1 Introduction

L'analyse de complexité est l'étude formelle de l'utilisation des ressources d'un programme ou d'un algorithme : le temps nécessaire, la mémoire utilisée, etc... On ne s'intéresse pas à l'utilisation exacte de ressources : l'analyse de complexité se fait à l'ordre de grandeur.

Dans ce chapitre, on s'intéresse à la complexité temporelle, c'est à dire au temps nécessaire à l'exécution d'un programme, d'un algorithme, d'une fonction... L'unité de base de la complexité est ***l'opération élémentaire***. Une opération élémentaire aura par définition un coût de 1, et la complexité d'un programme sera donc le nombre d'opérations élémentaires utilisées.

La notion d'opération élémentaire peut varier en fonction du contexte. Pour commencer, considérons que les opérations suivantes sont élémentaires :

— les assignements de valeur à une variable

— les additions, multiplications, comparaisons, etc...

— les référencements, déréférencements, accès à un élément d'un tableau

Le nombre d'opérations élémentaires effectuées par un appel de fonction dépend de ses arguments. Par exemple :

```
1  // compte le nombre d'occurences de x dans T tableau de taille n
2  int compter_occs(int* T, int n, int x){
3    int count = 0; // une o.e.
4    int i = 0; // une o.e.
5    while (i < n){ // une o.e.
6        if (T[i] == x){ // deux o.e.
7          count++; // une o.e.
8        }
9        i++; // une o.e.
10   }
11   return count;
12 }
```

Cette fonction fera 2 opérations élémentaires avant la boucle, puis pour chaque passage de boucle, entre 4 et 5 opérations élémentaires. Le nombre d'opérations élémentaires effectuées lorsqu'on appelle la fonction sur un tableau de taille n est donc dans l'intervalle

$$[\![2+4n, 2+5n]\!]$$

Lorsque l'on analyse la complexité d'une fonction, on regarde, pour une taille d'entrée $n \in \mathbb{N}$ donnée, le nombre maximal d'opérations élémentaires effectuées sur les entrées de taille n. On parle de ***complexité pire cas***. La complexité pire cas d'une fonction f est donc donnée par la fonction mathématique :

$$C_f(n) = \max_{e \text{ entrée de taille } n} Op_f(e)$$

Où $Op_f(e)$ est le nombre d'opérations élémentaires effectuées par f sur l'entrée e.

La manière dont on définit la taille exacte d'une entrée dépend du cadre de travail. Par exemple, ça peut être le nombre d'octets qui constituent l'entrée : pour la fonction compte_zero, la taille d'une entrée serait $4n + 4$.

2 Notation de Landau

On ne s'intéresse pas au nombre exact d'opérations qui sont faites, mais à l'ordre de grandeur. L'idée est qu'on ne fait pas la différence entre une fonction f dont la complexité vérifie $C_f(n) = n$ et une fonction g dont la complexité vérifie $C_g(n) = 8+3n$ car les deux sont **linéaires**.

Les notations de Landau permettent de formaliser cette notion d'ordre de grandeur.

Définition 1

Soit $v = (v_n)_{n\in\mathbb{N}} \in (\mathbb{R}^+)^{\mathbb{N}}$ une suite positive.
On définit les ensembles $\mathcal{O}(v)$, $\Omega(v)$ et $\Theta(v)$ par :

- $\forall u = (u_n)_{n\in\mathbb{N}} \in (\mathbb{R}^+)^{\mathbb{N}}$ suite positive, $u \in \mathcal{O}(v)$ si et seulement si u est majorée par v fois une constante à partir d'un certain rang, autrement dit si et seulement si :

$$\exists C \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq n_0 \Rightarrow u_n \leq C v_n$$

- $\forall u = (u_n)_{n\in\mathbb{N}} \in (\mathbb{R}^+)^{\mathbb{N}}$ suite positive, $u \in \Omega(v)$ si et seulement si u est minorée par v fois une constante à partir d'un certain rang, autrement dit si et seulement si :

$$\exists C \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq n_0 \Rightarrow u_n \geq C v_n$$

- $\forall u = (u_n)_{n\in\mathbb{N}} \in (\mathbb{R}^+)^{\mathbb{N}}$ suite positive, $u \in \Theta(v)$ si et seulement si u est à la fois majorée par v fois une constante et minorée par v fois une autre constante à partir d'un certain rang, autrement dit si et seulement si :

$$\exists C_1, C_2 \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq n_0 \Rightarrow C_1 v_n \leq u_n \leq C_2 v_n$$

$\mathcal{O}(v)$ est donc l'ensemble des suites qui grandissent au plus aussi vite que v asymptotiquement. Si u est dans $\mathcal{O}(v)$, on dit que u est ***dominée*** par v.

$\Omega(v)$ est l'ensemble des suites qui grandissent au moins aussi vite que v asymptotiquement. Si u est dans $\Omega(v)$, on dit que u et v sont équivalentes.

