Rappels mathématiques

François Lemieux

Département d'informatique et de mathématique
Université du Québec à Chicoutimi

1 Les ensembles

Un ensemble est un groupe d'objets appelés *éléments*.

Les éléments d'un ensemble peuvent être de n'importe quel type: des nombres entiers ou réels, des symboles, et même d'autres ensembles.

Voici quelques exemples d'ensembles:

Exemple 1.1

- $\mathbb{N} = \{0, 1, 2, 3, 4, 5, \ldots\}$
- $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$
- $\mathbb{R} =$ *ensemble des nombres réels*

Un ensemble peut être fini ou infini. Par exemples $\mathbb{N}$,$\mathbb{Z}$, et $\mathbb{R}$ sont des ensembles infinis. Voici quelques exemples d'ensembles finis:

Exemple 1.2

- *L'ensemble vide est dénoté* $\emptyset$
- $\{0, 1\}$
- $\{a, b, c, \ldots, z\}$
- $\{\alpha, \beta, \gamma, \delta, \epsilon, \phi, \psi, \pi\}$
- *L'ensemble des symboles ASCII*

La *cardinalité* d'un ensemble est le nombre d'éléments qu'il contient. La cardinalité d'un ensemble S est dénoté $|S|$.

Comme il a été mentionné, les éléments d'un ensemble peuvent eux-mêmes être des ensembles. C'est le cas de l'ensemble de tous les sous-ensembles d'un ensemble initial.

Exemple 1.3 *L'ensemble de tous les sous-ensembles de* $\{1, 2, 3\}$ *est*

$$\{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

1.1 Description d'un ensemble

Il y a trois principales façons de décrire un ensemble:

1. **Description en français:** On décrit simplement l'ensemble en utilisant un français aussi précis et complet que possible.

   Exemple 1.4

   - *L'ensemble des nombres pairs*
   - *L'ensemble des nombres premiers*
   - *L'ensemble de toutes les séquences de bits commençant par 1*
   - *Les cinq premières lettres minuscules de l'alphabet latin*

2. **Énumération:** On énumère les éléments de l'ensemble. Lorsque l'ensemble est infini, on énumère les premiers éléments de sorte que les autres puissent être déduits.

   Exemple 1.5

   - $\{0, 2, 4, 6, 8, \ldots\}$
   - $\{2, 3, 5, 7, 11, 13, 17, \ldots\}$
   - $\{10, 11, 100, 101, 110, 111, 1000, \ldots\}$
   - $\{a, b, c, d, e\}$

3. **Description formelle:** On utilise une notation mathématique formelle et sans ambiguïté.

Exemple 1.6

- $\{n \in \mathbb{N} \mid (\exists b \in \mathbb{N})\ [n = 2b]\ \}$
- $\{n \in \mathbb{N} \mid (\forall a, b \in \mathbb{N})\ [n = ab\ \Rightarrow\ (a = 1) \vee (b = 1)]\ \}$

Note: Le symbole $\exists$ signifie "il existe" et le symbole $\forall$ signifie "pour tout".

1.2 Opérations sur les ensembles

Si A et S sont deux ensembles, on dit que A est un *sous-ensemble* de S (dénoté $A \subseteq S$) si tous les éléments de A sont aussi contenus dans S. On dit que A est un sous-ensemble *non trivial* de S si $A \neq \emptyset$ et on dit que A est un sous-ensemble strict de S (dénoté $A \subset S$) si $A \neq S$. Pour tout ensemble non vide S, le nombre de sous-ensembles de S est $2^{|S|}$ où $|S|$ est la cardinalité de S.

Dans la plupart des applications, il existe un ensemble U correspondant à l'univers de référence, c'est-à-dire l'ensemble de tous les éléments avec lesquels on travaille. On est alors intéressé par certains sous-ensembles de cet univers.

Exemple 1.7 *Définissons U comme l'ensemble de toutes les séquences de bits. Un programme en C peut-être vu comme une séquence de bits et l'ensemble de tous les programmes syntactiquement corrects forme un sous-ensemble de U.*

Les opérations de base sur les ensembles sont l'*union*, l'*intersection*, la *complémentation* et la *différence*. Pour définir ces opérations, considérons un univers de référence U ainsi que deux sous-ensembles $A \subseteq U$ et $B \subseteq U$.

