Chapitre 3

Analyse d'algorithmes

1 Introduction par l'exemple

1.1 En itératif

Cas d'une boucle répétitive indexée (boucle `for`)

Exemple 1. Considérons l'algorithme itératif suivant calculant $n!$:

```
let factorielle1 n =
    let p = ref 1 in
        for k = 1 to n do
            p := (!p) * k
        done;
   !p
;;
```

Preuve de correction :
Nous pouvons commencer par vérifier la correction de notre algorithme pour pour $n = 5$ par exemple :

k						
p	1					

Nous n'allons pas nous contenter de cette illustration, nous allons mettre en place une **preuve de correction** :

- Si $n = 0$, alors `factorielle1 0` nous conduit à la séquence `p <- 1` et nous renvoie la valeur de p c'est-à-dire $1 = 0!$.
- Si $n \in \mathbb{N}^*$, alors nous allons décrire la situation à la fin de chaque itération (la "situation" étant souvent l'état des différentes variables). Nous voulons démontrer, pour tout $i \in [\![1; n]\!]$, que

  $$(\mathcal{H}_i) : \text{"à la fin de la } i\text{-ième itération, } k \text{ contient } i \text{ et } p \text{ contient } i!\text{"}.$$

  Cette description est appelée **invariant de boucle**. Montrer la correction de notre algorithme dans ce cas revient à montrer la correction de l'**invariant de boucle** par récurrence.

  Ini. p est initialisé à 1 (`p <- 1`). Ensuite, k prend la valeur 1 (`k <- 1`) et p prend la valeur $!p * k$ (`p <- 1 × 1`). Et donc à la fin de la première itération k contient 1 et p contient 1! donc $(\mathcal{H}_1)$ est vraie.

  Héré. Soit $i \in [\![2; n]\!]$. Supposons $(\mathcal{H}_{i-1})$ donc avant de commencer la i-ième itération, k contient $i-1$ et p contient $(i-1)!$.
  Nous passons à l'itération suivante. La valeur de k est incrémentée de 1 (d'après la sémantique de "pour") donc `k <- i`. Et p reçoit $!p * k$ donc `p <- (i-1)!× i = i!`. Donc $(\mathcal{H}_i)$ est vraie.

  Ccl. Par le principe de récurrence, pour tout i appartenant à $[\![1; n]\!]$, $(\mathcal{H}_i)$ est vraie.

  A la fin de cette répétitive, soit après n itérations, k contient n et p contient $n!$.

Dans les deux cas, après la répétitive, p contient $n!$. Nous renvoyons alors le contenu de p, à savoir $n!$, le résultat attendu, donc notre algorithme est correct.

Preuve de terminaison :
Seule une instruction répétitive attirera notre attention. Nous sommes en présence d'une **répétitive indexée** donc notre algorithme se termine.

Complexité :
Nous cherchons à évaluer la complexité **temporelle** de notre algorithme en dénombrant le nombre $T(n)$ d'opérations effectuées. Nous devons choisir :

- Une **taille de données**, c'est-à-dire une ou plusieurs données représentatives du temps d'exécution de l'algorithme. Ici nous choisissons n (la seule donnée).
- Nos **opérations fondamentales**, c'est-à-dire des opérations représentatives du travail réalisé par l'algorithme. Nous choisissons ici la **multiplication** (tout ceci est très subjectif...).

L'instruction `let p = ref 1 in` n'amène aucune multiplication. Il y a ensuite deux cas :

- Si $n = 0$, alors la répétitive ne s'exécute pas donc aucune multiplication.
- Si $n \in \mathbb{N}^*$, étudions alors la répétitive. Pour tout $k \in [\![1; n]\!]$, l'instruction `p := !p*k` amène 1 multiplication et donc la répétitive amène $\sum_{k=1}^{n} 1 = n$ multiplications.

Dans tous les cas, nous avons effectué n multiplications et l'instruction $!p$ n'amène aucune multiplication.
Ainsi, $T(n) = n = O(n)$. On dit que la **complexité** est **linéaire**.

Exemple 2. Considérons l'algorithme itératif suivant de recherche de l'emplacement du maximum parmi les valeurs d'un ensemble ordonné contenues dans un tableau :

```
let maximum t =
    let n = Array.length t and p = ref 0 in
        for k = 1 to (n-1) do
            if t.(k) > t.(!p) then
                p := k
        done;
   !p
;;
```

Prouver sa correction, sa terminaison et étudier sa complexité.

