Cours Turing
Semaine 13

1 Représentation binaire des nombres et cie

Représentation binaire des nombres entiers positifs

Pour représenter un nombre entier positif N en binaire, il importe d'écrire celui-ci comme une somme de puissances de 2, de la même façon que la représentaion décimale de ce même nombre utilise les puissances de 10. Ainsi, nous avons :

$$1984 = 1000 + 900 + 80 + 4 = \mathbf{1} \cdot 10^3 + \mathbf{9} \cdot 10^2 + \mathbf{8} \cdot 10^1 + \mathbf{4} \cdot 10^0$$

d'où la représentation décimale "1984". Voici maintenant la même chose en binaire :

$$\begin{aligned} 1984 &= 1024 + 512 + 256 + 128 + 64 \\ &= \mathbf{1} \cdot 2^{10} + \mathbf{1} \cdot 2^9 + \mathbf{1} \cdot 2^8 + \mathbf{1} \cdot 2^7 + \mathbf{1} \cdot 2^6 + \mathbf{0} \cdot 2^5 + \mathbf{0} \cdot 2^4 + \mathbf{0} \cdot 2^3 + \mathbf{0} \cdot 2^2 + \mathbf{0} \cdot 2^1 + \mathbf{0} \cdot 2^0 \end{aligned}$$

d'où la représentation binaire "11111000000".

Avec n bits, il est possible de représenter tous les nombres entiers allant de $0\,(= 000\ldots0)$ à $2^n - 1\,(= 111\ldots1)$. En particulier, avec 8 bits (équivalant à 1 octet), on peut représenter tous les nombres de 0 à $2^8 - 1 = 255$.

A l'intérieur d'un ordinateur, les nombres sont toujours enregistrés au format binaire, mais pour favoriser la communication entre les humains et les ordinateurs, on utilise plutôt la représentation décimale des nombres en Python. Toutefois, il est possible d'utiliser directement la représentation binaire des nombres en Python, en faisant commencer un nombre par "0b", suivi de sa représentation binaire. Ainsi, vous pouvez tester par exemple les égalités suivantes : $12 == 0b1100$, $22 == 0b10110$ ou encore $32 == 0b100000$. Toutes ces égalités ont la valeur True en Python.

Pour faire apparaître la représentation binaire d'un nombre entier, il est possible d'utiliser la fonction bin(). Par exemple, bin(14)="0b1110". Mais attention ! bin(14) est une chaîne de caractères (str) et non la représentation binaire du nombre 14. Encore une fois, 14 est enregistré directement sous format binaire dans votre ordinateur : celui-ci n'a pas besoin d'effectuer de conversion. La fonction bin() est là pour nous permettre de visualiser la représentation binaire d'un nombre entier sous forme de chaîne de caractères.

Représentation hexadécimale des nombres entiers positifs

Bien sûr, s'il est pratique de représenter tous les nombres en binaires à l'intérieur d'un ordinateur, il est aussi possible de choisir une autre base que la base 2 pour faire des calculs. En informatique, la base 16 (représentation "hexadécimale") est souvent utilisée : les chiffres dans cette base sont $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f$ (les derniers "chiffres" a, b, c, d, e, f de cette série ayant pour valeurs respectives $10, 11, 12, 13, 14, 15$ en décimal). Un chiffre hexadécimal peut lui-même être représenté sur 4 bits, et s'écrit en Python précédé de "0x". Ainsi, les égalités suivantes ont la valeur True en Python :

$$12 == 0xc, \quad 140 == 0x8c, \quad 0xa == 0b1010, \quad 0xff == 0b11111111$$

De même que pour la représentation binaire, il existe une fonction en Python permettant d'obtenir la représentation hexadécimale d'un nombre entier, c'est la fonction hex(). Ainsi, hex(61)="0x3d" (mais notez bien à nouveau que ceci est une chaîne de caractères et non le nombre en lui-même).

