Introduction à l'informatique
Codage de l'information

Benjamin Monmege

2019/2020

Comment encode-t-on de l'information dans une machine, afin de calculer dessus par la suite ? Il est crucial de se mettre d'accord sur une façon d'encoder les informations puisque nous allons ensuite potentiellement devoir échanger de l'information entre nous. Il existe donc des standards internationaux. Mais la notion principale permettant d'établir ces standards est l'*abstraction* : nous allons construire petit à petit les encodages d'informations complexes, à partir d'informations plus simples.

1 Codage des entiers naturels

Commençons donc par coder des entiers naturels : $0, 1, 2, \ldots$ Prenons l'entier 24 par exemple. Hormis cette représentation d'un caractère « 2 » suivi d'un caractère « 4 », on peut opter pour la représentation latine « XXIV » ou même la représentation en français « vingt-quatre ». Pour nous aider à choisir une représentation, il faut savoir ce qu'on souhaite faire avec cette représentation. Ce qui fait privilégier l'écriture décimale « 24 » aux deux autres, c'est la simplicité de calculer avec cette représentation : on a ainsi bien du mal à ajouter « deux cent trente-quatre » et « deux cent quatre-vingt-un » si on pose l'addition :

```
      deux cent trente-quatre
+   deux cent quatre-vingt-un
-----------------------------
                          ???
```

La représentation décimale est pratique pour nous, puisque nous avons 10 doigts : nous pouvons donc nous aider de nos mains pour compter. Mais d'ailleurs, jusqu'à combien peut-on compter sur une main ? Une réponse naturelle consiste à répondre 5. Une autre réponse, a priori moins naturelle, consiste à préférer 31 en utilisant davantage de combinaisons de nos doigts levés ou pas… Pour cela, on doit passer d'une représentation décimale des entiers, à une représentation *binaire*, c'est-à-dire en base 2. Suivons la figure 1 pour compter de 0 à 31 :

— naturellement, on part de 0 qu'on représente avec la main fermée ;
— tout aussi naturellement, on représente 1 en levant le pouce ;
— un peu d'originalité : représentons 2 en abaissant le pouce, mais en relevant l'index ;
— puis 3 est obtenu en relevant le pouce ;
— la représentation de 4 n'est pas des plus élégantes, avec le seul majeur levé ;
— et 5 s'obtient en levant à nouveau le pouce.

On continue ainsi en passant d'un entier à l'autre
— en levant le pouce s'il est baissé,