Cas d'une répétitive conditionnelle (boucle `while`)

Exemple 3. Considérons l'algorithme itératif suivant permettant de chercher un élément x dans un tableau t :

```
let recherche t x =
    let n = Array.length t and i = ref 0 in
        while (!i < n) && (t.(!i) <> x) do
            i := !i+1
        done;
    (!i < n)
;;
```

Preuve de correction :
Proposons, pour tout $k \in [\![1; n]\!]$, l'invariant de boucle :

$(\mathcal{H}_k)$: à la fin de la k-ième itération, i contient k et x n'appartient pas aux valeurs $\{t.(0), ..., t.(k-1)\}$".

Montrons la correction de l'invariant de boucle par récurrence :

Ini. Si on fait une première itération, c'est que la condition $t.(0) \neq x$ est vraie donc x n'appartient pas à $\{t.(0)\}$. Donc $(\mathcal{H}_1)$ est vraie.

Héré. Soit $k \in [\![2; n]\!]$. Supposons $(\mathcal{H}_{k-1})$, c'est-à-dire au début de la k-ième itération, i contient $k-1$ et x n'appartient pas à $\{t.(0), ..., t.(k-2)\}$.

Alors, en supposant les tests vérifiés $!i < n$ et $t.(!i) \neq x$, $t.(k-1)$ est donc différent de x. Et comme x n'appartient pas à $\{t.(0), ..., t.(k-2)\}$, alors x n'appartient pas à $\{t.(0), ..., t.(k-1)\}$, et i reçoit $i+1$ donc i contient k.

Ainsi, à la fin de la k-ième itération, i contient k et x n'appartient pas à $\{t.(0), ..., t.(k-1)\}$ donc $(\mathcal{H}_k)$ est vraie.

Ccl. Par le principe de récurrence, pour tout k appartenant à $[\![1; n]\!]$, $(\mathcal{H}_k)$ est vraie et la condition `while (!i<n) && (t.(!i) <> x)` n'est plus vérifiée à partir d'un nombre fini d'itérations.

Finalement, deux cas se présentent :

- soit $(!i < n)$ et $t.(!i) = x$: x apparaît alors dans t et le booléen $(!i < n)$ est évalué à `true`.
- soit $(!i \geq n)$ et à la fin de la n-ième itération, i contient n et x n'appartient pas à $\{t.(0), ..., t.(n-1)\}$, donc x n'apparaît pas dans le tableau et le booléen $(!i < n)$ est évalué à `false`.

Donc notre algorithme est correct.

Preuve de terminaison :
La preuve est basée sur le fait qu'il n'existe pas de suite strictement décroissante d'entiers naturels, c'est-à-dire que $(\mathbb{N}, \leq)$ est un ordre bien fondé.

Introduisons la suite des $(n-i)$:

- $(n-i)$ est une suite d'entiers et la condition $(!i < n)$ suivie de $i :=!i+1$ majore les valeurs de i par n donc $n - i \in \mathbb{N}$.
- $(n-i)$ est une suite strictement décroissante car

$$i_{\text{suivant}} \leftarrow i_{\text{précédent}} + 1 \text{ donc } (n-i)_{\text{suivant}} - (n-i)_{\text{précédent}} = -i_{\text{suivant}} + i_{\text{précédent}} = -1.$$

La suite $(n-i)$ est donc une suite strictement décroissante d'entiers naturels. Comme $(\mathbb{N}, \leq)$ est un ordre bien fondé, l'algorithme se termine en un nombre fini d'étapes.

Complexité :
Choisissons comme **taille de données** n la taille du tableau et comme **opérations fondamentales** les comparaisons.

Nous ne connaissons pas le nombre d'itérations, nous pouvons alors par exemple nous placer dans le pire des cas (**complexité au pire**). Dans le pire des cas, x n'appartient pas au tableau, nous effectuons alors n itérations avec à chaque fois 2 comparaisons.

En notant $T(n)$ notre complexité, on obtient $T(n) = 2n = O(n)$, donc la **complexité** est **linéaire**.

1.2 En récursif

Exemple 4. Considérons l'algorithme récursif suivant calculant $n!$:

```
let rec factorielle2 n = match n with
    | 0 -> 1
    | _ -> n*(factorielle2 (n-1))
;;
```

Preuve de correction :
Nous singeons la définition mathématique :

$$n! = \begin{cases} 1 & \text{si } n = 0, \\ n \times (n-1)! & \text{si } n \geq 1. \end{cases}$$

Donc l'algorithme est forcément correct.

