Fondements de l'informatique

I. Ensembles

Les notions sur les ensembles sont omniprésentes en informatique.
Voici quelques rappels importants qui permettent de comprendre la structure des raisonnements mathématiques et logiques qui ont donnés naissance à la théorie de l'informatique.

1) Ensembles

Définitions :

- Soit E un **ensemble** et e un **élément** de E.
  On note $e \in E$ pour signifier que e est un élément de l'ensemble E.
- Si A et B sont deux ensembles, on note $A \subset B$ pour signifier que tout élément de A est un élément de B. On dit, dans ce cas que A est une **partie** de B.
  Les parties de E constituent un ensemble que l'on note $\mathcal{P}(E)$.
- On notera $A \cup B$ pour l'**union** des ensembles A et B.
- On notera $A \cap B$ pour l'**intersection** des ensembles A et B.
- Lorsque A est une partie de E on notera A^c pour le **complémentaire** de A dans E.

Remarque :
L'**ensemble vide** est l'ensemble ne contenant aucun élément.

2) Produit cartésien

Définitions :

- On appelle **produit cartésien** de deux ensembles E et F, noté $E \times F$, l'ensemble des **couples** formés d'un élément de E et d'un élément de F :
  $$E \times F = \{(x, y) | x \in E \text{ et } y \in F\}$$
- Pour un entier $n \geqslant 1$, on note $E^n = E \times \ldots \times E$, le produit cartésien de E par lui-même n fois.

3) Application

Définitions :

- Une **fonction** f d'un ensemble E vers un ensemble F est une partie Γ de $E \times F$.
- Son **domaine** est l'ensemble des $x \in E$ tels que $(x, y) \in \Gamma$ pour un certain $y \in F$.
- Son **image** est l'ensemble des $y \in F$ tels que $(x, y) \in \Gamma$ pour un certain $x \in E$.
- Une **application** f d'un ensemble E vers une ensemble F est une fonction dont le domaine est E.
- Une **famille** $(x_i)_{i \in I}$ d'éléments d'un ensemble X est une application d'un ensemble I dans un ensemble X.
  I est appelé l'ensemble des indices, et l'image par cette application de l'élément $i \in I$ est noté x_i.

II. Langages

L'étude des langages est une notion fondamentale pour la théorie de l'informatique.
La linguistique s'est intéressé aux structures de la langue naturelle et en a proposé une modélisation qui a conduit aux concepts de langages formels.
Ces notions s'avèrent fondamentales (de façon anecdotique en algorithmique où l'on rencontre les chaînes de caractères) dans la mise en place de **langages de programmation**.
Il est, en effet, nécessaire d'être capable de communiquer avec la machine.
L'objectif sera donc de créer des langages de programmation et en maîtriser la **syntaxe** (c'est l'objet des notions présentées ci-dessous).
Un enjeu encore plus fondamental sera d'étudier la **sémantique** de ces langages (on mettra alors en place des notions plus évoluées tels que les grammaires).

1) Alphabets

Définitions :
On fixe un ensemble fini Σ que l'on appelle **alphabet**.
Les éléments de Σ sont appelés des **lettres** ou des **symboles**.

Exemples :

- $\Sigma_{bin}=\{0, 1\}$ est l'alphabet binaire.
- $\Sigma_{latin}=\{A, B, C, \dots, Z, a, b, c, \dots, z\}$ est l'alphabet correspondant aux lettres de l'alphabet latin.
- $\Sigma_{nombre}=\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ est l'alphabet correspondant aux chiffres en base 10.
- L'ensemble des caractères ASCII est un alphabet, que l'on peut noter Σ_{ASCII}.
- $\Sigma_{exp}=\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, \times, /, (,)\}$ où +, –, ×, /, (,) désignent des symboles particuliers, est l'alphabet des expressions arithmétiques.

2) Mots

Définitions :
Un **mot** *w* sur un alphabet Σ est une suite finie $w_1 w_2 \dots w_n$ de lettres de Σ.
L'entier *n* est appelé la **longueur** du mot *w*. Il est noté $|w|$

Exemples :

- *010011* est un mot sur l'alphabet Σ_{bin} de longueur 6.
- *9120* est un mot sur l'alphabet Σ_{nombre} mais n'est pas un mot sur l'alphabet Σ_{bin}.
- *Bonjour* est un mot de longueur 6 sur l'alphabet Σ_{latin}.
  azertyu est un mot de longueur 7 sur ce même alphabet.
  Elève n'est pas un mot sur cet alphabet (le symbole *è* n'est pas dans cet alphabet).
- *Eléphant* et *z-è!?'* sont des mots sur l'alphabet Σ_{ASCII}.
- *243+(5×(1+6))* est un mot sur l'alphabet Σ_{exp}.
  24×((((5/+)//+ est aussi un mot sur l'alphabet Σ_{exp}.

3) Langage

Définitions :
Un **langage** sur Σ est un ensemble de mots sur Σ.

Remarques :

- L'ensemble de tous les mots sur l'alphabet Σ est noté Σ^*.
- Le mot vide ϵ est le seul mot de longueur 0.
- Σ^* contient toujours, par définition, le mot vide.