Représentation binaire des caractères

Dans le code ASCII étendu, un caractère est encodé sous la forme d'un nombre entier allant de 0 à 255 (et donc représentable sous forme binaire par 1 octet) :

ASCII control characters

00	NULL	(Null character)
01	SOH	(Start of Header)
02	STX	(Start of Text)
03	ETX	(End of Text)
04	EOT	(End of Trans.)
05	ENQ	(Enquiry)
06	ACK	(Acknowledgement)
07	BEL	(Bell)
08	BS	(Backspace)
09	HT	(Horizontal Tab)
10	LF	(Line feed)
11	VT	(Vertical Tab)
12	FF	(Form feed)
13	CR	(Carriage return)
14	SO	(Shift Out)
15	SI	(Shift In)
16	DLE	(Data link escape)
17	DC1	(Device control 1)
18	DC2	(Device control 2)
19	DC3	(Device control 3)
20	DC4	(Device control 4)
21	NAK	(Negative acknowl.)
22	SYN	(Synchronous idle)
23	ETB	(End of trans. block)
24	CAN	(Cancel)
25	EM	(End of medium)
26	SUB	(Substitute)
27	ESC	(Escape)
28	FS	(File separator)
29	GS	(Group separator)
30	RS	(Record separator)
31	US	(Unit separator)
127	DEL	(Delete)

ASCII printable characters

32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Extended ASCII characters

128	Ç	160	á	192	└	224	Ó
129	ü	161	í	193	┴	225	ß
130	é	162	ó	194	┬	226	Ô
131	â	163	ú	195	├	227	Ò
132	ä	164	ñ	196	─	228	õ
133	à	165	Ñ	197	┼	229	Õ
134	å	166	ª	198	ã	230	µ
135	ç	167	º	199	Ã	231	þ
136	ê	168	¿	200	╚	232	Þ
137	ë	169	®	201	╔	233	Ú
138	è	170	¬	202	╩	234	Û
139	ï	171	½	203	╦	235	Ù
140	î	172	¼	204	╠	236	ý
141	ì	173	¡	205	═	237	Ý
142	Ä	174	«	206	╬	238	¯
143	Å	175	»	207	¤	239	´
144	É	176	▒	208	ð	240	≡
145	æ	177	▒	209	Ð	241	±
146	Æ	178	▓	210	Ê	242	‗
147	ô	179	│	211	Ë	243	¾
148	ö	180	┤	212	È	244	¶
149	ò	181	Á	213	ı	245	§
150	û	182	Â	214	Í	246	÷
151	ù	183	À	215	Î	247	¸
152	ÿ	184	©	216	Ï	248	°
153	Ö	185	╣	217	┘	249	¨
154	Ü	186	║	218	┌	250	·
155	ø	187	╗	219	▒	251	¹
156	£	188	╝	220	▄	252	³
157	Ø	189	¢	221	¦	253	²
158	×	190	¥	222	Ì	254	■
159	ƒ	191	┐	223	▀	255	nbsp

Pour représenter une chaîne de n caractères sous forme binaire, n octets sont donc nécessaires.

Notez bien que tout type d'information, que ce soit un nombre, une chaîne de caractères ou une structure de données plus complexe, est toujours représentée sous la forme d'une séquence de 0 et de 1 à l'intérieur d'un ordinateur.

2 Opérations logiques élémentaires

De la même manière, à l'intérieur d'un ordinateur, toutes les opérations effectuées, que ce soit sur des nombres, des chaînes de caractères, des listes, des dictionnaires ou autres, se ramènent toujours au niveau le plus élémentaire à des opérations sur les 0 et les 1 qui constituent ces données. Voici les "tables de vérité" des quatre opérations de base à partir desquelles on peut recréer tout type d'opérations plus complexes (comme par exemple des additions, des multiplications) :

x	$\sim x$
0	1
1	0

NOT

x	y	$x\&y$
0	0	0
0	1	0
1	0	0
1	1	1

AND

x	y	$x\|y$
0	0	0
0	1	1
1	0	1
1	1	1

OR

x	y	$x \wedge y$
0	0	0
0	1	1
1	0	1
1	1	0

XOR

On parle ici d'opérations "logiques", car si on interprète 1 comme "vrai" et 0 comme "faux", alors on peut interpréter ces opérations comme des propositions logiques :

- négation (NOT) : $\sim x$ est une proposition vraie si et seulement si x ne l'est pas ;
- conjonction (AND) : $x\&y$ est une proposition vraie si et seulement si x est vraie *et* y est vraie (cf. ligne 4 de la table de vérité) ;
- disjonction (OR) : $x|y$ est une proposition vraie si et seulement si x est vraie *ou* y est vraie (cf. lignes 2,3,4 de la table de vérité) ;
- "ou exclusif" (XOR) : $x \wedge y$ est une proposition vraie si et seulement si x est vraie *ou* y est vraie, mais pas les deux en même temps (cf. lignes 2,3 de la table de vérité).