Union: $A \cup B = \{x \in U \mid (x \in A) \vee (x \in B)\}$

Intersection: $A \cap B = \{x \in U \mid (x \in A) \wedge (x \in B)\}$

Différence: $A - B = \{x \in U \mid (x \in A) \wedge (x \notin B)\}$

Complémentation: $\overline{A} = \{x \in U \mid x \notin A\} = U - A$

Exemple 1.8 *Considérons les deux ensembles suivants:*

Début_t = ensemble des mots du dictionnaire commençant par t.

Fin_s = ensemble des mots du dictionnaire se terminant par s.

On a donc

$tabac \in Début\_t$

$trois \in Début\_t$

$trois \in Fin\_s$

$amas \in Fin\_s$

$trois \in Début\_t \cap Fin\_s$

$tabac \in Début\_t \cup Fin\_s$

$\{tabac,\ amas\} \subseteq Début\_t \cup Fin\_s$

$\{tabac,\ amas\} \subseteq \overline{Début\_t \cap Fin\_s}$

2 Les suites

Une suite est une succession d'éléments disposés dans un ordre donné. Contrairement à un ensemble, un élément peut apparaître plusieurs fois dans une suite.

Exemple 2.1

- *La suite contenant aucun élément: ()*
- *Une suite de cinq entiers: (3,1,4,1,5)*
- *Une suite de deux suites ((1,2)), (3,4,5)))*
- *Une suite de deux ensembles ({1,2},{3,4,5})*

Remarque: La suite $(\{1,2\},\{3,4,5\})$ est identique à $(\{2,1\},\{5,4,3\})$ puisque les ensembles ne sont pas ordonnés. Cependant $(\{1,2\},\{3,4,5\})$ et $(\{3,4,5\},\{1,2\})$ sont deux suites distinctes.

Une suite de deux éléments est appelée *paire*, une suite de trois éléments est appelée *triplet* et une suite de k éléments est appelé **k-uplet**.

Exemple 2.2 *Un octet est une suite de 8 bits (un 8-uplet). Une suite de bits telle que* $(0,1,1,0,0,1,0,0)$ *est habituellement dénotée* 01100100 *afin d'abréger la notation.*

Exemple 2.3 *Dans le langage C, une chaîne de caractères est une suite d'octets se terminant avec le caractère NUL (l'octet* 00000000*).*

Exemple 2.4 *Dans le langage C, le type d'une variable indique (entre autres) l'ensemble des valeurs possibles qu'on peut lui assigner. Par exemple, sur certains ordinateurs on a:*

- *Un* char *appartient à l'ensemble des suites de 8 bits.*
- *Un* short int *appartient à l'ensemble des suites de 16 bits.*
- *Un* int *appartient à l'ensemble des suites de 32 bits.*
- *Un* double *appartient à l'ensemble des suites de 64 bits.*

Exemple 2.5 *En informatique, un tableau est une suite d'éléments appartenant à un même type. Un tableau à deux dimensions est donc une suite de suites d'éléments appartenant à un même type.*

3 Les relations

Soit A et B, deux ensembles. Le *produit direct* $A \times B$ est l'ensemble de toutes les paires dont le premier élément appartient à A et le second appartient à B. Plus formellement on a:

$$A \times B = \{(a,b) \mid a \in A \text{ et } b \in B\}$$

Relation binaire sur A et B: sous ensemble $R \subseteq A \times B$

Relation binaire sur A: sous-ensemble $R \subseteq A \times A$

Exemple 3.1 $R = \{(x,y) \in \mathbb{R} \times \mathbb{R} \mid y = x^2\}$

Exemple 3.2 *Un graphe (dirigé)* $G = (N,E)$ *est une relation* $E \subseteq N \times N$.

Une relation binaire $R \subseteq A \times A$ est appelée *relation d'équivalence* si elle satisfait les trois propriétés suivantes:

Réflexivité: $(\forall a \in A)[(a, a) \in R]$

Symétrie: $(\forall a, b \in A)[(a, b) \in R \Leftrightarrow (b, a) \in R]$

Transitivité: $(\forall a, b, c \in A)[(a, b) \in R \text{ et } (b, c) \in R \Rightarrow (a, c) \in R]$

Si $R \subseteq A \times A$ est une relation d'équivalence et si a et b sont deux éléments de A alors on dit que a *est équivalent* à b si $(a, b) \in R$. Il est facile de voir que R partitionne l'ensemble A en sous-ensembles disjoints (appelés *partitions*): deux éléments $a, b \in A$ sont dans la même partition si et seulement s'ils sont équivalents. À l'inverse, toute partition de A correspond à une relation d'équivalence R telle que deux éléments sont équivalents si et seulement s'ils sont dans la même partition. Les notions de partitions et de relations d'équivalence sont donc essentiellement identiques.