Exemples :

- $\{0, 1\}^*$ désigne l'ensemble des mots sur l'alphabet Σ_{bin}.
  00001101$\in\{0, 1\}^*$. De même, $\epsilon\in\{0, 1\}^*$.
- {*bonjour*, *aurevoir*} est un langage sur Σ_{latin}. Ce langage contient deux mots.
- L'ensemble des mots du dictionnaire du français écrits sans caractères spéciaux (é, à, è, ç, œ, ê, î, ï, ...) est un langage sur l'alphabet Σ_{latin}.
- L'ensemble des mots du dictionnaire du français est un langage sur l'alphabet Σ_{ASCII}.
- Chaque phrase de ce document est un langage sur l'alphabet Σ_{ASCII}. On remarque que le caractère espace blanc (utilisé pour séparer les mots dans une phrase) est un symbole de Σ_{ASCII}.
- Σ_{exp}^* contient des mots comme $4\times((((5/+)//+$ qui ne correspondent à aucune expression arithmétique valide.
  L'ensemble des mots correspondant à une expression arithmétique valide, comme $243+(5\times(1+6))$, est un langage particulier sur Σ_{exp}^*.

4) Concaténation

Définition :
On définit une opération de **concaténation** sur les mots.
La concaténation du mot $u=u_1u_2\ldots u_n$ et du mot $v=v_1v_2\ldots v_m$ est le mot $u.v$ défini par :

$$u.v=u_1u_2\ldots u_nv_1v_2\ldots v_m$$

Remarques :

- L'opération de concaténation notée . est associative, mais non-commutative.
- Le mot vide ϵ est un élément neutre à droite et à gauche de cette opération.
- L'opération de concaténation donne à l'alphabet Σ une structure de monoïde :

$$\Sigma^* = (\Sigma,\ .,\ \epsilon)$$

- En fait tout mot $w_1w_2\ldots w_n$ peut se voir comme $w_1.w_2.\ \ldots\ .w_n$ où w_i représente le mot de longueur 1 réduit à la lettre w_i.
  La confusion entre les mots de longueur 1 et les lettres est souvent pratique.

Exemple :
Soit l'alphabet $\Sigma=\{a, b\}$, alors *aaab* est le mot de longueur 4 dont les trois premières lettres sont *a*, et la dernière est *b*.
C'est aussi la concaténation des quatre mots de longueur 1 : *a*, *a*, *a* et *b*.

Remarque :
Lorsque *i* est un entier, et *w* un mot, on écrit w^i pour le mot obtenu en concaténant *i* fois le mot *w* :

- w^0 est le mot vide ϵ.
- w^1 est le mot *w*.
- $w^{i+1}=w^i.w=w.w^i$ pour tout entier *i*.

Exemple :
aaabbc peut aussi s'écrire a^3b^2c .

Définitions :
Un mot *u* est un **préfixe** d'un mot *w*, s'il existe un mot *z* tel que $w=u.z$.
C'est un **préfixe propre** si $u \neq w$.
Un mot *u* est un **suffixe** d'un mot *w* s'il existe un mot *z* tel que $w=z.u$.

5) Changement d'alphabet

Il est souvent utile de pouvoir réécrire un mot sur un alphabet en un mot sur un autre alphabet.
Par exemple, on a souvent besoin, en informatique de coder en binaire, c'est-à-dire avec l'alphabet $\Sigma_{bin}=\{0, 1\}$. Une façon de faire, pour changer d'alphabet est de procéder par réécriture, lettre par lettre.

Exemple :
Si Σ est l'alphabet $\{a, b, c\}$ et $\Gamma=\{0, 1\}$, alors on peut coder les mots de Σ^* sur Γ^* par la fonction *h* telle que $h(a)=01$, $h(b)=10$, $h(c)=11$.
Le mot *abab* se code alors par $h(abab)=10100110$, soit le mot obtenu en codant lettre par lettre.

Définition :
Soient Σ et Γ deux alphabets.
Un **homomorphisme** est une application de Σ^* dans Γ^* telle que :

- $h(\epsilon)=\epsilon$
- Pour tous mots *u* et *v* de Σ^* : $h(u.v)=h(u).h(v)$

Remarque :
Tout homomorphisme est parfaitement déterminé par son image sur les lettres de Σ. Il s'étend alors aux mots de Σ^* par :

$$h(w_1w_2\ldots w_n)=h(w_1.w_2.\ \ldots\ .w_n)=h(w_1).h(w_2).\ \ldots\ .h(w_n)$$

III. L'ensemble $\mathbb{N}$

Nous avons déjà effectué des rappels sur les ensembles. Nous nous intéressons maintenant à un ensemble fondamental dans la structure de la théorie de l'informatique : l'ensemble des entiers naturels $\mathbb{N}$.

1) *Axiomatique de Peano*

Axiomes :
L'ensemble $\mathbb{N}$ est défini comme de la manière suivante :

- L'élément appelé zéro et noté 0, est un élément de $\mathbb{N}$.
- Tout élément de $\mathbb{N}$, n a un unique successeur, souvent noté, s(n).
- Aucun élément de $\mathbb{N}$ n'a 0 pour successeur.
- Deux éléments de $\mathbb{N}$ ayant même successeur sont égaux.
- Si un ensemble E contient 0 et contient le successeur de chacun de ses éléments, alors cet ensemble est égal à $\mathbb{N}$.

Définition :
Les éléments de $\mathbb{N}$ sont les **entiers naturels**.

Remarques :

- Le premier axiome permet de poser que l'ensemble des entiers naturels n'est pas vide.
- Le second que le successeur est une fonction.
- Le quatrième que cette fonction est injective.
- Le troisième qu'il possède un premier élément (ces deux axiomes assurent que l'ensemble des entiers naturels est infini).
- Le cinquième est une formulation du principe de récurrence.