Proposition 1

Pour $u = (u_n)_{n\in\mathbb{N}} \in (\mathbb{R}^+)^{\mathbb{N}}$ et $v = (v_n)_{n\in\mathbb{N}} \in (\mathbb{R}^+)^{\mathbb{N}}$, on a :

- $u \in \mathcal{O}(v) \Leftrightarrow v \in \Omega(u)$
- $u \in \Theta(v) \Leftrightarrow u \in O(v) \cap \Omega(v)$

Démonstration. Exercice ! □

Exercice 1

Montrer les propriétés suivantes :

- $(n)_{n\in\mathbb{N}} \in \mathcal{O}((n^2)_{n\in\mathbb{N}})$
- $(3n+2)_{n\in\mathbb{N}} \in \theta((n)_{n\in\mathbb{N}})$
- $(3n+2)_{n\in\mathbb{N}} \in \mathcal{O}((n^2))_{n\in\mathbb{N}}$
- $(3n+2)_{n\in\mathbb{N}} \notin \theta((n^2)_{n\in\mathbb{N}})$

Proposition 2

- La relation définie par $\mathcal{O}$ est transitive : pour $u, v, w \in (\mathbb{R}^+)^{\mathbb{N}}$, on a :

$$u \in \mathcal{O}(v) \text{ et } v \in \mathcal{O}(w) \Rightarrow u \in \mathcal{O}(w)$$

- Elle est aussi réflexive : pour toute suite $u \in (\mathbb{R}^+)^{\mathbb{N}}$, on a $u \in \mathcal{O}(u)$.
- De même, Ω induit une relation transitive et réflexive.
- La relation induite par Θ est transitive, réflexive et symétrique : c'est une relation d'équivalence.

Démonstration. (On montre le premier point, le reste est laissé en exercice.)
Soient $u, v, w \in (\mathbb{R}^+)^{\mathbb{N}}$ telles que $u \in \mathcal{O}(v)$ et $v \in \mathcal{O}(w)$. Alors, par définition, il existe $n_1 \in \mathbb{N}$ et $C_1 > 0$ tels que $\forall n \geq n_1, u_n \leq C_1 v_n$. De même, il existe $n_2 \in \mathbb{N}$ et $C_2 > 0$ tels que $\forall n \geq n_2, v_n \leq C_2 w_n$.
Posons $n_0 = \max(n_1, n_2)$. Pour $n \geq n_0$, les deux inégalités sont vraies, et donc $u_n \leq C_1 v_n \leq C_1 C_2 w_n$. Donc $u \in \mathcal{O}(w)$. □

Notation Soient $u = (u_n)_{n\in\mathbb{N}} \in (\mathbb{R}^+)^{\mathbb{N}}$ et $v = (v_n)_{n\in\mathbb{N}} \in (\mathbb{R}^+)^{\mathbb{N}}$ sont deux suites entières. Pour noter $u \in \mathcal{O}(v)$, on écrit :

$$u_n = \mathcal{O}(v_n)$$

Attention, **ce n'est qu'une notation** ! Par exemple, on pourra écrire $n = \mathcal{O}(n^2)$ et $n^2 = O(n^2)$, alors que ça n'aurait aucun sens d'écrire $n = n^2$.

3 Complexité asymptotique

A Premier exemple

Lorsque l'on donne la complexité d'une fonction, on l'exprime avec les notations de Landau, en fonction de la taille de ses entrées.

Exemple 1

On considère le problème suivant : étant donné un tableau T de longueur n, on souhaite vérifier si T est trié dans l'ordre croissant, i.e. si :

$$\forall i \in [\![0, n-1]\!], \forall j \in [\![0, i]\!], T[j] \leq T[i]$$

On propose un premier algorithme pour résoudre ce problème :

Algorithme 1 : Est_trié_1

Entrée(s) : T tableau de longueur n
Sortie(s) : Vrai si le tableau est trié, faux sinon

```
1 pour i = 0 à n − 1 faire
2     pour j = 0 à i faire
3         si T[j] > T[i] alors
4             retourner Faux;
5 retourner Vrai;
```

On dit que cet algorithme est **naïf** car il reflète exactement l'énoncé du problème : on vérifie littéralement si pour tout $i \in [\![0, n-1]\!]$, pour tout $j \in [\![0, i]\!]$, on a bien $T[j] \leq T[i]$: dès qu'on trouve deux indices qui ne vérifient pas la condition, on renvoie faux, et si l'on a essayé tous les couples (i, j) sans jamais contredire la condition, on renvoie vrai.

Avant d'évaluer la complexité de cet algorithme, il faut identifier la taille de ses entrées. Si T est un tableau de n cases, alors sa taille exacte dépend de la taille des cases de T, c'est à dire, par analogie avec le C, du type du tableau. Dans la plupart des cas, on ignore le fait que les cases du tableaux peuvent prendre une place arbitrairement grande, et on considère que la taille d'un tableau de longueur n est n. On exprimera donc la complexité de l'algorithme **en fonction de** n.

Soit $n \in \mathbb{N}$. Comptons le nombre d'opérations effectuées par cet algorithme sur un tableau de n cases. A la ligne 3, on effectue 2 accès à un tableau, et une comparaison, soit 3 opérations élémentaires. De plus, à chaque début de passage de boucle on doit incrémenter la variable de boucle, soit une opération par passage. Enfin, on doit calculer $n-1$ au tout début du programme pour pouvoir lancer la boucle pour. Le nombre total d'opérations est donc au plus :

$$\begin{aligned} C(n) &\leq 1 + \sum_{i=0}^{n-1} \left(1 + \sum_{j=0}^{i}(1+3)\right) \\ &\leq 1 + n + 4n^2 \\ &\leq 1 + 5n^2 \end{aligned}$$

Donc, la complexité $C(n)$ de la fonction vérifie $C(n) = \mathcal{O}(n^2)$. Autrement dit, l'algorithme fera toujours **au plus** de l'ordre de n^2 opérations. Remarquons que l'on a aussi techniquement $C(n) = O(n^{100})$. Lorsque l'on exprime la complexité d'une fonction, on essaie toujours de trouver le $\mathcal{O}$ le plus petit possible, car ça sera le plus fidèle.
De plus, pour $n \in \mathbb{N}$, si l'on considère le tableau $T_n = [0, 1, \ldots, n-1]$, on remarque que l'exécution de l'algorithme sur T_n prendra exactement de l'ordre de n^2 opérations, car on exécute tous les passages des deux boucles. Ainsi, on sait que le pire cas (qui n'est peut être pas ce T_n) demande **au moins** de l'ordre de n^2 opérations, autrement dit que $C(n) = \Omega(n^2)$. En combinant les deux points précédents, on obtient que $C(n) = \Theta(n^2)$, autrement dit, le pire cas de cet algorithme prend exactement de l'ordre de n^2 opérations. Finalement, on voit que la famille $(T_n)_{n\in\mathbb{N}}$ considérée plus haut est bien un pire cas !
Notons que $\Theta(n^2)$ ne veut pas dire que l'algorithme s'exécute en temps proportionnel à n^2 sur toutes les entrées. Par exemple, si les deux premières cases du tableau sont dans le mauvais ordre, peu importe sa taille, l'algorithme s'arrête après au plus 6 opérations.