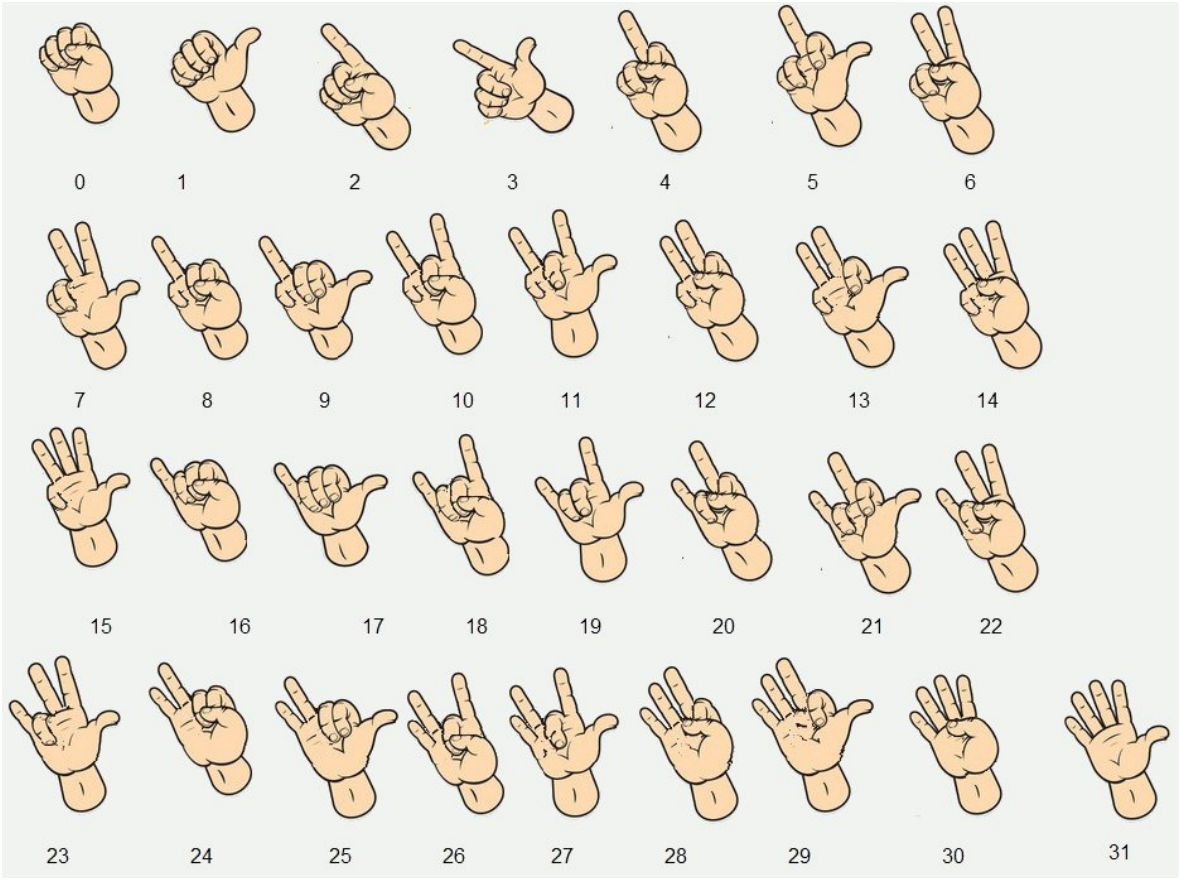

FIGURE 1 – Compter sur ses doigts en binaire

— ou alors en baissant le pouce et levant l'index s'il est baissé,
— ou alors en baissant le pouce et l'index et levant le majeur s'il est baissé...

L'entier 5 se représente donc par 101, alors que l'entier 6 est codé par 110 : 1 code un doigt levé, et 0 un doigt baissé. Pour mieux comprendre cette représentation en binaire, rappelons-nous ce que signifie la représentation décimale des entiers. Ainsi, l'entier 317 signifie qu'on a 3 centaines, 1 dizaine et 7 unités, représentant donc :

$$\begin{aligned} 317 &= 300 + 10 + 7 \\ &= 3 \times 10^2 + 1 \times 10^1 + 7 \times 10^0 \end{aligned}$$

C'est donc tout naturellement que la séquence 100111101 représente en binaire l'entier :

$$\begin{aligned} & 1 \times 2^8 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^0 \\ = {} & 256 + 32 + 16 + 8 + 4 + 1 \\ = {} & 317 \end{aligned}$$

Définition 1. Un bit est l'unité d'information la plus simple, pouvant prendre deux valeurs communément notée 0 et 1. On représente l'entier naturel 0 avec le bit 0. On représente un entier naturel non nul $a \in \mathbb{N}^* = \{1, 2, 3, \ldots\}$ par une *suite de bits* $a_{n-1}a_{n-2}\cdots a_1a_0$, avec $a_0, a_1, \ldots, a_{n-2}, a_{n-1} \in \{0, 1\}$, telle que $a_{n-1} = 1$ et

$$a = a_{n-1} \times 2^{n-1} + a_{n-2} \times 2^{n-2} + \cdots + a_1 \times 2^1 + a_0 \times 2^0 = \sum_{i=0}^{n-1} a_i 2^i$$

Cette suite est unique, grâce à la condition sur a_{n-1} : on l'appelle la *représentation binaire* de l'entier a.