2) *Propriétés de $\mathbb{N}$*

L'addition

Propriété :
Soit $n \in \mathbb{N}$.
Il existe une application $m \longmapsto n+m$ de $\mathbb{N}$ dans $\mathbb{N}$ définie par :

- $n+0=n$
- $n+s(p)=s(n+p)$

Démonstration :
Il faut vérifier que l'ensemble A des m pour lesquels l'application est définie est $\mathbb{N}$.
Comme A contient 0 et le successeur de chacun de ses éléments, alors $A=\mathbb{N}$.

Définition :
Cette application définit une **opération** sur $\mathbb{N}$, soit une application de $\mathbb{N}\times\mathbb{N}$ dans $\mathbb{N}$ qui au couple (n, p) associe $n+p$.
Cette opération est appelée **addition** et l'entier $n+p$ est appelé **somme** de n et p.

Le principe de récurrence

Propriété :
Soit $\mathcal{P}(n)$ une propriété de l'entier $n \in \mathbb{N}$.
Si les deux conditions suivantes sont vérifiées :

- $\mathcal{P}(n)$ est vraie (initialisation)
- pour tout $n \in \mathbb{N}$, $\mathcal{P}(n)$ implique $\mathcal{P}(n+1)$ (hérédité)

Alors $\mathcal{P}(n)$ est vraie pour tout $n \in \mathbb{N}$.

Démonstration :
On vérifie que l'ensemble $A=\{n \in \mathbb{N} \mid \mathcal{P}(n) \text{ est vraie}\}$ contient 0 et le successeur de chacun de ses éléments, alors $A = \mathbb{N}$.

La relation d'ordre

Définition :
Soient $(p,q) \in \mathbb{N} \times \mathbb{N}$.
On dit que p est **supérieur** ou égal à p, et on écrit $q \geqslant p$, s'il existe $n \in \mathbb{N}$ tel que $q = n + p$.
On dit que q est **strictement supérieur** à p si on a $q \geqslant p$ et $q \neq p$ et on note $q > p$.

Propriété :
La relation $\geqslant$ est une **relation d'ordre**, c'est-à-dire :

- Elle est **réflexive** : $p \geqslant p$.
- Elle est **antisymétrique** : si $p \geqslant q$ et $q \geqslant p$ alors $p = q$.
- Elle est **transitive** : si $r \geqslant q$ et $q \geqslant p$ alors $r \geqslant p$.

Démonstration :
Cela résulte des propriétés de l'addition.

Propriété :
Toute partie non vide de $\mathbb{N}$ admet un plus petit élément.
On dit que $\mathbb{N}$ est **bien ordonné**.

Démonstration :
On démontre que si A est une partie de $\mathbb{N}$ qui n'a pas de plus petit élément, A est vide.
On montre, par récurrence, la propriété $\mathcal{P}(n) : \{\forall\, i \leqslant n,\ i \notin A\}$.

- $\mathcal{P}(0)$ est vraie (sinon 0 serait le plus petit élément de A).
- On suppose que $\mathcal{P}(n)$ est vraie.
  On sait alors qu'aucun des entiers 0, 1, … , n n'est dans A.
  $n+1$ n'est pas non plus dans A (sinon $n+1$ serait le plus petit élément de A ce qui est contraire à l'hypothèse faite sur A).
  Ainsi $\mathcal{P}(n+1)$ est vraie.
- On conclut donc que A est l'ensemble vide et, par contraposition, que tout ensemble non vide de $\mathbb{N}$ admet un plus petit élément.

3) Dénombrabilité

Un ensemble E est dénombrable s'il est possible de mettre en correspondance $\mathbb{N}$ avec E.

Définition :
Une application $f : A \longmapsto B$ entre deux ensembles est une **bijection** si, pour tout $y \in B$, l'équation $f(x)=y$ possède une et une seule solution.

Définition :
Un ensemble E est **dénombrable** s'il existe une bijection de E sur un sous-ensemble de $\mathbb{N}$.

Exemples :

- Tout ensemble fini E est dénombrable.

E	x_0	x_1			…			x_{n-1}	x_n
$\mathbb{N}$	0	1			…			$n-1$	n

- L'ensemble des entiers naturels pairs est dénombrable.

P	0	2	…		$2(n-1)$	$2n$	…		
$\mathbb{N}$	0	1	…		$n-1$	n	…		

- L'ensemble des entiers relatifs est dénombrable

$\mathbb{Z}$	0	1	-1	2	-2	…	n	$-n$	…
$\mathbb{N}$	0	1	2	3	4	…	$2n-1$	$2n$	…

- $\mathbb{N}\times\mathbb{N}$ est dénombrable

On ordonne les couples selon la somme de leurs éléments, puis selon un ordre lexicographique :

$\mathbb{N}\times\mathbb{N}$	$(0,0)$	$(0,1)$	$(1,0)$	$(0,2)$	$(1,1)$	$(2,0)$	…	$(0,n)$	$(1,n-1)$	…	$(n,0)$	…
$\mathbb{N}$	0	1	2	3	4	5	…	$\frac{n(n+1)}{2}$		…	$\frac{n(n+3)}{2}$	…

On en déduit ensuite que tout ensemble de la forme $\mathbb{N}^k$ ($k \in \mathbb{N}$) est dénombrable, de même que $\mathbb{Q}$.
On remarquera également que tout sous ensemble d'un ensemble dénombrable est dénombrable.