On propose maintenant un deuxième algorithme, plus efficace, sur lequel on effectue le même travail :

Algorithme 2 : Est_trié_2

Entrée(s) : T tableau de longueur n
Sortie(s) : Vrai si le tableau est trié, Faux sinon

```
1 pour i = 0 à n − 2 faire
2     si T[i] > T[i + 1] alors
3         retourner Faux;
4 retourner Vrai;
```

Montrons la correction de cet algorithme. On remarque tout d'abord que si l'algorithme renvoie Faux, alors on a trouvé un indice i tel que $T[i] > T[i+1]$, et donc le tableau n'est pas trié. Supposons maintenant que l'algorithme renvoie Vrai, et montrons qu'alors le tableau est trié. Notons qu'on ne rentre donc jamais dans le Si ligne 2.

Pour $k \in [\![0, n-1]\!]$ on note $T[0..k]$ le sous tableau de T composé des case $T[0], T[1]...T[k]$. On montre l'invariance de la propriété suivante :

"le sous tableau T[0..i] est trié"

Pour $k \in \mathbb{N}$, notons T_k et i_k les valeurs de T et i après k passages de boucles. Montrons donc par récurrence que $\forall k \in \mathbb{N}$, $P(k)$: "le sous tableau $T[0..i_k]$ est trié"

— En entrée de boucle, $i_0 = 0$, le sous tableau $T[0..0]$ ne contient qu'une seule case, il est forcément trié : $P(0)$ est vraie.

— Soit $k \in \mathbb{N}$ telle que $P(k)$ vraie. Considérons un $k+1$-ème passage de boucle. Par H.R., le sous tableau $T_k[0..i_k]$ est trié. Montrons que $T_{k+1}[0..i_{k+1}]$ est trié.
Notons que l'on a $i_{k+1} = i_k + 1$, et $T_{k+1} = T_k$ (le tableau n'est jamais modifié lors de l'algorithme). Dans la suite, on notera simplement T.
On a $T[0..i_k]$ trié, donc pour tout $0 \leq j_1 \leq j_2 \leq i_k$, on a $T[j_1] \leq T[j_2]$. De plus, comme on ne rentre pas dans le Si, on sait que $T[i_k] \leq T[i_k+1] = T[i_{k+1}]$. Donc, pour tout $j \in [\![0, i_k]\!]$, on a $T[j] \leq T[i_k] \leq T[i_{k+1}]$. Finalement, $T[0..i_{k+1}]$ est trié : $P(k+1)$ est vraie.

P est un invariant de boucle, et en particulier en sortie de boucle $i = n-1$ et le tableau tout entier est donc trié.
Donc l'algorithme renvoie bien Vrai si et seulement si le tableau est trié : il est correct.

Étudions maintenant la complexité de cet algorithme. Comme précédemment, sur un tableau de taille n, on effectue au plus n passages de boucle, chacun prenant un nombre borné d'opérations (au plus 5) : l'algorithme est en $\mathcal{O}(n)$.
De plus, pour $n \in \mathbb{N}$, en considérant à nouveau $T_n = [0, 1, \ldots, n-1]$, on voit que l'algorithme fait tous les tours de boucles, donc au moins n opérations. Ainsi, l'algorithme est en $\Omega(n)$, et donc en $\Theta(n)$.

A retenir Pour exprimer la complexité $C(n)$ d'un algorithme :

— Sous la forme d'un $\mathcal{O}$: on trouver une borne supérieure sur le nombre d'opérations effectuées pour une entrée quelconque de taille n.

— Sous la forme d'un Ω : on étudie une famille particulière d'entrées $(e_n)_{n\in\mathbb{N}}$ avec chaque e_n de taille n, et on trouve une borne inférieure sur le nombre d'opérations exécutées.

— Sous la forme d'un Θ : il faut trouver le $\mathcal{O}$ le plus petit possible, et deviner le pire cas de l'algorithme pour trouver le Ω le plus grand possible.

B Calculer la complexité

Voyons comment compter le nombre d'opérations effectuées par un algorithme ou une fonction. Dans une boucle pour, de la forme :

```
1 pour i = 0 à n − 1 faire
2 |  Corps de la boucle;
```

il faut sommer le coût de tous les passages de la boucle. Si l'on note s_i le coût d'un passage i, le coût total de la boucle est $s_0 + s_1 + \cdots + s_{n-1}$. Parfois, le coût des passages ne dépend pas de i, et le coût total de la boucle est simplement n fois le coût du corps de la boucle.

Exercice 2

Donner la complexité asymptotique des fonctions suivantes, sous la forme d'un $\mathcal{O}$.