3 Opérations logiques sur des nombres entiers

En Python, voici ce qu'on obtient lorsqu'on utilise ces opérations logiques entre deux nombres entiers positifs x et y (et non deux bits 0 et 1) : $x\&y$, par exemple, est un nombre entier positif dont chaque bit de la représentation binaire est le résultat de l'opération & appliquée sur les bits des deux nombres x et y placés en même position. Voici un exemple :

$$13 \ \& \ 7 = 0b1101 \ \& \ 0b0111 = 0b0101 = 5$$

Il en va de même pour les opérations | et $\wedge$ (pour éviter de potentielles confusions, nous ne parlerons pas ici de l'opération $\sim$ ici) :

$$13 \mid 7 = 0b1101 \mid 0b0111 = 0b1111 = 15$$

et

$$13 \ \wedge \ 7 = 0b1101 \ \wedge \ 0b0111 = 0b1010 = 10$$

Note : Vous aurez sans doute remarqué au passage qu'on a ici l'égalité : $x|y = x\&y + x \wedge y$; c'est en fait toujours le cas : voyez-vous pourquoi ?

Il est clair à part ça que ces trois opérations sur les nombres entiers sont tout sauf intuitives... Seule une grande familiarité avec la représentation binaire des nombres (que clairement nous humains ne possédons pas en général) permettrait d'effectuer de telles opérations avec aisance.

Finalement, nous aurons encore besoin de deux autres opérations, notées << ("décalage à gauche") et >> ("décalage à droite") dont la fonction est de décaler (comme leur nom l'indique...) les bits qui composent un nombre d'un certain nombre de positions.

Par exemple, $x << 3$ veut dire décaler de 3 positions vers la gauche les bits qui composent le nombre x (en rajoutant des 0). Si par exemple $x = 156 = 0b10011100$ en binaire, alors $x << 3 = 0b10011100000 = 1248 = 156 \cdot 8$, vu que chaque décalage vers la gauche en binaire revient à multiplier le nombre par 2.

De même, $x >> 3$ décale de 3 positions vers la droite les bits du nombre x, en laissant éventuellement tomber les 1 qui se trouveraient aux 3 dernières places. Par exemple, si $x = 156 = 0b10011100$, alors $x >> 3 = 0b10011 = 19$ (qui n'est rien d'autre que le quotient de 156 par 8, où le reste de la division est négligé).

4 Algorithme Xorshift (Marsaglia, 2003)

Dans sa version la plus simple, l'idée de l'algorithme est la suivante :

1. partir d'une "graine", c'est-à-dire un nombre entier positif x représenté en binaire sur 32 bits et tiré au hasard d'une façon ou d'une autre ;
2. effectuer un XOR de x avec une version décalée vers la gauche de x pour générer le prochain nombre ;
3. ceci ne suffit toutefois pas (vous comprendrez pourquoi en faisant l'exercice correspondant) : il faut encore effectuer *deux fois* une opération similaire pour obtenir le prochain nombre x ;
4. repartir du point 2 avec ce nouveau nombre x.

Cet algorithme génère une suite de nombres qui est proche d'une suite aléatoire, si les décalages sont judicieusement choisis. Dans le détail, voici une possibilité pour les trois décalages successifs :

$$\begin{aligned} x &= (x \wedge (x << 13)) \,\&\, 0xffffffff \\ x &= (x \wedge (x >> 17)) \,\&\, 0xffffffff \\ x &= (x \wedge (x << 5)) \;\&\, 0xffffffff \end{aligned}$$

Chaque fin de ligne, une opération & avec le nombre $0xffffffff = 2^{32} - 1$ est effectuée pour assurer que les nombres entiers positifs ainsi générés ne deviennent pas plus grands que $2^{32} - 1$.

Notez que de nombreuses autres versions de l'algorithme existent, qui ont pour but de perfectionner celui-ci. Notamment, plus l'algorithme est sophistiqué, plus celui-ci passe un nombre important de tests qui vérifient le caractère aléatoire de la séquence ainsi produite : mais rentrer dans ces détails nous emmènerait un peu trop loin...