Remarque :
En fait, il suffit d'être judicieux dans le "rangement" des éléments de E pour mettre en évidence la bijection.

IV. Récursivité et induction

L'induction structurelle est une généralisation de la preuve par récurrence.

1) Définitions inductives

Les définitions inductives visent à définir des parties d'une ensemble *E*. Intuitivement, une partie *X* se définit inductivement si on peut la définir avec la donnée explicite de certains éléments de *X* et de moyens de construire de nouveaux éléments de *X* à partir d'éléments de *X*.

Exemple :
Les entiers pairs peuvent se définir par : $P=\{n \in \mathbb{N} | \exists k \in \mathbb{N},\ n=2\times k\ \}$
On peut également définir *P* comme étant le **plus petit ensemble** tel que :

- $0 \in P$
- $n \in P \Rightarrow\ n+2 \in P$

Définition :
Soit *E* une ensemble.
Une **définition inductive** d'une partie *X* de *E* consiste à donner :

- un sous ensemble non vide *B* de *E* (appelé ensemble de base)
- un ensemble de règles *R* (une règle est une fonction de $E^k \rightarrow E$)

Propriété (théorème du point fixe) :
À une définition inductive correspond un plus petit ensemble *X* qui vérifie :

- il contient *B* : $B \subset X$
- il est stable par les règles de *R* (pour chaque règle $r_i \in R$, pour tout $x \in X$ on a $r_i(x) \in X$)

On dit que cet ensemble *X* est alors défini inductivement.

Exemples :

- Soit $\Sigma=\{(,)\}$ l'alphabet constitué de la parenthèse ouvrante et de la parenthèse fermante.
  L'ensemble $D \subset \Sigma^*$ des parenthésages bien formés (appelé langage de Dick) est défini par :
  - $\epsilon \in D$
  - $x \in D \Rightarrow\ (x) \in D \quad ; \quad (x, y) \in D^2 \Rightarrow\ xy \in D$
- Le langage *L* sur l'alphabet $\Sigma=\{a, b\}$ des mots de la forme $a^n b c^n$, $(n \in \mathbb{N})$ se définit par :
  - $b \in L$
  - $w \in L \Rightarrow\ awc \in L$
- $\Sigma_{exp}=\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, –, \times, /, (,)\}$
  - L'ensemble $\mathcal{N}$ des nombres entiers non nuls écrits en base 10 est la partie de Σ_{exp}^* définit par :
    - $a \in \mathcal{N}$ pour chaque $a \in \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
    - $g \in \mathcal{N} \Rightarrow\ ga \in \mathcal{N}$ pour chaque $a \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - L'ensemble *Arith* des expressions arithmétiques est la partie de Σ_{exp}^* définit par :
    - $0 \in Arith$ et $\mathcal{N} \in Arith$
    - $(g, d) \in Arith^2 \Rightarrow\ g+d \in Arith$; $(g, d) \in Arith^2 \Rightarrow\ g \times d \in Arith$;
      $(g, d) \in Arith^2 \Rightarrow\ g / d \in Arith$; $(g, d) \in Arith^2 \Rightarrow\ g-d \in Arith$;
      $g \in Arith \Rightarrow\ (g) \in Arith$

2) Preuves inductives

On est généralement amené à prouver des propriétés sur les éléments d'un ensemble X défini inductivement. Cela est possible en utilisant la preuve par induction.

Propriété :
Soit $X \subset E$ un ensemble défini inductivement à partir d'un ensemble de base B et de règle R.
Soit P un prédicat exprimant une propriété d'un élément $x \in E$ (c'est-à-dire une propriété $P(x)$ qui est soit vraie, soit fausse en un élément $x \in E$.
Si les conditions suivantes sont vérifiées :

- $P(x)$ est vérifiée pour chaque élément $x \in B$.
- P est héréditaire (c'est-à-dire stable par les règles de R)

Alors $P(x)$ est vraie pour chaque élément $x \in X$.

La preuve par induction généralise la preuve par récurrence.

Exemple :
On considère le langage L sur l'alphabet Σ={a, b} des mots de la forme $a^n b c^n$, ($n \in \mathbb{N}$) définit par :

- $b \in L$
- $w \in L \Rightarrow awc \in L$

Pour prouver que tous les mots de L possèdent autant de a que de c, il suffit de constater que :

- c'est vrai pour le mot réduit à la lettre b (qui possède 0 fois la lettre a et la lettre c)
- si cela est vrai pour le mot w alors le mot awb possède aussi le même nombre de fois la lettre a que la lettre c (à savoir une fois de plus que dans w)

3) Fonctions définies inductivement

Il est alors important de définir des fonctions sur un ensemble X défini inductivement et de façon non ambiguë.
Cette notion d'ensemble définie de façon non ambiguë signifie qu'il n'existe qu'une unique façon de construire chaque élément de X.

Exemple :
La définition de *Arith* est ambiguë car 1+2+3 peut s'obtenir de différentes manières : 1+(2+3) ou (1+2)+3.
Pour lever cette ambiguïté, on définit *Arith'* l'ensemble des expressions arithmétiques parenthésées comme la partie de Σ_{exp}^* définie par :

- $0 \in Arith'$ et $\mathcal{N} \in Arith'$
- $(g,d) \in Arith'^2 \Rightarrow (g+d) \in Arith'$; $(g,d) \in Arith'^2 \Rightarrow (g \times d) \in Arith'$;
  $(g,d) \in Arith'^2 \Rightarrow (g / d) \in Arith'$; $(g,d) \in Arith'^2 \Rightarrow (g - d) \in Arith'$;
  $g \in Arith' \Rightarrow (g) \in Arith'$

Propriété :
Soit $X \subset E$ un ensemble défini inductivement, de façon non ambiguë, à partir d'un ensemble de base B et de règle R. Soit Y un ensemble.
Pour qu'une application f de X dans Y soit définie, il suffit de donner

- la valeur de $f(x)$ pour chacun des éléments $x \in B$.
- pour chaque règle, la valeur de $f(x)$ pour $x \in E^k$ en fonction de x_i et $f(x_i)$.