1. Exponentiation simple (donner la complexité en fonction de n) :

```c
int pow(int x, int n){
  int res = 1;
  for (int i = 0; i < n; i++){
    res = res * x;
  }
  return res
}
```

2. Recherche de somme (donner la complexité en fonction de n) :

```c
/* Renvoie true s'il existe deux indices i, j tels que
   T[i] + T[j] = s, false sinon. n est la taille de T */
bool somme(int* T, int n, int s){
  for (int i = 0; i < n; i++){
    for (int j = 0; j < i; j++){
      if (T[i] + T[j] == s){
        return true;
      }
    }
  }
  return false
}
```

3. Longueur d'une chaîne de caractères (donner la complexité en fonction de l la longueur de la chaîne) :

```c
int longueur(char* s){
  int i = 0;
  while (s[i] != '\0'){
    i++;
  }
  return i;
}
```

4. Savoir si une chaîne de caractères est un palindrome :

```
bool est_palindrome(char* s){
  for (int i = 0; i < longueur(s); i++){
    if (s[i] != s[longueur(s) - 1 - i]){
      return false;
    }
  }
  return true;
}
```

5. Comment améliorer la complexité de la fonction `est_palindrome` ?
6. Donner la complexité en fonction de n :

```
bool f(int n){
  for (int i = 0; i < 1999000; i++){
    if (n == i){
      return true;
    }
  }
  return false;
}
```

Pour les boucle tant-que, il faut déterminer le nombre de passages faits par la boucle. Notons que si une boucle B admet un variant V, alors en notant V_0 la valeur de V avant d'entrer dans B, la boucle B fait au plus V_0 passages. On peut essayer de chercher des variants les plus petits possibles. Un variant de boucle idéal décroitrait d'exactement 1 à chaque tour de boucle et atteindrait 0 en sortie.

Exercice 3

Trouver un variant de boucle idéal pour les fonctions suivantes, et en déduire les complexités asymptotiques :

1. Exponentiation rapide (donner la complexité en fonction de n) :

```
int pow(int x, int n){
  int res = 1;
  while (n>0){
    if (n%2 == 1){
      res = x*res;
    }
    x = x*x;
    n = n/2;
  }
  return res;
}
```

2. Racine entière (donner la complexité en fonction de n) :

```
/* Renvoie i maximal tel que i^2 <= n */
int racine(int n){
  int i = 0;
  while ((i+1)*(i+1) <= n){
    i++;
  }
  return i-1;
}
```

4 Complexités usuelles, opérations

A Échelle de complexités

Certaines complexités sont très courantes, et ont un nom particulier. Soit $a > 1$:

- $\mathcal{O}(1)$: complexité ***constante***
- $\mathcal{O}(n)$: complexité ***linéaire***
- $\mathcal{O}(n^2)$: complexité ***quadratique***
- $\mathcal{O}(n^3)$: complexité ***cubique***
- $\mathcal{O}(n^k)$ complexité ***polynomiale***
- $\mathcal{O}(\log_a(n))$: complexité ***logarithmique***
- $\mathcal{O}(a^n)$: complexité ***exponentielle***

Par exemple, l'algorithme d'exponentiation rapide est linéaire, alors que l'exponentiation naïve est linéaire.

Proposition 3

$a, a' \in \mathbb{R}^{+*}$ avec $a' > 1$ et $a > 1$. Alors $\log_a(n) = \Theta(\log_{a'}(n))$.

Démonstration. Découle du fait que pour $n \in \mathbb{N}^*$, $log_a(n) = log_{a'}(n) log_a(a')$. □

La propriété précédente fait que lorsque l'on exprime une complexité qui fait apparaître du logarithme, on peut écrire $\log(n)$ sans se soucier de la base, car tous les logarithmes sont équivalents.

Proposition 4

Soient $a, a' \in \mathbb{R}^{+*}$ avec $1 < a < a'$, et $k, k' \in \mathbb{R}^{+*}$ avec $k < k'$. On peut ordonner les complexités usuelles précédentes comme suit :

$$\mathcal{O}(1) \subseteq \mathcal{O}(\log_a(n)) \subseteq \mathcal{O}(n^k) \subseteq \mathcal{O}(n^{k'}) \subseteq \mathcal{O}(a^n) \subseteq \mathcal{O}(a'^n)$$

De plus, aucun $\mathcal{O}$ n'est un Θ. Par exemple, $a^n \neq O(a'^n)$.

Proposition 5

Soit $k \in \mathbb{N}$.

1. $\sum_{i=1}^{n} i = \Theta(n^2)$
2. $\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$
3. $\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log(n))$
4. $\sum_{i=1}^{n} 2^i = \Theta(2^n)$
5. $\sum_{i=1}^{n} \frac{1}{2^i} = \Theta(1)$
6. $\sum_{i=1}^{n} \log(i) = \Theta(n \log(n))$

Les propriétés précédentes s'utilisent en pratique lorsque l'on calcule la complexité d'un algorithme avec des boucles. Par exemple, si l'on considère le code suivant :

```
1 for (int i = 1; i <= n; i++){
2   for (int j = 0; j < n/i; j++){
3     printf("%d\n", j);
4   }
5 }
```

Chaque passage de la boucle intérieure s'exécute en temps constant $\mathcal{O}(1)$, disons en 5 opérations au plus. Donc à i fixé, le i-ème passage de la boucle extérieure prends un temps au plus $5\frac{n}{i}$. Le code s'exécute donc finalement en temps au plus $5\sum_{i=1}^{n} \frac{n}{i} = \mathcal{O}(n \log n)$, la dernière égalité découlant du point 3 de la propriété précédente.

B Opérations

Proposition 6

Soient u, v, w, t des suites positives. Alors :

- Si $u_n = \mathcal{O}(w_n)$ et $v_n = \mathcal{O}(w_n)$ alors $u_n + v_n = \mathcal{O}(w_n)$
- Si $u_n = \mathcal{O}(w_n)$ et $v_n = \mathcal{O}(t_n)$ alors $u_n v_n = \mathcal{O}(w_n t_n)$
- Si $u_n = \Theta(w_n)$ et $v_n = \Theta(w_n)$ alors $u_n + v_n = \Theta(w_n)$
- Si $u_n = \Theta(w_n)$ et $v_n = \mathcal{O}(w_n)$ alors $u_n + v_n = \Theta(w_n)$

La notation $u_n = \mathcal{O}(v_n)$ peut parfois amener à des raisonnements faux.

