser des appels consomme aussi de la mémoire même si ce n'est pas explicite : comme on le verra dans le chapitre Chapitre 6, les appels récursifs successifs sont stockés dans une *pile d'appel*, où chaque bloc contient les variables locales et les arguments des appels récursifs encore en cours. Notons que ce stockage ralentit aussi la vitesse d'exécution des algorithmes récursifs, comparé aux algorithmes impératifs équivalents. Quand cette pile est remplie, OCaml vous préviens et affiche `Stack overflow during evaluation (looping recursion?).`. Notez que quand cela apparaît, c'est presque instantané : la taille maximale de pile d'appels récursifs est très rapide à rem-

plir, ce qui arrive surtout quand on effectue des appels récursifs qui bouclent à l'infini. Notons aussi qu'il est possible d'augmenter la taille de la pile d'appel en C comme en OCaml, ce qui est rarement utile.

V. Compromis coût-mémoire

On a déjà rencontré des problèmes dont deux versions différentes effectuent un compromis entre la complexité temporelle et mémoire. Par exemple, pour calculer le coefficient binomial $\binom{n}{k}$, on peut l'implémenter par une version récursive naïve `binom_rec` ou par une fonction impérative dans laquelle on remplit une matrice des valeurs de 0 .

C
```c
int binom_rec(int k, int n) {
  if (k == 0 || k == n) { return 1; }
  else { return binom_rec(k-1, n-1) + binom_rec(k, n-1); }
}
```

C
```c
// Un peu plus de travail pour cette version :
int** creer_matrice_nulle(int nb_lignes, int nb_colonnes) {
  int** mat = malloc(nb_lignes * sizeof(int*));
  for (int i = 0; i < nb_lignes; i++) {
    mat[i] = malloc(nb_colonnes * sizeof(int));
    for (int j = 0; j < nb_colonnes; j++) {
      mat[i][j] = 0;
    }
  }
  return mat;
}

void liberer_matrice(int** mat, int nb_lignes) {
  // pas besoin de connaître le nb de colonnes ici
  for (int i = 0; i < nb_lignes; i++) {
    free(mat[i]);
  }
  free(mat);
}

int binom_mem(int k, int n) {
  int** mat = creer_matrice_nulle(n+1, n+1);
  int aux(int i, int j) {
    if (i < 0 || i > j) { return 0; }
    if (i == 0 || i == j) { return 1; }
    else {
      if (mat[i][j] == 0) {
        mat[i][j] = aux(i-1, j-1) + aux(i, j-1);
      }
      return mat[i][j];
```

```
    }
  }
  int res = aux(k, n);
  liberer_matrice(mat, n+1);
  return aux(k, n);
}
```

Le principe de la deuxième version est de stocker dans une matrice de taille $(n+1, n+1)$ les valeurs successives de $\binom{n}{k}$ dès qu'on les a calculées.

La première version a une complexité exponentielle en temps, mais n'utilise de la mémoire que pour la pile d'appel ($\mathcal{O}(n)$). La seconde version a une complexité mémoire quadratique pour stocker toutes les valeurs de $\binom{i}{j}$, mais n'a besoin de les calculer chacune qu'une seule fois ce qui entraîne une complexité temporelle en $\mathcal{O}(n^2)$.

Dans la seconde version, on est parti de la valeur à calculer, et l'on a utilisé des appels récursifs jusqu'à tomber sur des cas de base où sur des cas déjà calculés, ce qui est illustré dans la Figure 7.1.

	$k=0$	1	2	3	4	5	6
$n=0$	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4			
5		5	10	10			
6				20			

FIGURE 7.1. – Matrice des nombres binomiaux à la fin de la fonction `binom_mem 6 3`

Une autre façon de procéder serait de calculer la matrice des $\binom{n}{k}$ ligne par ligne ("n par n") jusqu'à arriver à la ligne contenant le n voulu. Cela viens de l'observation que pour calculer $\binom{n}{k}$, par la formule du triangle de Pascal il suffit de connaître la valeur des coefficients binomiaux au rang $n-1$. Ainsi, on peut calculer *ligne par ligne* la matrice ci-dessus, en ne stockant qu'une ligne à la fois et en oubliant la ligne $n-1$ au fur et à mesure qu'on calcule la ligne n.

C
```
int binom(int k, int n) {
  int* ligne = malloc((n+1) * sizeof(int));
  for (int i = 0; i <= n; i++) {
    ligne[i] = 1;
    for (int j = i-1; j >= 1; j--) {
      ligne[j] = ligne[j] + ligne[j-1];
    }
  }
  int res = ligne[k];
  free(ligne);
  return res;
```

```
12 }
```

La complexité temporelle est aussi quadratique dans cette troisième version, mais la complexité spatiale est maintenant *linéaire*. On reverra ce genre de considérations quand on parlera de programmation dynamique dans le Chapitre 13.

En résumé :

Version	Complexité temporelle	Complexité spatiale
`binom_rec`	$\mathcal{O}(2^n)$	$\mathcal{O}(n)$
`binom_mem`	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
`binom`	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$

Bibliographie