Exemples :

- La fonction factorielle *Fact* de $\mathbb{N}$ dans $\mathbb{N}$ se définit par :
  - $Fact(0)=1$
  - $Fact(n+1)=(n+1)\times Fact(n)$
- La fonction h qui à un mot de $\mathcal{N}$ associe sa valeur en tant que rationnel se définit par :
  - $h(a)=a$ pour chaque $a \in \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - $h(ga)=10\times h(g)+a$ pour chaque $a \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- On définit alors la fonction v donnant la valeur d'une expression arithmétique de *Arith'* (v est une fonction de *Arith'* dans $\mathbb{Q}$) :
  - $v(0)=0$ et $v(x)=h(x)$ pour $x \in \mathcal{N}$
  - $v((g+d))=v(g)+v(d)$; $v((g\times d))=v(g)\times v(d)$
    $v((g/d))=v(g)/v(d)$; $v((g-d))=v(g)-v(d)$
    $v((g))=v(g)$

V. Logique

La logique est un pan fondamental de l'informatique. Nous verrons, en effet, que l'unité d'information étant le bit, l'architecture d'un ordinateur se base sur des portes logiques (l'objectif étant de contrôler le flux des informations).
Mais, avant ces considérations pratiques, la logique est à la source de la théorie de l'informatique. En effet la **logique propositionnelle** (qui constitue la base des systèmes logiques) est à l'origine de la construction des raisonnements mathématiques.
Ce modèle s'avérera toutefois insuffisant et les logiciens, en étudiant la structure des démonstrations et afin d'exprimer, au mieux, les propriétés mathématiques se baseront finalement sur le **calcul des prédicats**. Ce formalisme est très utilisé en informatique pour décrire les objets (et est à l'origine des premiers modèles informatiques).

1) Calcul propositionnel

Soit un ensemble fini ou dénombrable $P = \{ p_0, p_1, ... \}$ de symboles que l'on appelle variables propositionnelles.

Définition :
L'ensemble des formules propositionnelles F sur P est le langage sur l'alphabet :

$$P \cup \{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, (,)\}$$

définit inductivement par :

- il contient P (toute variable propositionnelle est une formule propositionnelle)
- si $f \in F$ alors $\neg f \in F$;
  si $f, g \in F$ alors $(f \wedge g) \in F$, $(f \vee g) \in F$, $(f \Rightarrow g) \in F$, $(f \Leftrightarrow g) \in F$

Remarque :
Soit f une formule propositionnelle.
Alors f est d'une (et exactement d'une) des formes suivantes :

- une variable propositionnelle $p \in P$
- $\neg g$ où g est une formule propositionnelle
- $(g \wedge h)$ où g et h sont des formules propositionnelles
- $(g \vee h)$ où g et h sont des formules propositionnelles
- $(g \Rightarrow h)$ où g et h sont des formules propositionnelles
- $(g \Leftrightarrow h)$ où g et h sont des formules propositionnelles

Remarque :
Lorsque l'on s'intéresse à la sémantique du calcul propositionnel, on interprète les symboles $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$ par la *négation* logique, le *et* logique (conjonction) le *ou* logique (disjonction) l'*implication* et la *double implication* (équivalence).

Définitions :
Une **tautologie** est une formule qui est toujours vraie.
On définit ainsi des **formules équivalentes**.

Exemple :
$(p \vee \neg p)$ est une tautologie.
Les formules (p) et $(\neg \neg p)$ sont équivalentes. On note $(p) \equiv (\neg \neg p)$

Remarques :
On établit alors :

- Les tautologies suivantes :
  Soit f, g et h des formules propositionnelles :
  $(f \Rightarrow f)$; $(f \Rightarrow (g \Rightarrow f))$; $(f \Rightarrow (g \Rightarrow h)) \Leftrightarrow ((f \Rightarrow g) \Rightarrow (f \Rightarrow h))$
- Les équivalences suivantes :
  Soit f, g et h des formules propositionnelles :
  - *Idempotence* :
    $(f \wedge f) \equiv (f)$; $(f \vee f) \equiv (f)$
  - *Associativité* :
    $(f \wedge (g \wedge h)) \equiv ((f \wedge g) \wedge h)$; $(f \vee (g \vee h)) \equiv ((f \vee g) \vee h)$
  - *Commutativité* :
    $(f \wedge g) \equiv (g \wedge f)$; $(f \vee g) \equiv (g \vee f)$
  - *Distributivité* :
    $(f \wedge (g \vee h)) \equiv ((f \wedge g) \vee (f \wedge h))$; $(f \vee (g \wedge h)) \equiv ((f \vee g) \wedge (f \vee h))$
  - *Lois de Morgan* :
    $\neg(f \wedge g) \equiv (\neg f \vee \neg g)$; $\neg(f \vee g) \equiv (\neg f \wedge \neg g)$
  - *Absorption* :
    $(f \wedge (f \vee g)) \equiv (f)$; $(f \vee (f \wedge g)) \equiv (f)$

Propriété :
Toute formule propositionnelle est équivalente à une formule qui est construite uniquement avec les connecteurs ¬ et ∧.