Exemple 3.3 *Considérons un graphe dirigé $G = (N, E)$ où N est l'ensemble des noeuds et E est l'ensemble des flèches, c'est-à-dire que $E \subseteq N \times N$ est une relation sur N. En général E n'est pas une relation d'équivalence puisqu'il peut exister une flèche de $a \in N$ à $b \in N$ (autrement dit $(a, b) \in E$) sans qu'il y ait une flèche de b à a (autrement dit $(b, a) \notin E$). Nous allons maintenant définir une relation $R \subseteq N \times N$ différente de E. Plus précisément, définissons R comme l'ensemble des pairs $(a, b) \in N \times N$ telles qu'il existe un chemin de a à b et il existe aussi un chemin de b à a. Il est facile de vérifier que R est une relation d'équivalence qui partitionne les noeuds du graphe en sous-ensembles appelés* composantes fortement connexes.

On peut généraliser la notion de relation binaire en relation k-aire. Soit $k \geq 2$ et considérons k ensembles $A_1, A_2, \ldots, A_k$. On défini le produit direct de ces k ensembles de la façon suivante:

$$A_1 \times A_2 \times \cdots \times A_k = \{(a_1, a_2, \ldots, a_k) \mid a_1 \in A_1, a_2 \in A_2, \ldots, a_k \in A_k\}$$

Relation k-aire: sous ensemble $R \subseteq A_1 \times A_2 \times \cdots \times A_k$.

Exemple 3.4 *Lorsqu'un programmeur défini un nouveau type, il défini un ensemble de valeurs possibles. Considérez, par exemple, la déclaration suivante:*

```
struct exemple {
    char a,b;
    int n;
    };
typedef struct exemple Exemple;
```

Les valeurs appartenant au nouveau type Exemple *forment un ensemble pouvant être vu comme le produit direct* $char \times char \times int$.

Exemple 3.5 *Considérez la fonction C suivante:*

```
int f(int x, int y, int z){
    return (x*y == z);
}
```

Soit R l'ensemble des triplets (a, b, c) d'entiers tels que f(a,b,c) retourne la valeur 1. Alors R est une relation $R \subseteq int \times int \times int$.

4 Les fonctions

Soit X et Y, deux ensembles. Une relation $f \subseteq X \times Y$ est appelée *fonction* si pour chaque $x \in X$ il y a au plus un $y \in Y$ tel que $(x, y) \in R$. Le *domaine* de f est l'ensemble suivant:

$$\{x \in X \mid (\exists y \in Y)[(x, y) \in f]\}$$

L'image de f est l'ensemble suivant:

$$\{y \in Y \mid (\exists x \in X)[(x, y) \in f]\}$$

On dit que la fonction f est *totale* si son domaine est X.

Exemple 4.1 *Considérons un ensemble N et une relation $R \subseteq N \times N$ définissant un graphe dirigé. Supposons que ce graphe est un arbre de sorte qu'il y a une flèche du noeud n au noeud m si et seulement si n est un enfant de m. Cette relation est une fonction telle que $(n, m) \in f$ si et seulement si m est le parent de n. On remarque que la racine $r \in N$ ne possède pas de parent. Cela implique qu'il n'y aucun élément s tel que $(s, r) \in R$. La fonction f n'est donc pas totale.*

On voit souvent une fonction $f \subseteq X \times Y$ comme un object mathématique qui associe à une valeur d'entrée une valeur de sortie. On dénote alors cette fonction sous la forme $f : X \to Y$. Par exemple, l'addition, la soustraction et la multiplication sont des fonctions totales dont le domaine est $\mathbb{R} \times \mathbb{R}$ et dont l'image est $\mathbb{R}$.

La division est aussi une fonction $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$ mais celle-ci n'est pas totale puisqu'on ne peut pas diviser par 0. Le domaine de cette fonction est donc $\mathbb{R} \times (\mathbb{R} - \{0\})$.

Exemple 4.2 *La fonction* plancher *est dénoté* $\lfloor \cdot \rfloor : \mathbb{R} \to \mathbb{N}$ *et donne le plus grand entier plus petit ou égal à la valeur d'entrée. Par exemple* $\lfloor 3.1416 \rfloor = 3$.