Preuve de terminaison :
Pour tout $n \in \mathbb{N}^*$, `factorielle2 n` amène un appel récursif sur la donnée $(n-1)$ et $n-1 < n$. Ainsi, l'ordre usuel sur $\mathbb{N}$ étant bien fondé (autrement dit, il n'existe pas de suite strictement décroissante), notre algorithme se termine.

Complexité :
Choisissons n comme **taille de données** et la multiplication comme **opération fondamentale**. Notons $T(n)$ la complexité de notre algorithme.

$$T(0) = 0 \quad \text{et} \quad \forall n \in \mathbb{N}^*,\ T(n) = T(n-1) + 1.$$

C'est une suite arithmétique de raison 1 et de premier terme 0 donc $T(n) = n = O(n)$ et la **complexité** est **linéaire**.

2 Terminaison d'un algorithme

2.1 Ensemble ordonné

Définition.

Soit E un ensemble non vide.
On appelle **relation binaire** sur E, toute application $\mathcal{R}$ de $E \times E$ dans $\mathbb{B} = \{0; 1\}$.

Exemples

1. Soit E un ensemble non vide, la relation d'égalité sur E est une relation binaire sur E.
2. La relation de comparaison usuelle $\leq$ est une relation binaire sur $\mathbb{R}$, notée $\leq$.
3. La relation de divisibilité est une relation binaire sur $\mathbb{N}$ notée $|$.

Remarque. De manière générale, nous utiliserons la **notation infixe** pour une relation binaire.

Définition.

Soit E un ensemble non vide, $\mathcal{R}$ une relation binaire sur E.

- La relation $\mathcal{R}$ est dite **réflexive** sur E si :
$$\forall x \in E, \quad x \mathcal{R} x.$$
- La relation $\mathcal{R}$ est dite **anti-symétrique** sur E si :
$$\forall (x, y) \in E^2, \quad (x \mathcal{R} y \text{ et } y \mathcal{R} x) \Rightarrow (x = y).$$
- La relation $\mathcal{R}$ est dite **transitive** sur E si :
$$\forall (x, y, z) \in E^3, \quad (x \mathcal{R} y \text{ et } y \mathcal{R} z) \Rightarrow (x \mathcal{R} z).$$

Dans le cas où $\mathcal{R}$ vérifie ces trois points, on dit que $\mathcal{R}$ est une **relation d'ordre** sur E et que $(E, \mathcal{R})$ est un ensemble ordonnée.

Exemples.

1. $\mathbb{R}$ muni de la relation de comparaison usuelle $\leq$ est un ensemble ordonné.
2. L'ensemble $\mathbb{N}^*$ muni de la relation de divisibilité est un ensemble ordonné.
   Mais c'est faux dans $\mathbb{Z}$!
3. Définissons sur $\mathbb{N}^2$ une relation binaire par :
$$\forall (a_1, a_2) \in \mathbb{N}^2,\ \forall (b_1, b_2) \in \mathbb{N}^2, \quad (a_1, a_2) \leq (b_1, b_2) \Leftrightarrow (a_1 < b_1) \text{ ou } (a_1 = b_1 \text{ et } a_2 \leq b_2).$$
   Cette relation d'ordre est appelée l'**ordre lexicographique** sur $\mathbb{N}^2$, et se note $\leq_{\text{lex}}$. On peut remarquer que Python applique cet ordre sur les couples.

Définition.

Soit $(E, \leq)$ un ensemble ordonné.
Nous définissons sur E un **ordre strict** noté $<$ par :
$$\forall (x, y) \in E^2, \quad x < y \Leftrightarrow (x \leq y \text{ et } x \neq y).$$

2.2 Ordre total, ordre partiel

Définition.

Soit $(E, \leq)$ un ensemble ordonné.
On dit que deux éléments x et y de E sont **comparables** si $x \leq y$ ou $y \leq x$.

Exemples.

1. Dans l'ensemble $(\mathbb{N}, \leq)$, deux entiers naturels sont toujours comparables.
2. Dans l'ensemble ordonné $(\mathbb{N}^*, |)$, 4 et 7 ne sont pas comparables.

Définition.

Soit $(E, \leq)$ un ensemble ordonné.

- On dit que $(E, \leq)$ est un ensemble **totalement ordonné** si deux éléments de E sont toujours comparables.
- On dit que $(E, \leq)$ est un ensemble **partiellement ordonné** s'il n'est pas totalement ordonné.