On peut ainsi transformer des formules (on souhaite évidemment obtenir une forme équivalente la plus simple possible).
On peut utiliser la méthode suivante :

- élimination des ⇒ par :
  $(f \Rightarrow g) \equiv (\neg f \vee g)$
- entrée des négations le plus à l'intérieur possible :
  $\neg(f \wedge g) \equiv (\neg f \vee \neg g)$; $\neg(f \vee g) \equiv (\neg f \wedge \neg g)$
- Distributivité :
  $(f \wedge (g \vee h)) \equiv ((f \wedge g) \vee (f \wedge h))$; $(f \vee (g \wedge h)) \equiv ((f \vee g) \wedge (f \vee h))$

2) Démonstration

On peut ainsi, à partir du calcul propositionnel, formaliser la notion de démonstration.
Par exemple, une démonstration (suivant Frege et Hilbert) est un système de déduction qui se base sur un système d'axiomes (qui sont des tautologies) et on utilise une unique règle de déduction : le *modus ponens*.
La règle dit qu'à partir de f et $(f \Rightarrow g)$ on déduit g.

Une preuve par *modus ponens* utilise l'un des axiomes suivants :

- $(f \Rightarrow (g \Rightarrow f))$
- $(f \Rightarrow (g \Rightarrow h)) \Rightarrow ((f \Rightarrow g) \Rightarrow (f \Rightarrow h))$
- $(f \Rightarrow \neg\neg f)$
- $(\neg\neg f \Rightarrow f)$
- $((f \Rightarrow g) \Rightarrow (\neg g \Rightarrow \neg f))$
- $(f \Rightarrow (g \Rightarrow (f \wedge g)))$
- $((f \wedge g) \Rightarrow f)$
- $((f \wedge g) \Rightarrow g)$
- $(f \Rightarrow (f \vee g))$
- $(g \Rightarrow (f \vee g))$
- $((((f \vee g) \wedge (f \Rightarrow h)) \wedge (g \Rightarrow h)) \Rightarrow h)$

Exemple :
Soient f , g et h trois formules propositionnelles.
Voici une preuve de $(f \Rightarrow h)$ à partir de $\{ (f \Rightarrow g) , (g \Rightarrow h) \}$

- F_1 : $(g \Rightarrow h)$ — *hypothèse*
- F_2 : $((g \Rightarrow h) \Rightarrow (f \Rightarrow (g \Rightarrow h)))$ — *axiome 1*
- F_3 : $(f \Rightarrow (g \Rightarrow h))$ — *modus ponens à partir* F_1 *et* F_2
- F_4 : $(f \Rightarrow (g \Rightarrow h)) \Rightarrow ((f \Rightarrow g) \Rightarrow (f \Rightarrow h))$ — *axiome 2*
- F_5 : $((f \Rightarrow g) \Rightarrow (f \Rightarrow h))$ — *modus ponens à partir* F_3 *et* F_4
- F_6 : $(f \Rightarrow g)$ — *hypothèse*
- F_7 : $(f \Rightarrow h)$ — *modus ponens à partir* F_6 *et* F_5

3) Calcul des prédicats

Le calcul propositionnel permet essentiellement d'exprimer des opérations booléennes sur des propositions.
Si l'on veut raisonner sur des assertions mathématiques, nous avons besoin de constructions plus riches.
Pour écrire une formule dans le cadre du calcul des prédicats on utilise :

- les connecteurs : $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$
- les parenthèses (et) et la virgule ,
- les quantificateurs : $\forall$ (universel) et $\exists$ (existentiel)
- un ensemble fini de symboles (constantes, fonctions, relations)

Exemple :
On peut construire l'énoncé suivant :

$$\forall x((Premier(x) \wedge (x > 1+1)) \Rightarrow Impair(x))$$

Cet énoncé ne peut être fait en logique propositionnelle car il utilise des fonctions (dont la valeur de vérité dépend d'une variable x). On utilise également des quantificateurs.

Le calcul des prédicats reste le formalisme le plus courant pour exprimer des propriétés mathématiques. Il est en effet beaucoup plus fin et permet de décrire différents objets et parler de leurs propriétés.
Il permet, en outre, de formaliser les notions de théorie et de modèle.

VI. Le cadre de l'informatique

1) *Modèle informatique*

Définition :
Un **algorithme** est défini comme un ensemble fini et organisé d'**instructions** qui doit répondre à un problème et qui doit vérifier les conditions suivantes :

- L'algorithme doit pouvoir être écrit dans un certain langage.
- La question posée est déterminée par une donnée, appelée **entrée**, pour laquelle l'algorithme sera exécuté.
- L'algorithme est un processus qui s'exécute étape par étape.
- L'action, à chaque étape, est strictement déterminée par l'algorithme, l'entrée et les résultats obtenus dans les étapes précédentes.
- La réponse, appelée **sortie**, est clairement spécifiée.
- Quelle que soit l'entrée, l'exécution se termine en un nombre fini d'étapes.

Intuitivement, un algorithme est une méthode automatique (que l'on souhaite implémenter dans un programme informatique) pour résoudre un problème.
Les techniques utilisées pour réaliser une addition, une multiplication ou une division sur des nombres correspondent à des algorithmes. Par ailleurs, les méthodes de démonstrations elle-mêmes peuvent être considérées comme des algorithmes.