Par exemple, la représentation binaire de 35 est 100011 car $35 = 2^5 + 2^1 + 2^0$ et celle de 16 est 10000 car $16 = 2^4$.

Il est donc important de connaitre, ou au moins pouvoir retrouver rapidement, les valeurs de 2^n pour n un petit entier naturel :

n	0	1	2	3	4	5	6	7	8	9	10
2^n	1	2	4	8	16	32	64	128	256	512	1024

Lorsqu'on fait des calculs en puissance de 2, on approche donc souvent 2^{10} avec 10^3.

2 Codage des entiers relatifs

Au-delà des entiers naturels, il est important de pouvoir coder des entiers relatifs : …,-3,-2,-1,0,1,2,3,… Il s'agit donc d'un entier *signé*. Une représentation naturelle d'un entier relatif $n \in \mathbb{Z}$ consiste donc à ajouter un *bit de signe* à gauche de la représentation en binaire de l'entier naturel $|n|$ correspondant à la valeur absolue de n : le bit de signe vaut 0 si $n \geq 0$ et 1 si $n < 0$. Ainsi, lorsqu'on code des entiers relatifs, le codage de 7 est 0111 et celui de -7 est 1111.

Notons qu'il ne s'agit pas de la représentation privilégiée dans les ordinateurs modernes : on utilise plutôt la représentation par *complément à deux*, mais nous n'en parlerons pas dans ce cours.

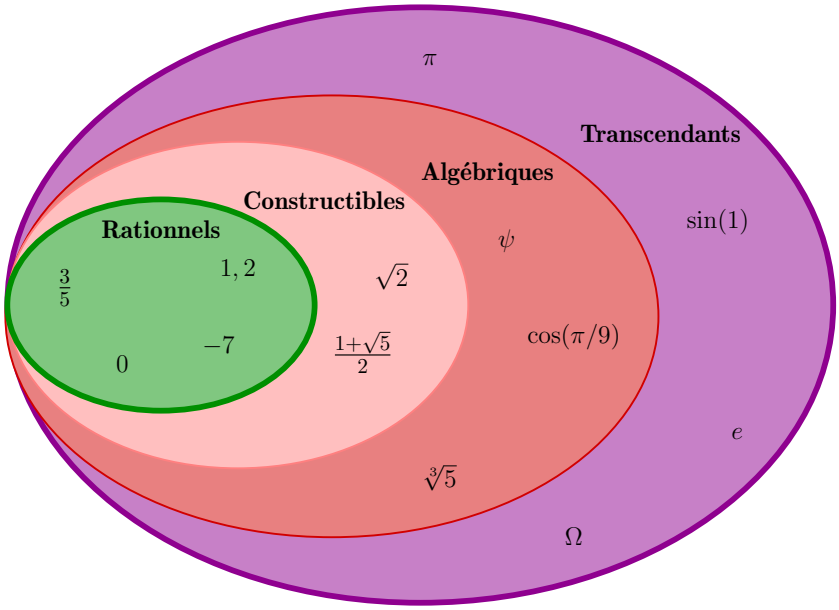

FIGURE 2 – Au-delà des entiers relatifs...

3 Codage flottant

Il n'y a pas que des entiers à savoir coder. On peut aussi vouloir coder des « nombres à virgule » en binaire : il existe une zoologie importante d'ensemble de nombres, au-delà des entiers relatifs (cf FIGURE 2). Coder un nombre rationnel $r \in \mathbb{Q}$ n'est pas très compliqué, une fois qu'on se rappelle qu'il s'agit d'un quotient $r = a/b$ avec $a \in \mathbb{Z}$ un entier relatif et $b \in \mathbb{N}^*$ un entier naturel non nul. On a donc vu comment représenter a et b précédemment : fort de cette abstraction, on peut donc représenter le rationnel r comme une *paire* (a, b).