Exemples.

1. $(\mathbb{N}, \leq)$ et $(\mathbb{N}^2, \leq_{\text{lex}})$ sont totalement ordonnés.
   L'ordre lexicographique est un ordre total : les mots (à deux lettres ici) sont totalement ordonnés dans le dictionnaire.
2. $(\mathbb{N}^*, |)$ est partiellement ordonné.

2.3 Plus petit élément, élément minimal

Définition.

Soit $(E, \leq)$ un ensemble ordonné.

- On appelle **plus petit élément** de E tout élément m tel que :
$$m \in E \quad \text{et} \quad \forall x \in E,\ m \leq x.$$
- On appelle **élément minimal** de E tout élément m tel que :
$$m \in E \quad \text{et} \quad \forall x \in E,\ (x \leq m \Rightarrow x = m)$$

Remarque. On peut prouver que s'il existe un plus petit élément alors il n'y a qu'un seul élément minimal (et c'est le plus petit élément) :

En revanche, s'il n'existe pas de plus petit élément, alors il peut y avoir plusieurs éléments minimaux.

Exemple.

- Déterminer les éléments minimaux de l'ensemble $(\mathbb{N} \setminus \{0;1\}, |)$.

- Déterminer les plus petits éléments de l'ensemble $(\mathbb{N} \setminus \{0;1\}, |)$.

Propriété 1 (Élément minimal et plus petit élément)

Soit $(E, \leq)$ un ensemble ordonné.

(1) Si E admet un **plus petit élément** m, alors m est un **élément minimal** de E.

(2) En supposant de plus $(E, \leq)$ **totalement ordonné**, alors tout **élément minimal** de E est un **plus petit élément** de E.

Preuve.

□

2.4 Ensemble bien ordonné, ensemble bien fondé

Définition.

Soit $(E, \leq)$ un ensemble ordonné.

1. On dit que $(E, \leq)$ est **bien ordonné** si toute partie non vide de E admet un plus petit élément.
2. On dit que $(E, \leq)$ est **bien fondé** si toute partie non vide de E admet un élément minimal.

Remarques. On déduit de la propriété 1 que :

1. Un ensemble bien ordonné est aussi bien fondé.
2. Un ensemble **totalement ordonné** est bien ordonné si et seulement si il est bien fondé.

Exemples.

1. $(\mathbb{N}, \leq)$ est bien ordonné et bien fondé.
2. $(\mathbb{N}^2, \leq_{\text{lex}})$ est bien ordonné et bien fondé.
3. $(\mathbb{N} \setminus \{0; 1\}, |)$ est bien fondé.

Théorème 2 (Caractérisation d'un ensemble bien fondé)

Soit $(E, \leq)$ un ensemble ordonné. Les assertions suivantes sont équivalentes :

(1) $(E, \leq)$ est bien fondé.

(2) Il n'existe pas de suite strictement décroissante d'éléments de E.

Preuve.

□

2.5 Preuves de terminaison

Nous est donné un algorithme qui, à partir des données, nous fournit des résultats. Nous espérons obtenir un résultat en un temps fini. Un algorithme étant une suite d'instructions, nous avons à montrer que cette suite est finie.

En itératif

Seules les instructions répétitives attireront notre attention.

Afin de montrer qu'une répétitive correspond à un nombre fini d'instructions, nous allons construire une suite (u_n) strictement décroissante d'éléments d'un ensemble $(E, \leq)$ bien fondé telle que chaque itération de la répétitive nous fasse passer au terme suivant de la suite (u_n).

L'ordre sur E étant bien fondé, une telle suite ne peut comporter qu'un nombre fini de termes : notre répétitive amènera un nombre fini d'itérations.

Exemple. Nous avons fait ce type de raisonnement dans l'exemple 3 avec l'algorithme itératif `recherche` permettant de chercher un élément x dans un tableau t.

Remarques.

1. Les répétitives indexées ne demandent pas de preuve de terminaison.
2. En cas de répétitives imbriquées, le travail commencera sur la répétitive intérieure.

En récursif

L'idée est semblable en construisant une suite strictement décroissante d'un ensemble bien fondé $(E, \leq)$ telle que chaque appel récursif nous fasse passer au terme suivant.

Exemples.