La formalisation de ce qui est calculable par un dispositif informatique digital a donné naissance à de nombreux modèles (machines de Turing, systèmes de Post, λ-calcul de Church) qui se sont avérés, dans une certaine mesure, équivalents et qui permettent de fixer un cadre (et donc des limites) à l'informatique.

Définition :
L'**informatique théorique** a pour objet d'étude la notion d'**information** et des **procédés de traitement** automatique de celle-ci, l'**algorithmique**.

La formalisation et les modèles actuellement choisis pour l'informatique correspondent à des machines ne pouvant traiter (théoriquement) qu'un ensemble dénombrable d'informations (dont la nature même doit appartenir à un ensemble dénombrable) et selon un ensemble dénombrable de procédés.

Remarque :
En pratique, nous serons, bien entendu, limités à des ensembles finis : les contraintes spatiales de stockage d'information et évidemment temporelles devront être prises en compte.

2) L'information

Un programme informatique ne permet pas de traiter des informations représentant des nombres réels (ou tout type de données de nature non dénombrable).

Propriété :
$\mathbb{R}$ n'est **pas dénombrable**.

Démonstration :
La démonstration utilise la méthode de diagonalisation.
On peut s'appuyer sur le développement décimal des réels (un développement décimal d'un réel est une suite de chiffres).
Il suffit de montrer que l'intervalle [0 ;1] n'est pas dénombrable (tout sous-ensemble d'un ensemble dénombrable l'est aussi). On représentera un nombre en notation décimale en évitant la représentation périodique ... $\overline{9}$ afin d'éliminer les ambiguïtés. Par exemple, 0,1324 $\overline{9}$ est représenté par 0,1325. De cette façon, tout nombre admet une représentation décimale unique. Les nombres dans l'intervalle [0 ;1] sont précisément ceux qui ont une représentation de la forme $0, x_1 x_2 x_3 \ldots$.
Supposons que l'intervalle [0 ;1] est dénombrable.
Il existe donc une suite $r=(r_1, r_2, r_3, ...)$ dans laquelle chaque nombre entre 0 et 1 se retrouve une fois.
Par exemple :

r_1 = 0, **0** 1 0 5 1 1 0 …
r_2 = 0, 4 **1** 3 2 0 4 3 …
r_3 = 0, 8 2 **4** 5 0 2 6 …
r_4 = 0, 2 3 3 **0** 1 2 6 …
r_5 = 0, 4 1 0 7 **2** 4 6 …
r_6 = 0, 9 9 3 7 8 **1** 8 …
r_7 = 0, 0 1 0 5 1 3 **0** …

On va maintenant construire (par diagonalisation) un nombre *x* dans l'intervalle qui ne peut être dans cette suite, ce qui, par contradiction, montrera que [0 ;1] n'est pas dénombrable.
Le nombre réel *x* est construit par la donnée de ses décimales suivant la règle : si la $n^{\text{ième}}$ décimale de r_n est différente de 1, alors la $n^{\text{ième}}$ décimale de *x* est 1, sinon la $n^{\text{ième}}$ est 2.
Par exemple avec la suite ci-dessus, la règle donne *x* = 0, 1 2 1 1 1 2 1 …
Le nombre *x* est clairement dans l'intervalle [0 ;1] mais ne peut pas être dans la suite $r=(r_1, r_2, r_3, ...)$, car il n'est égal à aucun des nombres de la suite : il ne peut pas être égal à r_1 car la première décimale de *x* est différente de celle de r_1, de même pour r_2 en considérant la deuxième décimale, etc.
Il diffère donc de chaque terme de la suite par au moins une décimale

Remarques :

- Ceci doit être pris en compte : la machine n'est pas capable (même en théorie) d'effectuer des calculs sur des données qui sont des nombres réels.
  Il est exact qu'en informatique tout est nombre. La réciproque est plus délicate !
- Il appartient alors, aux concepteurs de programmes, selon le but recherché, de choisir une représentation de l'information la plus adaptée possible en essayant d'être capable de quantifier les erreurs.
- Les calculs effectués par les ordinateurs se basent donc essentiellement sur l'arithmétique des entiers et non sur les notions de calculs des réels.

3) *Le traitement des informations*

Un programme informatique ne permet pas d'effectuer un nombre non dénombrable de traitements ou encore de traiter un nombre non dénombrable d'informations.

Propriété :
L'ensemble des parties de $\mathbb{N}$ n'est **pas dénombrable**.

Démonstration :
Supposons qu'il existe une telle bijection $f : \mathbb{N} \longmapsto \mathcal{P}(\mathbb{N})$.
Posons :

$$F=\{k \in \mathbb{N} | k \notin f(k)\}$$

Puisque $F \subset \mathbb{N}$, on peut trouver k_0 tel que $f(k_0)=F$.
On a donc soit $k_0 \in F$, soit $k_0 \notin F$.

- Si $k_0 \in F$, par définition, $k_0 \notin F$ (contradiction)
- Si $k_0 \notin F$, par définition, $k_0 \in F$ (contradiction)

L'hypothèse de départ était donc absurde : il n'existe pas de bijection entre $\mathbb{N}$ et $\mathcal{P}(\mathbb{N})$.