Exemple 4.3 *La fonction* plafond *est dénoté* $\lceil \cdot \rceil : \mathbb{R} \to \mathbb{N}$ *et donne le plus petit entier plus grand ou égal à la valeur d'entrée. Par exemple* $\lceil 3.1416 \rceil = 4$.

Pour tout nombre entier n, on a l'identitée suivante:

$$n = \left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil$$

Exemple 4.4 *Si S est un ensemble alors on appelle* fonction caractéristique de S *la fonction* $\chi_S : S \to \{0, 1\}$ *telle que*

$$\chi_S(x) = \begin{cases} 1 & \textit{si } w \in S \\ 0 & \textit{sinon} \end{cases}$$

Exemple 4.5 *Nous avons vu que les types de données d'un langage de programmation sont des ensembles de suites de bits. En fait, un type est beaucoup plus que cela. En effet, considérez les types* int *et* float *du langage C. Sur certaines machines un float et un int utilisent 32 bits. Ainsi, ces deux types correspondent au même ensemble de suites de 32 bits. Mais alors qu'est ce qui fait la différence entre les types int et float? La réponse est simple, ce sont les opérations définies sur ces ensembles. Les opérations sont des fonctions définies à l'intérieur du langage.*

Par exemple, l'opérateur % prend deux paramètres de type int en entrée et donne le reste de la division du premier par le second. Ainsi, $9\%3$ *vaut 0 alors que* $10\%3$ *vaut 1. Si vous tentez d'utiliser des valeurs de type float comme paramètres vous aurez une erreur de compilation.*

Considérons un second exemple. En C, la valeur maximale d'un int est INT_MAX qui, sur une machine à 32 bits, vaut $2^{31} - 1$. *La valeur minimale d'un int est INT_MIN qui, sur une machine à 32 bits, vaut* -2^{31}. *La*

définition de ces constantes se trouve dans le fichier limits.h de la bibliothèque standard. Analysons le code suivant:

```
int i=INT_MAX;
float f=i;
i=i+1;
f=f+1;
```

En C, incrémenter une variable de type int qui contient déjà la valeur maximale ne provoque pas d'erreur à l'exécution. Après l'incrémentation, la variable i contiendra la valeur INT_MIN. La situation est tout à fait différente pour les variables de types float. Après l'incrémentation la variable f contiendra la valeur $INT\_MAX + 1$ (sur certains systèmes cela peut être INT_MAX ou INT_MAX+2 à cause des erreurs d'arrondi). Ainsi, l'opérateur d'addition + se comporte différemment selon que ses paramètres sont des variables de type int ou float.

Nous voyons donc qu'un type n'est pas seulement défini par un ensemble de valeurs (suite de bits) mais aussi par un groupe d'opérateurs et leur comportement.

Il est important de bien distinguer les fonctions mathématiques, discutées dans cette section, des fonctions d'un langage de programmation tel que le C. L'exemple suivant illustre qu'il s'agit de deux concepts similaires mais pas identiques.

Exemple 4.6 *Considérez la fonction C suivante:*

```
int g(int n){
    static int x=0;
    x++;
    return n+x;
}
```

Une variable statique est une variable locale qui est définie et initialisée au tout début de l'exécution du programme. Elle conserve toujours la valeur qui lui a été assignée lors du dernier appel de la fonction où elle est déclarée. Ainsi, si on appelle g(1) une première fois, la fonction retournera la valeur 2. Au retour de la fonction, la variable statique x existera toujours et sa valeur sera 1 puisqu'elle est incrémentée à l'intérieur de la fonction. Si on appelle g(1) une seconde fois, cette fois la fonction retournera 3. Si on appelle g(1) une troisième fois elle retournera 4, et ainsi de suite.

4.1 Les polynômes

Un polynôme (à une variable) $p : \mathbb{R} \to \mathbb{R}$ est une fonction de la forme :

$$p(x) = \sum_{i=0}^{k} a_i x^i = a_k x^k + a_{k-1} x^{k-1} + \cdots + a_1 x + a_0$$

La valeur k est appelée le *degré* du polynôme.

Les **fonctions linéaires** sont des polynômes de degré 1.

Exemple 4.7 *Considérez la boucle suivante:*

```
p=b;
for (int i=0;i<x;i++)
    p=p+a;
```

La valeur de x au sortir de la boucle sera $ax + b$.

Les **fonctions quadratiques** sont des polynômes de degré 2.