1. Nous avons fait ce type de raisonnement dans l'exemple 4 avec l'algorithme itératif `factorielle2` permettant de calculer la factorielle d'un entier naturel.
2. Rappelons l'algorithme récursif du pgcd de deux entiers naturels vu au chapitre 2 :

```
let rec pgcd a b = match b with
    | 0 -> a
    | _ -> if (a>=b) then pgcd (a-b) b
                  else pgcd b a
;;
```

Exemple. $12 \wedge 5 = 7 \wedge 5 = 2 \wedge 5 = 5 \wedge 2 = 3 \wedge 2 = 1 \wedge 2 = 2 \wedge 1 = 1 \wedge 1 = 0 \wedge 1 = 1 \wedge 0 = 1$.

Prouver la terminaison de cet algorithme.

3 Correction d'un algorithme

3.1 Le principe d'induction

Définition.

Un prédicat sur un ensemble E est une application $\mathcal{P}$ de E dans les booléens.

Remarque. Pour un prédicat $\mathcal{P}$ donné et un $x \in E$ qui le vérifie, on écrira de préférence "$\mathcal{P}(x)$" au lieu de "$\mathcal{P}(x)$ est vrai" (qui est un pléonasme).

On peut alors énoncer le principe d'induction :

Théorème 3 (Induction simple)

Soit $(E, \leq)$ un ensemble **bien fondé**.
Soit A une partie non vide de E et $\varphi : E \setminus A \to E$ telle que : $\forall x \in E \setminus A$, $\varphi(x) < x$.
Si un prédicat $\mathcal{P}$ sur E vérifie :

- Initialisation : $\forall x \in A$, $\mathcal{P}(x)$;
- Hérédité : $\forall x \in E \setminus A$, $\mathcal{P}(\varphi(x)) \Rightarrow \mathcal{P}(x)$,

alors, pour tout $x \in E$, $\mathcal{P}(x)$.

Remarque. En appliquant le principe d'induction simple à $(\mathbb{N}, \leq)$ avec $A = \{0\}$ et $\varphi : \left\{ \begin{array}{ccc} \mathbb{N}^* & \to & \mathbb{N} \\ n & \mapsto & n-1 \end{array} \right.$, on obtient le principe de récurrence simple :

> Si un prédicat $\mathcal{P}$ sur $\mathbb{N}$ vérifie :
>
> - Initialisation : $\mathcal{P}(0)$;
> - Hérédité : $\forall n \in \mathbb{N}^*$, $\mathcal{P}(n-1) \Rightarrow \mathcal{P}(n)$,
>
> alors, pour tout $n \in \mathbb{N}$, $\mathcal{P}(n)$.

L'induction simple est donc une généralisation de la récurrence simple à n'importe quel ensemble $(E, \leq)$ bien fondé.

Preuve.

□

Théorème 4 (Induction forte)

Soit $(E, \leq)$ un ensemble **bien fondé** et M l'ensemble de ses éléments minimaux.
Si un prédicat $\mathcal{P}$ sur E vérifie :

- Initialisation : $\forall x \in M,\ \mathcal{P}(x)$;
- Hérédité : $\forall x \in E \setminus M,\ (\forall y < x,\ \mathcal{P}(y)) \Rightarrow \mathcal{P}(x)$,

alors, pour tout $x \in E$, $\mathcal{P}(x)$ est vraie.

Remarque. En appliquant le principe d'induction forte à $(\mathbb{N}, \leq)$, on obtient le principe de récurrence forte :

> Si $\mathcal{P}$ un prédicat sur $\mathbb{N}$ vérifie :
>
> - Initialisation : $\mathcal{P}(0)$;
> - Hérédité : $\forall n \in \mathbb{N}^*,\ (\forall k \in [\![0; n-1]\!],\ \mathcal{P}(k)) \Rightarrow \mathcal{P}(n)$,
>
> alors, pour tout $n \in \mathbb{N}$, $\mathcal{P}(n)$.

L'induction forte est donc une généralisation de la récurrence forte à n'importe quel ensemble $(E, \leq)$ bien fondé.

Preuve.

□

Exemple. Montrer que tout entier naturel $n \geq 2$ admet une décomposition en produit de facteurs premiers.

3.2 Preuves de correction

Afin de valider notre algorithme, nous devons montrer qu'à partir des données fournies, nous obtenons les résultats escomptés.

En itératif

Un algorithme itératif est une séquence finie d'instructions se succédant avec ou sans condition.

Les instructions simples et conditionnelles ne demandent pas réellement de preuve de correction, il nous suffit de les réaliser avec discussion éventuelle.