Remarques :

- L'idée générale étant que l'on peut tout coder par un mot (fini sur l'alphabet $\Sigma_{bin}=\{0, 1\}$), il y a un nombre dénombrable de modèles informatiques (qui permettent donc de résoudre de nombreux problèmes).
  Par contre, il y a un nombre non dénombrable de langages sur l'alphabet $\Sigma_{bin}=\{0, 1\}$. Il y a donc des problèmes qui ne correspondent à aucun modèle informatique.
  Il faudra donc être capable de rendre le problème compréhensible par l'ordinateur (ce qui est loin d'être évident) et se pose ensuite la question de savoir si ce problème est résoluble par ce dernier.
- En fait, il s'avère, non seulement, que certains problèmes ne peuvent pas se résoudre informatiquement mais, surtout, qu'il existe des problèmes pour lesquels on sait qu'il est impossible de produire une solution algorithmique : on parle de **problèmes indécidables.**

Exemple :
Le dixième problème de Hilbert demande de trouver une méthode algorithmique générale pour décider s'il existe des solutions entières des équations diophantiennes à plusieurs inconnues (c'est-à-dire des équations polynomiales à coefficients entiers).
Ce problème est indécidable.

Définition :
Un **problème algorithmique** est un problème posé de façon mathématique, c'est-à-dire qu'il est énoncé rigoureusement dans le langage des mathématiques (en utilisant généralement le calcul des prédicats). Il comprend des hypothèses, des données et une question.
On distingue deux types de problèmes :

- les **problèmes de décision** : ils posent une question dont la réponse est oui ou non.
- les **problèmes d'existence** ou de **recherche d'une solution**.

La théorie de la complexité étudie les problèmes dont on sait qu'ils ont une solution.

4) La complexité

Dans chaque catégorie de problèmes ci-dessus, on dit qu'un problème a une réponse algorithmique si sa réponse peut être fournie par un algorithme.

- Un problème est **décidable** s'il s'agit d'un problème de décision (donc d'un problème dont la réponse est soit oui soit non) et si sa réponse peut être fournie par un algorithme.
- Symétriquement, un problème est **calculable** s'il s'agit d'un problème d'existence et si l'élément calculé peut-être fourni par un algorithme.

La théorie de la complexité ne couvre que les problèmes décidables ou calculables et cherche à évaluer les ressources (temps et espace mémoire) mobilisées pour obtenir algorithmiquement la réponse.

Complexité en temps

On va donc s'intéresser au problème de la **complexité** d'un algorithme (à savoir : est-il efficace ?).
Pour la complexité en temps (qui est plus simple, car ne dépendant pas autant du modèle de calcul, à étudier que la complexité en mémoire), on quantifie l'efficacité en comparant les temps d'exécution (pour une même puissance de calculs).

Définition :
Un algorithme est **efficace** si sa complexité en temps est **polynomiale**.

Sans rentrer dans les détails, un algorithme est considéré, comme non efficace si sa complexité est **exponentielle**.

Nous avons donc affaire à des algorithmes permettant de donner des solutions (par un traitement exhaustif, par exemple) à un problème posé mais nous souhaitons savoir s'il est possible d'obtenir ces solutions dans un temps raisonnable.

Ces problèmes difficiles à traiter sont des **problèmes NP-complets** (un problème est dans NP s'il est décidable par un modèle non déterministe en temps polynomial). Une question ouverte est de découvrir si ces problèmes admettent un algorithme efficace (la classe P est la classe des problèmes qui se résolvent avec un algorithme polynomial).

Exemple :
Le problème du voyageur de commerce est un problème mathématique qui consiste, étant donné un ensemble de villes séparées par des distances données, à trouver le plus court chemin qui relie toutes les villes.
Il s'agit d'un problème d'optimisation pour lequel on ne connaît pas d'algorithme permettant de trouver une solution exacte en un temps polynomial.
De plus, la version décisionnelle de l'énoncé (pour une distance D, existe-t-il un chemin plus court que D passant par toutes les villes ?) est connue comme étant un problème NP-complet.

P=NP ?

Tous les algorithmes connus pour résoudre des problèmes NP-complets ont un temps d'exécution exponentiel en la taille des données d'entrée (dans le pire cas), et sont donc inexploitables en pratique même pour des instances de taille modérée.

Nous savons, par contre, que s'il existe un algorithme polynomial pour résoudre un quelconque des problèmes NP-complets, alors tous les problèmes de la classe NP peuvent être résolus en temps polynomial.

C'est un problème avec de grands enjeux :

- En effet, si P=NP alors une grande classe de problème pourra enfin être résolu (dans un temps raisonnable).
- Toutefois, les techniques de cryptographies les plus abouties et utilisées quotidiennement s'avèrent efficaces car elles s'appuient sur le postulat que $P \neq NP$.
  En effet, les algorithmes permettant de décoder les informations existent mais leur complexité est exponentielle (par exemple, la factorisation d'un grand nombre en nombres premiers est possible en théorie mais s'avère pratiquement irréalisable).

Remarque :

Le théorème des quatre couleurs indique qu'il est possible, en n'utilisant que quatre couleurs différentes, de colorer n'importe quelle carte découpée en régions connexes.
Déterminer si un graphe peut être ou non coloré en k couleurs pour $k>2$ est un problème NP-complet.
En 1976, deux Américains, Kenneth Appel et Wolfgang Haken, affirment avoir démontré le théorème des quatre couleurs. Leur démonstration partage la communauté scientifique : pour la première fois, en effet, la démonstration exige l'usage de l'ordinateur pour étudier les 1478 cas critiques (plus de 1200 heures de calcul).
Le problème de la démonstration du théorème se trouve alors déplacé vers le problème de la validation.