Exemple 4.8 *Considérez la boucle suivante:*

```
p=c;
for (int i=0;i<x;i++)
    p=p+b;
for (int i=0;i<x;i++)
    for (int j=0;j<x;j++)
        p=p+a;
```

La valeur de x au sortir de la seconde boucle sera $ax^2 + bx + c$.

Les **fonctions cubiques** sont des polynômes de degré 3.

Exemple 4.9 *Considérez la boucle suivante:*

```
p=d;
for (int i=0;i<x;i++)
    p=p+c;
for (int i=0;i<x;i++)
    for (int j=0;j<x;j++)
        p=p+b;
for (int i=0;i<x;i++)
    for (int j=0;j<x;j++)
        for (int k=0;k<x;k++)
            p=p+a;
```

La valeur de x au sortir de la troisième boucle sera $ax^3 + bx^2 + cx + d$.

4.2 La fonction exponentielle

La fonction exponentielle à la base b est dénotée b^n et correspond à la valeur obtenue lorsque l'on multiplie n fois la valeur b par elle même. Plus formellement on a

$$b^n = \begin{cases} 1 & \text{si } n = 0 \\ b * b^{n-1} & \text{sinon} \end{cases}$$

Exemple 4.10 *Considérez le code suivant:*

```
k=1;
for (int i=0;i<x;i++)
    k=k*b;
```

Au sortir de la boucle, on aura $k = b^x$.

Exemple 4.11 *Considérez la fonction C suivante:*

```
int exp(int b, int n){
    if (n==0) return 1;
    return b*exp(b,n-1);
}
```

Lorsque $n \geq 0$ *alors exp(b,n) retourne la valeur* b^n.

Les identités suivantes sont utiles.

1. $(b^n)^m = b^{nm}$
2. $b^n b^m = b^{n+m}$
3. $b^n * c^n = (bc)^n$

Remarque 1: Quel est la valeur de b^0? La réponse est simple puisque selon la seconde identité on a $b^n = b^{n+0} = b^n b^0$. On doit donc avoir $b^0 = 1$.

Remarque 2: Quel est la valeur de b^{-n}? Selon la seconde identité, on doit avoir $b^{-n}b^n = b^{-n+n} = b^0 = 1$. On doit donc avoir $b^{-n} = 1/b^n$.

Exemple 4.12 *Combien de suites distinctes de n bits y a-t-il? Il y a 2 sequences de 1 bit, 4 de 2 bits et 8 de 3 bits. De façon générale, il y a* 2^n *suites dictinctes de n bits. Par exemple, dans le langage C, le nombre de bits dans un caractère est 8. On peut donc représenter au plus* $2^8 = 256$ *caractères différents.*

Exemple 4.13 *Si S est un ensemble fini et* $\mathcal{P}(S)$ *est l'ensemble de tous les sous-ensembles de S alors on a* $|\mathcal{P}(S)| = 2^{|S|}$

Exemple 4.14 *Quel est le nombre maximal de noeuds que peut posséder un arbre binaire de profondeur k? La réponse est 1 si* $k = 0$, *3 si* $k = 1$, *et 7 si* $k = 2$. *En général, la réponse est* $2^{k+1} - 1$. *Le nombre maximal de feuilles que peut posséder un arbre binaire de profondeur k est donc* 2^k.

4.3 La fonction logarithmique

Le logarithme de x à la base b est dénoté $\log_b x$ et correspond à la valeur de l'exposant que l'on doit donner à b pour obtenir x. La fonction logarithmique est à la fonction exponentielle ce que la division est à la multiplication. On a donc

$$\log_b x = n \Leftrightarrow b^n = x$$

La fonction logarithmique est donc l'inverse de la fonction exponentielle et les identités suivantes découle directement de la définition.

- $x = b^{\log_b x}$
- $x = \log_b b^x$

Les quelques identités qui suivent sont d'une grande utilitées:

- $\log_b x = \frac{\log_a x}{\log_a b}$
- $\log_b xy = \log_b x + \log_b y$
- $\log_b x/y = \log_b x - \log_b y$
- $\log_b x^r = r \log_b x$

Exemple 4.15 *Combien de bits sont-ils nécessaires pour représenter un entier $x \geq 1$ en binaire? Considérons d'abord le cas ou x est une puissance de 2. Lorsque $x = 1$ on a besoin de 1 bit, lorsque $x = 2$ on a besoin de 2 bits, lorsque $x = 4$ on a besoin de 3 bits. De façon générale, lorsque $x = 2^n$, on a besoin de $n + 1$ bits. Donc, lorsque x est une puissance de 2, on a besoin de $log_2 x + 1$ bits.*

Supposons maintenant que x n'est pas une puissance de 2. Cela veut dire qu'il existe un entier n tel que $2^n < x < 2^{n+1}$. Cela veut aussi dire que le nombre de bits nécessaires pour exprimer x est le même que pour 2^n, c'est-à-dire $n + 1$.