Mais on utilise finalement assez peu les nombres rationnels, et on préfèrerait pouvoir écrire des « nombres à virgule » plus généraux en binaire. Par exemple, de même que le nombre $3,14$ vaut $3+\frac{1}{10^1}+\frac{4}{10^2}$, en binaire on peut écrire le nombre à virgule $1,011$ valant $1+\frac{0}{2^1}+\frac{1}{2^2}+\frac{1}{2^3}$. On est cependant encore limité. Par exemple, les physiciens ont souvent besoin de pouvoir encoder des grands entiers ou des nombres rationnels sous la forme de leur notation scientifique, tel que le nombre d'Avogadro par exemple qui décrit le nombre d'entités élémentaires (atomes, molécules ou ions) dans une *mole* de matières :

$$N_A = 6,022\ 140\ 857 \times 10^{23}\,\mathrm{mol}^{-1}$$

ou la constante universelle de gravitation apparaissant dans la loi universelle de la gravitation de Newton :

$$G = 6,674\ 08 \times 10^{-11}\,\mathrm{m}^3\,\mathrm{kg}^{-1}\,\mathrm{s}^{-2}$$

Pour stocker de tels nombres, on utilise la notation scientifique en binaire, appelée *représentation flottante*, pour approcher des nombres réel.

Définition 2. On considère donc des réels pouvant s'écrire sous la forme

$$s\ m \times 2^k$$

avec s le signe (+ ou -) du réel, m un nombre à virgule compris entre 1 inclus et 2 exclus, et k son exposant. En *simple précision* (c'est-à-dire quand on se réserve 32 bits pour stocker le nombre flottant),

— 1 bit est utilisé pour représenté le signe s (0 pour le signe +, 1 pour le signe -) ;
— 8 bits pour l'exposant k : l'exposant est un entier relatif entre -126 et 127 qu'on représente par l'entier naturel $k+127$ qui est donc compris entre 1 et 254 (avec 8 bits, on peut aussi représenter les deux exposants 0 et 255, qui sont cependant réservés pour des situations exceptionnelles telles que $+\infty$, $-\infty$, etc.) ;
— et 23 bits pour le nombre m qu'on appelle *mantisse* : puisqu'on représente m en binaire et qu'on le choisit dans l'intervalle $[1;2[$, le nombre avant la virgule vaut toujours 1 et on ne le stocke donc pas en machine, utilisant ainsi les 23 bits pour les chiffres après la virgule.

Par exemple, la séquence de bits

$$10101001111001000110000000000000$$

est la représentation flottante du réel $-\frac{1827}{1024} \times 2^{-44} \approx -1,01 \times 10^{-13}$ puisque :

— le signe est encodé par le premier 1, et est donc - ;
— l'exposant est encodé par les huit bits suivants, 01010011, représentation binaire de l'entier 83, impliquant que l'exposant vaut $83 - 127 = -44$;
— la représentation en binaire de la mantisse est $1,11001000110000000000000$ qui vaut :

$$1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^5} + \frac{1}{2^9} + \frac{1}{2^{10}} = \frac{2^{10} + 2^9 + 2^8 + 2^5 + 2 + 1}{2^{10}} = \frac{1827}{1024}$$