Exercice 4

Expliquer pourquoi la preuve suivante que $n = \mathcal{O}(1)$ est fausse :
"Montrons par récurrence que $n = \mathcal{O}(1)$:

— Pour $n = 1$, on a bien $1 = \mathcal{O}(1)$

— Soit $n \in \mathbb{N}$, et supposons que $n = \mathcal{O}(1)$. Alors $n + 1 = \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(1)$ par la propriété précédente. "

En particulier, lorsque l'on évalue la complexité d'un algorithme ou d'une fonction, il faut faire attention à ce que l'on écrit. Le meilleur moyen de ne jamais faire d'erreur est de **revenir à la définition** de $\mathcal{O}$, Ω et Θ dans les cas compliqués.

Exemple 2

On suppose que l'on a défini une fonction `void f(int k, int n)` avec, à k entier fixé, la complexité du calcul de `f(k, n)` en $\mathcal{O}(n)$. On considère la fonction suivante :

```
1 void iterer_f(int n){
2   for (int k=0; k < n; k++){
3     f(k, n); // O(n)
4   }
5 }
```

Autrement dit, on effectue n passages de boucle, chacun en $\mathcal{O}(n)$. On peut alors s'attendre à ce que la fonction `iterer_f` soit en $\mathcal{O}(n^2)$. car on somme simplement les complexités des passages. Cependant, ce n'est pas forcément le cas. Par exemple, si on considère les deux versions de `f` suivantes :

```
1 void f_v1(int k, int n){
2   for (int i=0; i < n; i++){
3     printf("%d\n", i*k);
4   }
5 }
```

```
1 void f_v2(int k, int n){
2   for (int i=0; i < k*n; i++){
3     printf("%d\n", i);
4   }
5 }
```

On considèrera que l'affichage est la seule opération élémentaire, les additions et multiplications étant négligeables.
Pour la première version, `f` fait n opérations, soit bien $\mathcal{O}(n)$ opérations à k fixé. La fonction `iterer_f` effectue alors $n + n + \cdots + n = n^2$ opérations, soit bien $\mathcal{O}(n^2)$.
En revanche, pour la deuxième version, `f` fait nk opérations. Lorsque k est fixé, c'est bien du $\mathcal{O}(n)$, mais `iterer_f` va faire un nombre d'opérations total égal à :

$$\sum_{k=0}^{n-1} nk = n\sum_{k=0}^{n-1} k = n\frac{n(n-1)}{2} = \Omega(n^3)$$

Ainsi, **on ne peut pas sommer les $\mathcal{O}$ n'importe comment**. Le problème dans l'exemple précédent vient du fait que les constantes de domination des suites $(f(n,k))_{n\in\mathbb{N}}$ peuvent dépendre de k. En revanche, lorsque les constantes et indices de domination sont les mêmes, la somme se passe bien :

Proposition 7

Soit $v = (v_n)_{n\in\mathbb{N}}$ une suite d'entiers. Posons, pour $k \in \mathbb{N}$, u^k une suite d'entiers ($u = (u^k)_{k\in\mathbb{N}}$ est donc une suite de suite). On note $u^k = (u^k_n)_{n\in\mathbb{N}}$, et on suppose qu'il existe $C \in \mathbb{R}^{+*}$ et $n_0 \in \mathbb{N}$ tels que $\forall k \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq n_0 \Rightarrow u^k_n \leq v_n$.
Autrement dit, on suppose que chaque u^k est dominée par v avec la même constante et à partir du même rang. Alors :

$$\sum_{k=0}^{n} u^k_n = \mathcal{O}(nv_n)$$

Exercice 5

L'exemple précédent donne un contre-exemple si l'on enlève la contrainte "avec la même constante de domination". Trouver un contre exemple si l'on enlève plutôt la contrainte "à partir du même rang".

5 Étude de complexité : tris par comparaison

On s'intéresse dans toute cette partie au problème classique suivant : étant donné un tableau T d'éléments d'un ensemble totalement ordonné E, trier T dans l'ordre croissant.
Certains algorithmes de tri modifient le tableau en entrée sans utiliser de mémoire additionnelle, et d'autres créent une copie triée du tableau. Nous allons commencer par étudier des algorithmes de la première sorte, dont on dit qu'ils sont **en place**.

Exercice 6

Donner une spécification précise pour les algorithmes de tri en place :

Algorithme 3 : Tri

Entrée(s) : T tableau de taille n
Sortie(s) : ...

Tri par sélection

Le tri par sélection consiste à sélectionner consécutivement les plus grands éléments de T, et de les placer à la fin du tableau :

Algorithme 4 : Indice_maximum

Entrée(s) : T un tableau de taille $n > 0$
Sortie(s) : $i \in [\![0, n-1]\!]$ tel que $T[i]$ est maximal

```
1 i_m ← 0 // indice du maximum courant
2 pour j = 0 à n − 1 faire
3     si T[j] > T[i_m] alors
4         i_m ← j;
5 retourner i_m
```

Algorithme 5 : Tri par sélection

Entrée(s) : T un tableau de taille n
Sortie(s) : Rien, et T est trié dans l'ordre croissant

```
1 pour i = 0 à n − 1 faire
2     j ← indice du maximum de T[0], ..., T[n − 1 − i];
3     Échanger T[n − i − 1] et T[j];
```

Exemple 3

Exécution de l'algorithme sur $T = [4, 1, 7, 7, 2]$.