Donc, le nombre de bits nécessaires pour exprimer $x \geq 1$ en binaire est $\lfloor \log_2 x \rfloor + 1$.

Exemple 4.16 *Considérez le code suivant:*

```
l=0;
for (int i=x;i>0;i=i/2)
    l=l+1;
```

Lorsque l'on utilise la division entière $i/2$, le résultat est identique à celui que l'on obtiendrait si l'on faisait un décalage vers la droite des bits de i. Par exemple, avec la division entière on a $5/2 = 4/2 = 2$ alors que la représentation binaire de 4 est 100, celle de 5 est 101 et celle de 2 est 10. Ainsi, au sortir de la boucle, la variable l contiendra le nombre de bits nécessaires pour exprimer x en binaire, c'est-à-dire $l = \lfloor \log_2 x \rfloor + 1$.

Exemple 4.17 *La profondeur d'un arbre binaire est la longueur du plus long chemin entre la racine et une feuille. Quelle est la profondeur minimale que peut avoir un arbre binaire de n noeuds?*

Considérons d'abord le cas où n est de la forme $n = 2^k - 1$. On réalise alors qu'un arbre binaire a une profondeur minimale s'il est complet, c'est-à-dire si chaque noeud est soit une feuille ou possède exactement 2 enfants. Lorsque $n = 1$, la profondeur minimale est 0, lorsque $n = 3$, la profondeur minimale est 2, lorsque $n = 7$, la profondeur minimale est 3. De façon générale, lorsque $n = 2^k - 1$, la profondeur minimale est $\lfloor \log_2 n \rfloor = k - 1$.

Il est facile de se convaincre que, quel que soit $n \geq 1$, la profondeur minimale d'un arbre binaire de n noeuds est $\lfloor \log_2 n \rfloor$.

Exemple 4.18 *Une propriété importante de la fonction logarithmique est qu'elle est* monotone*, c'est-à-dire que si $x < y$ alors $\log_b x < \log_b y$. Cette observation est utile lorsque l'on désire comparer deux fonctions complexes. Par exemple, si vous aviez le choix entre deux algorithmes, le premier prenant un temps $f(n) = (n/2)^n$ et le second prenant un temps $g(n) = n^{n/2}$. Lequel choisisseriez-vous? On observe que $\log(f(n)) = n \log n - n$ alors que $log(g(n)) = n \log n/2 = n \log n - n \log n/2$. On en conclu que $\log(f(n))$ augmente plus rapidement que $\log(g(n))$ ce qui implique que $f(n)$ augmente plus rapidement que $g(n)$. Le second algorithme est donc préférable.*

4.4 La fonction factorielle

La fonction factorielle est dénoté $n!$ et est définie de la façon suivante:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n * (n-1)! & \text{sinon} \end{cases}$$

On peut aussi définir la fonction factorielle de la façon suivante:

$$n! = \prod_{i=1}^{n} i = 1 * 2 * 3 * \cdots * n$$

Exemple 4.19 *Considérez le code de la fonction suivante:*

```
f=1;
for (int i=x;i>0;i--)
    f=f*i;
```

Au sortir de la boucle, on aura $f = x!$.

Exemple 4.20 *Considérez la fonction C suivante:*

```
int fact(int n){
    if (n==0) return 1;
    return n*exp(n-1);
}
```

Lorsque $n \geq 0$ *alors fact(n) retourne la valeur* $n!$.

Exemple 4.21 *Combien y a-t-il de façon de disposer* n *entiers distincts dans un tableau de taille* n. *On a* n *choix pour la première case du tableau. Pour chacun de ces choix, on a* $n-1$ *choix pour la seconde case. Il reste* $n-2$ *entiers possibles pour la troisième case, et ainsi de suite. Le nombre total de possibilités est donc* $n * (n-1) * (n-2) * \cdots * 2 * 1 = n!$.