Exemple. Voici un algorithme affichant le maximum de trois nombres :

```
let max3 a b c =
    let m = ref a in
        if (b < !m) then  m := b;
        if (c < !m) then m := c
    !m
;;
```

Prouver la correction de cet algorithme.

Reste les instructions répétitives qui elles vont demander une attention particulière. Une instruction répétitive est une instruction qui se répète un nombre fini de fois et, afin de valider cette instruction répétitive, nous allons valider chaque instruction qui se répète en décrivant, à chaque itération, l'état courant.

Nous adopterons le **principe d'induction** et la description de l'état courant sera appelé un **invariant de boucle**. Nous avons deux approches de la théorie des **invariants de boucle** :

1. Nous réfléchissons à un algorithme puis nous mettons en place un invariant de boucle afin de prouver notre algorithme.
2. Nous réfléchissons à un invariant de boucle puis nous en déduisons l'algorithme.

Exemple.

1. Voici un algorithme calculant la puissance entière d'un entier :

```
let puissance a n =
    let p = ref 1 in
        for k = 1 to n do
            p := !p * a
        done;
    !p
;;
```

   Prouver la correction de cet algorithme.

2. Considérons un mot m et un texte a et cherchons à écrire un algorithme qui détermine l'appartenance de m à a. Nous choisissons de représenter a et m par des chaînes de caractères (de type `String` en OCaml).

   Donner un invariant de boucle pour répondre au problème puis en déduire l'algorithme demandé.

En récursif

La preuve de correction d'un algorithme récursif nous conduit à singer des preuves par récurrence ou des preuves par induction.

Exemple. Reprenons l'algorithme calculant récursivement le pgcd de deux entiers naturels :

```
let rec pgcd a b = match b with
    | 0 -> a
    | _ -> if (a>=b) then pgcd (a-b) b
                     else pgcd b a
;;
```

L'algorithme est basé sur les trois propriétés mathématiques suivantes :

$$\begin{cases} \forall a \in \mathbb{N},\ a \wedge 0 = a \\ \forall (a,b) \in \mathbb{N}^2,\ a \wedge b = b \wedge a \\ \forall (a,b) \in \mathbb{N}^2,\ a \geq b \Rightarrow (a-b) \wedge b = a \wedge b \end{cases}$$

Prouver la correction de cet algorithme.

4 Complexité d'un algorithme

4.1 Généralités

La qualité d'un programme dépend essentiellement de deux choses :

- son temps d'exécution qui dépend de :
  - de la structure des données du problème pour ce programme,
  - de la qualité du code engendré par le compilateur,
  - de la rapidité des instructions offerte par l'ordinateur,
  - de l'efficacité de l'algorithme,
  - de la qualité de la programmation.
- son utilisation de l'espace mémoire (qui influe aussi sur le temps d'exécution : les temps d'accès).

L'idée de la suite est de s'affranchir de considérations subjectives (matériel, programme, ...) et de nous centrer sur l'algorithme en lui-même. Nous allons donc nous limiter dans un premier temps à :

- la structure des données,
- l'efficacité de l'algorithme,
- l'espace mémoire utilisé.

Le mot **complexité** recouvre en fait deux réalités :

- la complexité des algorithmes : c'est l'étude de l'efficacité comparée d'algorithmes réalisant la même tâche, nous mesurons le temps d'exécution et aussi l'espace mémoire utilisé de chaque algorithme,
- la complexité des problèmes : c'est la classification des problèmes en fonction des performances des meilleurs algorithmes connus à ce jour qui les résolvent. Citons le problème du voyageur de commerce qui a une telle complexité que nous pouvons le considérer comme non résolu.

Notre exposé se limite à la complexité des algorithmes et plus particulièrement à leur **complexité temporelle** (temps d'exécution), même si comme nous l'avons déjà dit, la **complexité spatiale** (occupation mémoire) peut influer sur la complexité temporelle (temps d'accès) .

4.2 Mise en place de la complexité d'un algorithme

La complexité temporelle de notre algorithme va dépendre des données manipulées, de leur structure et de leur taille.

Le choix de la structure des données se fera par bon sens. Par exemple, pour effectuer la division euclidienne de deux entiers, nous manipulerons des variables de type entier plutôt que des variables de type flottant... La structure des données manipulée ne sera pas évoquée dans la mise en place de la complexité de l'algorithme.