Terminaison L'algorithme de calcul de l'indice du maximum contient uniquement une boucle pour, donc il termine. De même pour l'algorithme de tri par sélection.

Complexité La recherche de l'indice du maximum prend un temps $\mathcal{O}(k)$, où k est le nombre de cases à inspecter. Elle est même en $\Theta(k)$, car elle fait toujours exactement k passages.
L'algorithme de tri par sélection fait n passages de boucle, et au i-ème passage, lance la recherche du maximum sur $n - i$ cases. Le temps total est donc en $\Theta(n + (n-1) + \cdots + 1) = \Theta(n^2)$.

Correction Montrons que l'algorithme est correct, i.e. qu'en sortie de l'algorithme, T est trié. On réécrit pour cela l'algorithme avec une boucle while, afin de pouvoir introduire un invariant de boucle :

Algorithme 6 : Tri par sélection

Entrée(s) : T un tableau de taille n
Sortie(s) : Rien, et T est trié dans l'ordre croissant

```
1 i ← 0 ;
2 tant que i < n faire
3 |   m ← indice du maximum de T[0], ..., T[n − 1 − i];
4 |   Échanger T[n − i − 1] et T[j];
5 |   i ← i + 1 ;
```

On admet la correction de Indice_maximum (exercice : la démontrer).

On note $x_0 \leq x_1 \leq \cdots \leq x_{n-1}$ les valeurs de T dans l'ordre croissant.

On montre que la propriété suivante P est un invariant de boucle pour la boucle while :

$$P : \begin{cases} \forall j \in [\![n-i, n-1]\!], T[j] = x_j \\ T[0..n-1-i] \text{ contient } x_0, \dots x_{n-1-i}, \text{ dans un ordre quelconque.} \end{cases}$$

Autrement dit, après i passages de boucle, les i plus grandes valeurs de T sont rangées dans l'ordre à la fin de T, et les autres valeurs sont dans les $n-i$ premières cases de T.

Notons i_k, T_k les valeurs de i, T après k passages de boucles, et montrons par récurrence que $\forall k \in \mathbb{N}, P(k)$:

— En entrée de boucle : $i_0 = 0$, et les $n-0$ premières cases de T contiennent bien les n valeurs de T dans un ordre quelconque.

— Supposons $P(k)$ vraie pour un $k \in \mathbb{N}$ quelconque, et considérons un $k+1$-ème passage. On a :

- Pour $j \in [\![n - i_k, n-1]\!], T[j] = x_j$ (par HR)
- $T[0..n-1-i_k]$ contient $x_0, \dots x_{n-1-i_k}$, dans un ordre quelconque. (par HR)
- $i_{k+1} = i_k + 1$
- $T_{k+1}[m] = T_k[n-1-i_k]$, $T_{k+1}[n-1-i_k] = T_k[m]$ où m est l'indice d'un élément minimal de $T[0..n-i-1]$
- $T_{k+1}[j] = T_k[j]$ pour $j \neq m, j \neq n-1-i_k$

De plus, $T_k[m]$ contient la valeur maximale de $T_k[0..n-1-i_k]$, c'est à dire x_{n-1-i_k}. Donc, $T_{k+1}[n-i_k-1] = x_{n-i_k-1}$ et $T_{k+1}[m]$ contient $T_k[n-i_k-1]$. Pour résumer :

- Pour $j \in [\![n - i_{k+1}, n-1]\!], T_{k+1}[j] = x_j$ (l'élément $x_{n-i_{k+1}}$ a été ajouté).
- $T_{k+1}[0..n-i_{k+1}-1]$ contient $x_0, \dots x_{n-i_k+1-1}$, dans un ordre quelconque ($x_{n-i_{k+1}}$ a été extrait, les autres sont toujours dans cette partie du tableau).

Autrement dit, $P(k+1)$ est vraie.

Donc, P est un invariant de boucle, et est vraie en particulier lorsque l'on sort de la boucle, quand $i = n$:

$$\forall k \in [\![0, n-1]\!], T[j] = x_j$$

Autrement dit, T a été trié : l'algorithme est correct.

Tri par insertion

Le tri par insertion consiste à construire un tableau trié de plus en plus grand, en y insérant les valeurs de T une par une. C'est le tri des joueurs de cartes.

Algorithme 7 : Tri par insertion

Entrée(s) : T un tableau de taille n
Sortie(s) : Rien, T est trié

```
1 pour i = 0 à n − 1 faire
2     j ← i ;
3     tant que j > 0 et T[j] < T[j − 1] faire
4         Échanger T[j] et T[j − 1];
5         j ← j − 1;
```

Exemple 4

Exécution de l'algorithme sur le tableau $T = [5, 3, 7, 2, 2, 8]$.

Terminaison A chaque passage dans la boucle for, il y a une boucle while. A chaque fois, j est un variant de boucle : l'algorithme termine.

Complexité L'algorithme fait n passages dans la boucle ***pour***. Le passage d'indice i fait tourner une boucle ***tant que*** avec au plus i passages, chacun en temps constant. Donc chaque boucle ***tant que*** prend un temps $\mathcal{O}(n)$, et donc le temps total est $O(n^2)$. Lorsque T est trié dans l'ordre décroissant au départ, l'algorithme exécute tous les passages de boucle, et prend alors $\Theta(n^2)$ opérations, c'est donc le pire cas, prend alors $\Theta(n^2)$ opérations.

Notons que si T est déjà trié dans l'ordre croissant en entrée de l'algorithme, alors chaque boucle ***tant que*** intérieure fera un passage seulement. La complexité dans ce cas devient linéaire : $\mathcal{O}(n)$. Contrairement au tri par sélection qui avait un temps d'exécution dépendant uniquement de la taille de l'entrée, le tri par insertion s'exécute plus vite sur les entrées qui sont totalement ou même partiellement triées. On dit que le tri par insertion est un ***tri adaptatif***.