4.5 Les séries

Une *série* est une fonction $S : \mathbb{R} \to \mathbb{R}$ de la forme suivante:

$$S(n) = \sum_{i=a}^{n} f(i)$$

où a est une constante et $f : \mathbb{R} \to \mathbb{R}$ est une fonction.

Plusieurs séries sont récurrentes en informatique et peuvent être exprimées sous une forme plus simple. En voici quelques unes:

Somme des entiers entre 1 et n:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Somme des carrés:

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(n+2)}{6}$$

Somme des cubes:

$$\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$$

Somme d'une progression géométrique:

$$\sum_{i=0}^{n} r^i = \frac{r^{n+1}-1}{r-1}$$

Somme des nombres impairs:

$$\sum_{i=1}^{n} (2i-1) = n^2$$

Somme de id^i:

$$\sum_{i=1}^{n} id^i = \left(\frac{d}{(d-1)^2}\right)[(nd-n-1)d^n+1]$$

Exemple 4.22 *Considérez la boucle suivante:*

```
s=0;
for (i=1; i<=n; i++)
    s=s+i;
```

Au sortir de la boucle on aura $s = n(n+1)/2$.

Exemple 4.23 *Considérez la boucle suivante:*

```
s=0;
for (i=0; i<n; i++)
    for (j=0; j<i; j++)
         s=s+1;
```

Au sortir de la boucle on aura $s = n(n+1)/2$.

Exemple 4.24 *Considérez la boucle suivante:*

```
s=0;
for (int i=1; i<=n; i++)
    s=s+i*i;
```

Au sortir de la boucle on aura $s = n(n+1)(n+2)/6$.

Exemple 4.25 *Considérez la boucle suivante:*

```
s=0;
for (i=0; i<n; i++)
    for (j=0; j<i; j++)
        for (k=0 ;k<i; k++)
            s=s+1;
```

Au sortir de la boucle on aura $s = n(n+1)(n+2)/6$.

Exemple 4.26 *Considérez la boucle suivante en supposant que* n *est une puissance de 2:*

```
s=0;
for (i=1; i<=n; i=i*2)
    for (j=0; j<i; j++)
            s=s+1;
```

Au sortir de la boucle on aura

$$s = \sum_{k=0}^{\log_2 n} n = n \log_2 n$$

Exemple 4.27 *Considérez la boucle suivante en supposant que n est une puissance de 2:*

```
s=0;
for (i=1; i<=n; i=i*2)
    for (j=0; j<i; j++)
            s=s+1;
```

Au sortir de la boucle on aura

$$s = \sum_{k=0}^{\log_2 n} 2^k = 2n - 1$$

5 Alphabets, mots et langages

Un *alphabet* A est un ensemble non vide de symboles appelés *lettres*. Un *mot sur* A est une suite fini de lettres appartenant à A. Si A est un alphabet, on dénote par A^* l'ensemble de tous les mots sur A. Puisque A est non vide, l'ensemble A^* contient toujours une infinité d'éléments. Un sous-ensemble $L \subseteq A^*$ est appelé *langage sur* A.

Exemple 5.1 *Considérons l'alphabet $A = \{a, b, c\}$. La suite $w = abbbc$ est un mot sur A. Remarquez qu'il est d'usage d'écrire abbbc plutot que (a, b, b, b, c). C'est cette notation standard pour les mots que nous utiliserons par la suite. L'ensemble $\{ac, abc, abbc, abbbc, abbbbc, \ldots\}$ est le langage sur A contenant tous les mots dont la première lettre est a, la dernière c et toutes les autres sont b. On a donc $w \in L$ et on dit que w est un mot du langage L.*

Puisque tout mot est une suite de caractères et puisqu'une suite peut contenir aucun élément alors il existe un mot qui est une suite de 0 caractère. Ce mot est appelé *mot vide* et est dénoté par la lettre grecque ε.

Il ne faut pas confondre le mot vide ε et le langage vide $\emptyset$: le premier est une suite et le second un ensemble. De plus, il ne faut pas confondre le langage vide $\emptyset = \{\}$ et le langage $\{\varepsilon\}$ qui ne contient que le mot vide.

Soit u et v deux mots sur un alphabet A. On dénote par $|u|$ le nombre de lettres dans le mots u. En particulier $|\varepsilon| = 0$. Les langages étant des ensembles, toutes les opérations que nous avons vues sur les ensembles s'appliquent au langages. Cependant trois opérations, appelées *opérations régulières*, sont

particulièrement importantes: l'union, la concaténation et l'opération étoile (aussi appelée *itération de Kleene*). Nous avons déjà vu l'union, nous allons maintenant décrire les deux autres opérations régulières.