Nous ferons par contre attention à la **taille de données**, c'est-à-dire la taille des objets manipulés. Il est assez difficile de donner une définition théorique de la notion de taille de données. Citons plutôt quelques situations classiques afin d'appréhender cette notion :

Algorithme	Taille de données
opération sur les nombres	valeur maximale des nombres
calcul de sommes, de produits	nombre de termes de la somme, du produit
polynôme	degré, valeur maximale des coefficients
tableau	nombre de cases

La taille des données n étant choisie, nous allons maintenant chercher à évaluer le temps d'exécution $T(n)$ de notre algorithme en dénombrant tout simplement, le nombre d'**opérations fondamentales** que l'algorithme réalise. Cette notion d'opérations fondamentales est comme la notion de taille de données, difficile à définir et est associer aux types de problème étudié. Voici quelques exemples :

Type de données	Opération(s) fondamentale(s)
opérations sur les nombres	addition, multiplication, ...
recherche d'un élément dans un tableau	comparaison
tri des éléments d'un tableau	comparaison, échange

Une fois choisie(s), nous déciderons que les opérations fondamentales ont toutes le même coût.

4.3 Différents types de complexité

Nous rencontrerons trois types de complexité :

- la complexité **au pire** : évaluation de la complexité dans le pire des cas, synonyme de sécurité,
- la complexité **au mieux** : évaluation de la complexité dans le meilleur des cas, inutile,
- la complexité **en moyenne** : évaluation de la moyenne des complexités dans tous les cas, sans doute représentative mais difficile à évaluer.

4.4 Différentes échelles de complexité

Définition.

Nous avons différents types de complexité, dans un ordre croissant pour une taille de données entière n :

- **logarithmique** : de l'ordre de $O(\log_2(n))$,
- **linéaire** : de l'ordre de $O(n)$,
- **quadratique** : de l'ordre de $O(n^2)$,
- **polynômiale d'ordre** p , $p \in \mathbb{N}$: de l'ordre de $O(n^p)$,
- **exponentielle** : de l'ordre de $O(a^n)$,
- **hyper-exponentielle** : comme $O(n!)$ par exemple .

Les ordinateurs personnels sont actuellement capables d'accomplir 10^{11} opérations par seconde et imaginons des algorithmes qui effectuent un traitement de données dans un temps $T(n)$.

Nous donnons ci-dessous un tableau de temps de traitement que l'on peut espérer pour n éléments suivant la complexité de l'algorithme, où s, m, h, j, a désignent respectivement la seconde, la minute, l'heure, le jour, l'année et u l'âge estimé de l'univers (15 milliards d'années), et où les nombres totalement inimaginables n'ont pas été affichés.

	$\log_2(n)$	n	$n\log_2(n)$	n^2	n^4	2^n	4^n	$n!$
$n=5$	$2,3\cdot10^{-11}\,s$	$5\cdot10^{-11}\,s$	$1,1\cdot10^{-10}\,s$	$2,5\cdot10^{-10}\,s$	$6\cdot10^{-9}\,s$	$3,2\cdot10^{-10}\,s$	$1\cdot10^{-8}\,s$	$1,2\cdot10^{-9}\,s$
$n=10$	$3,3\cdot10^{-11}\,s$	$1\cdot10^{-10}\,s$	$3,3\cdot10^{-10}\,s$	$1\cdot10^{-9}\,s$	$1\cdot10^{-7}\,s$	$1\cdot10^{-8}\,s$	$3\cdot10^{-5}\,s$	$3,6\cdot10^{-5}\,s$
$n=15$	$3,9\cdot10^{-11}\,s$	$1,5\cdot10^{-10}\,s$	$5,8\cdot10^{-10}\,s$	$2,2\cdot10^{-9}\,s$	$5\cdot10^{-7}\,s$	$3,2\cdot10^{-7}\,s$	$1\cdot10^{-2}\,s$	$13\,s$
$n=20$	$4,3\cdot10^{-11}\,s$	$2\cdot10^{-10}\,s$	$8,6\cdot10^{-10}\,s$	$4\cdot10^{-9}\,s$	$1,6\cdot10^{-6}\,s$	$1\cdot10^{-5}\,s$	$11\,s$	$281\,j$
$n=30$	$4,9\cdot10^{-11}\,s$	$3\cdot10^{-10}\,s$	$1,4\cdot10^{-9}\,s$	$9\cdot10^{-9}\,s$	$8,1\cdot10^{-6}\,s$	$1\cdot10^{-2}\,s$	$133\,j$	$6470\,u$
$n=50$	$5,6\cdot10^{-11}\,s$	$5\cdot10^{-10}\,s$	$2,8\cdot10^{-9}\,s$	$2,5\cdot10^{-8}\,s$	$6,2\cdot10^{-7}\,s$	$3,1\,h$	$30u$	
$n=100$	$6,6\cdot10^{-11}\,s$	$1\cdot10^{-9}\,s$	$6,6\cdot10^{-9}\,s$	$1\cdot10^{-7}\,s$	$1\cdot10^{-3}\,s$	$30\,u$		
$n=500$	$8,9\cdot10^{-11}\,s$	$5\cdot10^{-9}\,s$	$4,4\cdot10^{-8}\,s$	$2,5\cdot10^{-6}\,s$	$6,2\cdot10^{-1}\,s$			
$n=1000$	$1\cdot10^{-10}\,s$	$1\cdot10^{-8}\,s$	$1\cdot10^{-7}\,s$	$1\cdot10^{-5}\,s$	$10\,s$			
$n=5000$	$1,2\cdot10^{-10}\,s$	$5\cdot10^{-8}\,s$	$6,1\cdot10^{-7}\,s$	$2,5\cdot10^{-4}\,s$	$1,7\,h$			
$n=10000$	$1,3\cdot10^{-10}\,s$	$1\cdot10^{-7}\,s$	$1,3\cdot10^{-6}\,s$	$1\cdot10^{-3}\,s$	$1\,j$			
$n=50000$	$1,6\cdot10^{-10}\,s$	$5\cdot10^{-7}\,s$	$7,8\cdot10^{-6}\,s$	$2,5\cdot10^{-2}\,s$	$2\,a$			