Correction

Exercice 7

On commence par étudier la boucle intérieure. Le rôle de cette boucle est de positionner l'élément $T[i]$ au bon endroit dans $T[0..i]$. Pour fonctionner, cette boucle a comme précondition que le sous-tableau $T[0..i-1]$ doit être trié.

Q1. On considère une valeur de i quelconque, et on se place au i-ème tour de la boucle extérieure. On suppose que $T[0..i-1]$ est trié. Donner un invariant de la boucle intérieure, et l'utiliser pour montrer qu'après la boucle intérieure, le tableau $T[0..i]$ est trié.

Q2. Montrer la correction du tri par insertion.

Tri rapide

Les algorithmes de tri vus précédemment sont assez peu utilisés en pratique, car la complexité en $\mathcal{O}(n^2)$ est trop lente. Le tri rapide est un algorithme permettant d'obtenir une complexité en $\mathcal{O}(n \log n)$, bien meilleure. Le principe du tri rapide est le suivant, pour trier un tableau T :

1. Si T est de taille 0 ou 1, il est déjà trié, on ne fait rien.
2. Sinon, on considère l'élément $x = T[0]$, et on **partitionne** le reste de T en deux parties : les éléments $\leq x$ (sauf $T[0]$), et les éléments $> x$.
3. On trie les les deux parties par tri rapide, on les colle pour former une copie triée de T.

L'élément choisi à chaque étape pour séparer le tableau en deux s'appelle le **pivot**. Cet algorithme est notre premier exemple d'algorithme **récursif**.

Exemple 5

Exécution de l'algorithme sur le tableau $T = [3, 1, 8, 5, 1, 6, 7, 6, 3]$, on dessine l'arbre d'appel.

Écrivons le pseudo-code correspondant :

Algorithme 8 : Tri_rapide

Entrée(s) : T un tableau de n éléments
Sortie(s) : T' copie triée de T

```
si n ≤ 1 alors
    retourner une copie de T
sinon
    p ← 0 // indice du pivot
    T_≤ ← [T[i] | i ∈ ⟦0, n−1⟧, i ≠ p, T[i] ≤ T[p]];
    T_> ← [T[i] | i ∈ ⟦0, n−1⟧, i ≠ p, T[i] > T[p]];
    Trier récursivement T_≤ et T_>;
    T' ← concaténation de T_≤, [x] et T_> retourner T'
```

La construction de $T_\leq$ et $T_>$ se fait en temps linéaire en n, et la concaténation aussi. Notons C_n le temps nécessaire pour trier un tableau de taille n. On aurait donc une relation de récurrence du style :

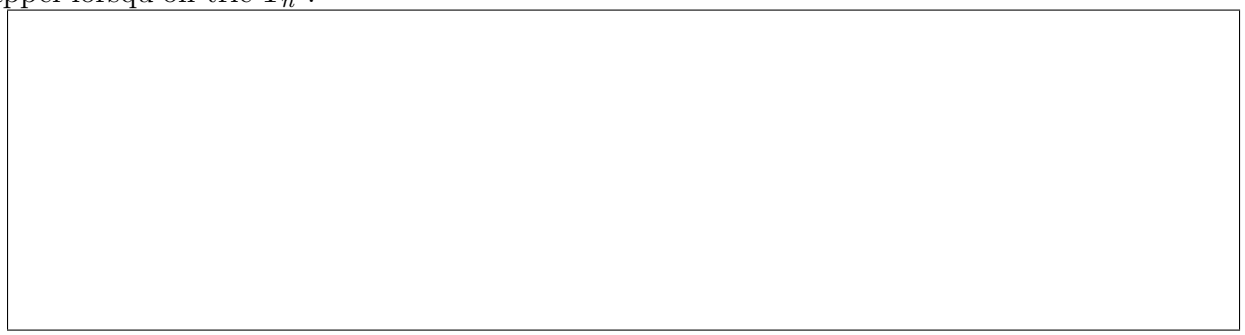
$$C_n = C_{n_1} + C_{n-1-n_1} + \Theta(n) + \text{coût du choix du pivot}$$

où n_1 est la taille de $T_\leq$ (et donc $n - 1 - n_1$ est la taille de $T_>$).

Dans la version du tri rapide étudiée jusqu'à présent, on choisit toujours le premier élément comme pivot, mais rien n'empêche de choisir un autre élément : prendre un élément aléatoire, chercher un "bon pivot", etc... Intuitivement, en s'inspirant de la recherche par dichotomie, on peut penser qu'un pivot est bon s'il sépare le tableau en deux parties égales, et mauvais s'il génère deux parties très déséquilibrées.

Exemple 6

Supposons que l'on choisit toujours le premier élément du tableau comme pivot (i.e. $p \leftarrow 0$ dans l'algorithme). Ce choix de pivot est très rapide : il se fait en temps constant $\mathcal{O}(1)$. Cependant, il peut générer deux sous-tableaux totalement déséquilibrés.
On considère la famille de tableaux $(T_n)_{n\in\mathbb{N}}$ avec $T_n = [0, \dots, n-1]$. Dessinons l'arbre d'appel lorsqu'on trie T_n :



Si on note D_n le temps d'exécution de l'algorithme sur T_n, la relation de récurrence devient :

$$D(n) = D(n-1) + \Theta(n)$$

Cette équation se lit : "pour trier T_n, je dois trier T_{n-1} et faire un nombre linéaire d'opérations en plus.
En déroulant la définition de Θ, la relation devient :

$$\forall n \geq n_0, D(n-1) + Bn \geq D(n) \geq D(n-1) + An$$

Avec n_0, A, B le rang et les constantes de domination. Gardons seulement la deuxième inégalité :

$$\forall n \geq n_0, D(n) \geq D(n-1) + An$$

et donc, en déroulant les inéquations :

$$D(n) \geq A(n + (n-1) + \cdots + n_0) + D(n_0 - 1) \geq An^2 + D(n_0 - 1)$$

Mais $D(n_0 - 1)$ est une constante, et donc finalement $D(n) = \Omega(n^2)$.
Autrement dit, on a trouvé une famille de tableaux sur laquelle, avec ce choix de pivot, l'algorithme prend un temps $\Omega(n^2)$. Avec ce choix de pivot, la complexité asymptotique dans le pire cas sera donc forcément pire que le tri par insertion ou par sélection.