L'opération de *concaténation* est une fonction $A^* \times A^* \to A^*$. Elle est définie de la façon suivante. Soit $a_1, \ldots, a_n, b_1, \ldots, b_m \in A$ et soit $u = a_1 \cdots a_n$ et $v = b_1 \cdots b_m$ deux mots dans A^* $(n, m \geq 0)$. La concaténation de u et v est le mot $uv = a_1 \cdots a_n b_1 \cdots b_m$, c'est-à-dire que uv est obtenu en ajoutant le mot v à la suite du mot u.

L'opération de concaténation peut être étendue aux langages. Elle consiste à prendre deux languages pour en définir un troisième. Si L_1 et L_2 sont deux languages sur l'alphabet A alors la concaténation L_1L_2 est définie par:

$$L_1L_2 = \{uv \in A^* \mid u \in L_1 \text{ et } v \in L_2\}$$

Remarque 1: Soit $w \in A^*$. Si $w \in L_1L_2$ alors il doit exister deux mots $u \in L_1$ et $v \in L_2$ tels que $w = uv$. Cela veut dire que si L_1 ou L_2 est vide alors L_1L_2 est aussi vide. On a donc pour tout langage $L \subseteq A^*$:

$$\emptyset L = L\emptyset = \emptyset$$

De ce point de vue, l'ensemble vide est à la concaténation des langages ce que le nombre 0 est à la multiplication des entiers.

Remarque 2: Poursuivant la remarque précédente, si $L_1 = \{\varepsilon\}$ alors $L_1L_2 = L_2$. Similairement, si $L_2 = \{\varepsilon\}$ alors $L_1L_2 = L_1$. On a donc pour tout langage $L \subseteq A^*$:

$$\{\varepsilon\}L = L\{\varepsilon\} = L$$

Le language $\{\varepsilon\}$ est donc à la concaténation des langages ce que le nombre 1 est à la multiplication des entiers.

Si $L \subseteq A^*$ est un langage alors pour tout $k \geq 0$ on défini:

$$L^k = \begin{cases} \{\varepsilon\} & \text{si } k = 0 \\ LL^{k-1} & \text{si } k > 0 \end{cases}$$

Remarque: La notation utilisée est justifiée entre autres par le fait qu'on peut vérifier que pour tout $i \geq 0$ et $j \geq 0$ on a $L^iL^j = L^{i+j}$. Cela démontre aussi la nécessité de définir $L^0 = \{\varepsilon\}$ car pour tout $k \geq 0$ on a $L^0L^k = L^kL^0 = L^k$ et que $L^0 = \{\varepsilon\}$ est la seule solution possible à cette équation.

Nous pouvons maintenant définir l'opération étoile qui consiste à prendre un langage pour en définir un autre. Pour tout langage $L \subseteq A^*$ on défini

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

Remarquez que l'alphabet A peut être vu comme un langage dans A^* et qu'appliquer l'opération étoile sur A donne bien l'ensemble A^*. Il n'y a donc aucune ambiguïté dans cette notation.

Exemple 5.2 *Nous avons vu que pour tout langage $L \subseteq A^*$ on a $\emptyset L = \emptyset$. En particulier, cela signifie que pour tout $k > 0$ on a $\emptyset^k = \emptyset$. Cependant puisque, par définition, $\emptyset^0 = \{\varepsilon\}$ alors $\emptyset^* = \{\varepsilon\}$.*

Exemple 5.3 *Soit $A = \{a, b, c\}$ un alphabet. Le singleton $\{b\}$ est un langage dans A^* ne contenant qu'un seul mot. Si on applique l'opération étoile sur $\{b\}$ on obtient*

$$\{b\}^* = \{\varepsilon, b, bb, bbb, \ldots\}$$

Si on concatène le langage $\{a\}$ avec $\{b\}^*$ on obtient

$$\{a\}\{b\}^* = \{a, ab, abb, abbb, \ldots\}$$

Finalement si on concatène ce dernier langage avec $\{c\}$ on obtient le langage

$$\{a\}\{b\}^*\{c\} = \{ac, abc, abbc, abbbc, \ldots\}$$

Le langage L contient tous les mots dont le premier caractère est a, le dernier est c et tous les autres sont b.