Mais pourquoi fait-on ce décalage de +127 pour calculer l'exposant ? On a vu que l'exposant est codé sur 8 bits : on a donc toutes les possibilités entre 00000000 et 11111111, c'est-à-dire entre 0 et 255. On a ainsi 256 codages possibles pour l'exposant. Puisqu'on veut pouvoir avoir des exposants positifs et d'autres négatifs, on répartit ces 256 codages possibles entre les négatifs et les positifs : on choisit arbitrairement d'aller de -127 à 128. Oui mais voilà, la convention IEEE-754, qui a en charge de mettre tout le monde d'accord sur le format des nombres flottants, en a décidé autrement : elle s'est réservée les deux extrémités (-127 et 128) pour des usages exceptionnels, en particulier pour écrire le nombre $0,0$ (on décide alors de l'encoder avec 32 bits à 0), des flottants qui ne sont pas des nombres (lorsqu'on vient de faire une division par zéro interdite par exemple) et + ou − l'infini... Ainsi, on se restreint à l'intervalle $[-126;127]$. Maintenant, il faut représenter ça sur 8 bits : pour faire simple, on décale tout de 127, pour arriver à l'intervalle [1,254] qui est donc codé par les codages binaires de 00000001 à 11111110 (réservant les codes extrémaux 00000000 et 11111111 pour les usages exceptionnels).

Attention, choisir de représenter un nombre à l'aide d'un codage à virgule (flottante) induit nécessairement des imprécisions qui peuvent être préjudiciables si on n'y prend pas garde. Par exemple, considérons le nombre $0,1$. Comment s'écrit-il en binaire ? Il faut qu'on essaie de l'écrire avec un nombre à virgule, c'est-à-dire comme une somme de puissance de 2. Puisque $0,1 \in [0,0625;0,125] = [\frac{1}{2^4};\frac{1}{2^3}]$, le premier 1 dans l'écriture à virgule est à la quatrième position : l'écriture commence donc par $0,0001...$ et l'imprécision restante est de $0,1 - 0,0625 = 0,0375$. Puisque $\frac{1}{2^5} = 0,03125$, la prochaine puissance de 2 doit être prise : l'écriture binaire commence donc par $0,00011...$ et il reste $0,0375 - 0,03125 = 0,00625 = 0,1 \times \frac{1}{2^4}$. On voit réapparaître le nombre $0,1$ duquel on était parti, signe qu'on s'apprête à

écrire un nombre infini de chiffres après la virgule (comme lorsqu'on essaie d'écrire le rationnel 1/3 comme un chiffre à virgule en décimal...). Pour être plus précis, résumons nos calculs précédents avec l'équation

$$0,1 = \frac{1}{2^4} + \frac{1}{2^5} + 0,1 \times \frac{1}{2^4}$$

dans laquelle on peut remplacer le $0,1$ à droite par cette même écriture

$$0,1 = \frac{1}{2^4} + \frac{1}{2^5} + \left(\frac{1}{2^4} + \frac{1}{2^5} + 0,1 \times \frac{1}{2^4}\right) \times \frac{1}{2^4} = \frac{1}{2^4} + \frac{1}{2^5} + \frac{1}{2^8} + \frac{1}{2^9} + 0,1 \times \frac{1}{2^8}$$

et ainsi de suite, de sorte qu'on trouve finalement le nombre binaire à virgule :

$$0,00011001100110011\ldots$$

On peut en déduire l'écriture en virgule flottante en faisant glisser le premier 1 avant la virgule, de sorte que $0,1$ vaut, en binaire,

$$1,10011001100110011001100110011\ldots \times 2^{-4}$$

En simple précision, on a donc :

— un bit de signe à 0 puisque le nombre $0,1$ est positif ;
— l'exposant qui vaut $-4 + 127 = 123$, encodé en binaire par les huit bits 01111011 ;
— les 23 bits de mantisse étant le début de ce qui suit la virgule ci-dessous, à savoir 10011001100110011001100.

La représentation en virgule flottante de $0,1$ est donc

$$0\,01111011\,10011001100110011001100$$

En faisant cela, on a introduit une imprécision de l'ordre de $2^{-27} \approx 7 \times 10^{-9}$. En particulier, si on fait exécuter par une machine (Python par exemple) le calcul $0.1 + 0.1 + 0.1$, on ne trouvera pas la même chose que 0.3 : un test d'égalité entre ces deux valeurs renverra donc faux, à la surprise des programmeurs en herbe ! Rappelez-vous donc qu'on ne fait jamais de test d'égalité entre deux nombres à virgule flottante : à la place, on préfère tester si la différence entre ces deux nombres est suffisamment petite vis-à-vis de la précision voulue.