Exercice 8

Supposons maintenant que l'on sait calculer la médiane d'un tableau en $\mathcal{O}(n)$. Alors, en choisissant la médiane comme pivot, on divise le tableau en deux parties de tailles égales (à 1 près), et la relation de récurrence devient alors :

$$C(n) = C(\lfloor \frac{n}{2} \rfloor - 1) + C(\lceil \frac{n}{2} \rceil) + \Theta(n)$$

Pour simplifier, nous allons nous intéresser à l'équation suivante :

$$C(n) = C(\frac{n}{2}) + C(\frac{n}{2}) + \Theta(n) = 2C(\frac{n}{2}) + \Theta(n)$$

Cette équation se lit : "pour trier un tableau de taille n, je trie deux tableaux de taille $\frac{n}{2}$ et je fais un nombre d'opérations linéaire en plus".

Q1. En remplaçant le Θ par sa définition, et en déroulant la relation obtenue, montrer que $C(n) = \mathcal{O}(n \log n)$.

Avec ce choix de pivot, le tri rapide est donc en $\mathcal{O}(n \log n)$. Il faut cependant être capable de calculer la médiane en temps linéaire : vous verrez l'année prochaine un algorithme qui permet de trouver un pivot assez proche de la médiane pour que le tri rapide soit en $\mathcal{O}(n \log n)$.

En pratique, on peut se contenter de choisir le pivot aléatoirement, ou même de toujours prendre le premier élément : on peut montrer qu'**en moyenne**, le temps d'exécution reste $\mathcal{O}(n \log n)$.

Tri rapide en place L'algorithme proposé pour le tri rapide n'est pas en place car il crée une copie triée du tableau donnée en entrée. On peut modifier l'algorithme pour qu'il fasse tous les calculs en place, directement dans T (voir TP).

6 Complexité des problèmes

Définition 2

Un ***problème de décision*** est un couple (P, K) où P est un ensemble, appelé ensemble d'***instances***, et $K \subseteq P$ est un sous-ensemble d'instances, appelé ensemble des ***instances positives***.

Informellement, un problème de décision est une question Oui/Non à paramètres. P est l'ensemble des paramètres, aussi appelé l'ensemble des **instances**, et K est l'ensemble des instances pour lesquels la réponse est "Oui". On notera souvent simplement P pour désigner le problème (P, K) lorsqu'il n'y a pas d'ambiguïté.

Exemple 7

Voici quelques exemples de problèmes :

- $P = \mathbb{N}$, $K = \{n \in \mathbb{N} | n \text{ est premier}\}$ est le problème de primalité **PRIME** : Étant donné $n \in \mathbb{N}$, n est-il premier ?
- $P = \{\text{tableaux de réels}\}$, $K = \{\text{tableaux de réels triés par ordre croissant}\}$ est le problème de savoir si un tableau donné est trié
- $P = \{(t, m) | t \text{ et } m \text{ chaînes de caractères}\}$, $K = \{(t, m) \in P | t \text{ contient } m\}$ est le problème de savoir si un texte contient un motif donné.

Définition 3

Soit (P, K) un problème, et A un algorithme renvoyant un booléen. On dit que A résout P si pour toute instance $I \in P$, A renvoie 1 quand on l'exécute sur I si et seulement si $I \in K$. Autrement dit, l'algorithme détermine si une instance est positive ou négative.

On peut classifier les problèmes selon la complexité des algorithmes qui les résolvent :

Définition 4

Soit $f : \mathbb{N} \to \mathbb{R}^+$ On note $\mathbf{DTIME}(f(n))$ l'ensemble des problèmes ayant un algorithme de résolution en $O(f(n))$.

Exemple 8

On considère le problème **Trié** : "Étant donné un tableau T de taille n, T est-il trié ?" Nous avons vu une méthode naïve en $\mathcal{O}(n^2)$ de résolution de **Trié**, donc **Trié** $\in \mathbf{DTIME}(n^2)$. Nous avons aussi vu un algorithme plus efficace en $\mathcal{O}(n)$, donc **Trié** $\in \mathbf{DTIME}(n)$.

Définition 5

La classe de problèmes **P** est la classe des problèmes pouvant se résoudre en temps polynomial :

$$\mathbf{P} = \bigcup_{k=0}^{+\infty} \mathbf{DTIME}(n^k)$$

Explicitement, un problème est dans **P** s'il existe un entier $k \in \mathbb{N}$ et un algorithme en $\mathcal{O}(n^k)$ qui résout ce problème.

Il existe des problèmes dont on ne connait pas la complexité. Par exemple, le problème des sacs équilibrés :

"Étant donné un poids limite M et des objets de poids $w_1, \dots w_n$, peut on les séparer en deux sacs tels que chaque sac a un poids total inférieur à M ?"

On ne connaît pas à ce jour d'algorithme polynomial permettant de résoudre ce problème. Si vous en trouvez un, vous avez résolu $\mathbf{P} = \mathbf{NP}$, un des plus grands problèmes en informatique, et vous êtes maintenant millionnaire (la suite en 2ème année).