4.5 Calculs de complexité

Selon le paradigme utilisé la complexité s'obtient :

- comme une somme pour les structures itératives : on additionne le nombre d'opérations à chaque étape de la boucle ;
- comme le terme général d'une suite récurrente pour une fonction récursive.

Une fois établi ce que l'on doit calculer, on est ramené à un exercice de mathématiques. Voici quelques résultats utiles à connaître :

Propriété 5 (Calculs de complexité)

(1) **Sommes usuelles :**

Pour tous $q > 1$, $\alpha > 0$ et $\beta > 0$,

$$\sum_{k=0}^{n} q^k = O(q^{n+1}), \qquad \sum_{k=0}^{n} k^\alpha = O(n^{\alpha+1}), \qquad \sum_{k=1}^{n} k^\alpha (\ln(k))^\beta = O(n^{\alpha+1}(\ln(n))^\beta).$$

(2) **Suites arithmétiques :**

Si (u_n) est une suite arithmétique, alors $u_n = O(n)$.

(3) **Suites arithmético-géométriques :**

Si (u_n) est une suite arithmético-géométrique, telle que pour tout $n \in \mathbb{N}$, $u_{n+1} = au_n + b$ avec $a > 1$, alors $u_n = O(a^n)$.

(4) **Suites dichotomiques :**

Soient $p \in \mathbb{N}^*$, $\lambda \in \mathbb{R}_+$, $\gamma \in \mathbb{R}_+$ et la suite (u_n) vérifiant la relation :

$$\forall n \in \mathbb{N}^*, \quad u_n = pu_{\lfloor \frac{n}{2} \rfloor} + \lambda n^\gamma$$

- Si $\log_2(p) = \gamma$, alors $u_n = O\left(n^\gamma \log_2(n)\right)$.
- Si $\log_2(p) > \gamma$, alors $u_n = O\left(n^{\log_2(p)}\right)$.
- Si $\log_2(p) < \gamma$, alors $u_n = O\left(n^\gamma\right)$.

Preuve.

□

Nous pouvons généraliser ainsi le théorème (admis) :

Théorème 6 (Complexité des algorithmes "diviser pour régner")

Soient a et $b \in \mathbb{N}^*$, k un entier naturel ≥ 2 et $\gamma \in \mathbb{R}_+$. Soit (u_n) une suite telle que, pour tout $n \geq 2$,

$$u_n = au_{\lfloor \frac{n}{k} \rfloor} + bu_{\lceil \frac{n}{k} \rceil} + O\left(n^\gamma\right).$$

- Si $\log_k(a+b) = \gamma$, alors $u_n = O\left(n^\gamma \log(n)\right)$.
- Si $\log_k(a+b) > \gamma$, alors $u_n = O\left(n^{\log_2(a+b)}\right)$.
- Si $\log_k(a+b) < \gamma$, alors $u_n = O\left(n^\gamma\right)$.