4 Codage de texte

La prochaine étape, une fois qu'on sait stocker en machine des nombres, consiste à pouvoir stocker du texte, que ce soit le contenu d'un livre, un e-mail ou bien un mot de passe. Pour cela, faisons appel à l'abstraction : pour ne pas tout reprendre depuis le début, on peut utiliser le fait que nous savons désormais coder des entiers. Ainsi, on représente chaque caractère imprimable (« A », « h », « $ », mais aussi « 7 » ou « + ») par un entier. Il s'agit donc de se mettre d'accord afin que tout le monde utilise un codage que les autres peuvent comprendre. Le codage le plus simple consiste à utiliser la table ASCII (c'est l'acronyme de *American Standard Code for Information Interchange*). Cette table contient 128 caractères chacun associé avec un code (décimal) entre 0 et 127 : son contenu est représenté dans la Figure 3. Par exemple, la lettre « A » est codée par l'entier 65. Dans la machine, cet entier est évidemment stocké en binaire : son code binaire est 1000001. Le symbole « # » est représenté par l'entier 35. La table ASCII contient également le code d'éléments utiles pour coder du texte ou des touches du clavier

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

FIGURE 3 – Table ASCII

(la touche « escape », le « retour à la ligne » par exemple ou l'« espace », de codes ASCII respectifs 27, 13 et 32).

Puisqu'on sait coder les caractères, on sait aussi coder une *chaîne de caractères*, c'est-à-dire du texte. Ainsi la phrase « Dessine-moi un mouton. » est codée par la séquence d'entiers :

68 101 115 115 105 110 101 45 109 111 105 32 117 110 32 109 111 117 116 111 110 46

En réalité, évidemment, on ne stocke pas des entiers en décimal, mais bien des entiers codés en binaire, chacun sur 7 bits : par exemple, le tiret « - » est donc codé par la séquence de bits 0101101, où on a donc ajouté un 0 en premier pour utiliser les 7 bits à disposition. Puisque chaque caractère est codé sur 7 bits, on a donc pas besoin de « séparer » les codages de chaque caractère (comme ci-dessus).

Il existe d'autre façons de coder des caractères : citons par exemple d'autres tables, telles que l'UTF-8 (ou UTF-16, ou UTF-32) ou sa généralisation, l'Unicode, permettant de coder bien davantage de caractères (les caractères accentués ou les caractères d'autres alphabets que l'alphabet latin).

5 Codage d'images

On sait désormais coder un roman, mais pas une bande dessinée : il nous manque la possibilité de coder des images. Un format simple (mais gourmand en espace) consiste à représenter une image comme un tableau bidimensionnel (une matrice) : chaque élément du tableau est alors appelé un *pixel*. Le codage d'un pixel consiste à représenter la couleur de la zone correspondante de l'image. Pour ce faire, on utilise généralement trois entiers qui représentent les quantités (entre 0 et 255) de rouge, de vert et de bleu dans la couleur. Par exemple, dans la FIGURE 4, une zone de l'œil de la Joconde est agrandie : cette zone comporte

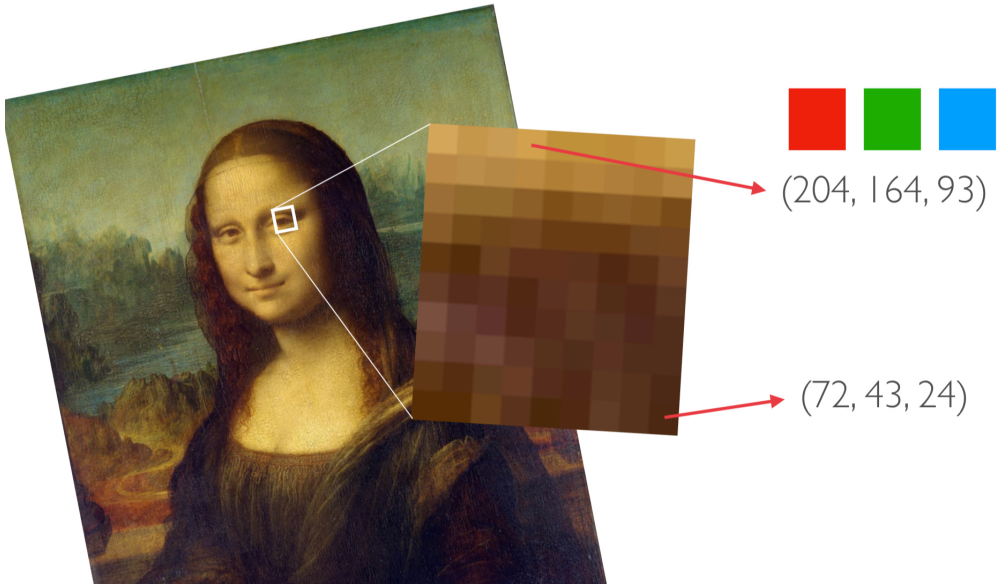

FIGURE 4 – Codage d'une image en format bitmap

9 colonnes et 10 lignes de pixels. Deux de ces pixels sont détaillés à droite. Celui du haut a une couleur marron pale qui est codée par le triplet (204, 164, 93) : la couleur est donc constituée de rouge à hauteur de $204/(204 + 164 + 93) = 44,25\%$. Ainsi, la couleur verte à 100% sera représentée par le triplet (0, 255, 0).

Notons que $255 = 2^8 - 1$. Ainsi, la représentation de 255 en binaire est 11111111. Pour coder un entier entre 0 et 255, on a donc besoin de 8 bits. Cette quantité de bits se retrouve souvent en informatique :

Définition 3. Une séquence de 8 bits s'appelle un *octet*.

On utilise les octets pour rendre également plus lisible la représentation des entiers. Par exemple, l'entier 10^8 a pour représentation binaire 101111101011110000100000000. C'est bien difficile à lire ! De la même façon qu'on peut séparer les chiffres par groupes de 3 dans l'écriture décimale d'un nombre (par exemple, 100 000 000), on choisit de séparer les bits de la représentation binaire par groupes de 8 : 101 11110101 11100001 00000000. On utilise souvent l'octet comme unité de capacité mémoire de disques dans le commerce : 256 Go signifie ainsi 256 Giga octets, soit 256 milliards d'octets, ce qui représente donc $256 \times 8 = 2048$ milliards de bits.

Revenons au codage des images. Stocker une image de 1920 par 1200 pixels nécessite de représenter $1920 \times 1200 = 2\ 304\ 000$ pixels, chacun prenant 3 octets en mémoire : au total, cette image prend donc 6 912 000 octets, soit 55 296 000 bits, ce qui équivaut à 55 Mégabits. C'est beaucoup pour une simple image... En pratique, on ne stocke donc que rarement les images en format bitmap : on utilise plutôt des formats *compressés* qui essaient d'épargner de la mémoire en profitant de redondances dans l'image ou en supprimant des détails invisibles à l'œil nu.

6 Codage de vidéos

Toujours grâce au principe d'abstraction, on utilise le fait qu'on sait désormais coder une image pour pouvoir coder une vidéo comme une séquence d'images (environ 24 par secondes, par exemple, afin que l'œil ne puisse pas distinguer les images qui défilent). Là aussi, on n'utilise pas cette représentation naïve en pratique, mais des formats compressés permettent de gagner de la place en mémoire. Pour stocker une vidéo, il faut aussi pouvoir stocker du son ce qui est un problème plus complexe encore qu'on ne traitera pas dans ce cours.