

LA CODIFICA DELLE INFORMAZIONI ALFANUMERICHE

CODIFICA ASCII

Definizione: con il termine **stringa di caratteri o stringa di testo**, o semplicemente **stringa**, indichiamo una qualsiasi sequenza di simboli che possono essere: lettere (in qualsiasi lingua), cifre, caratteri speciali, come ad esempio la virgola, i due punti, lo spazio bianco, emoticon ecc...

ESEMPI DI STRINGHE

un codice fiscale (16 caratteri) **LNAGMR75L23B149Q**

la targa di un'automobile (7 caratteri): **AX495JG**

una frase: **Pietro ha 25 anni, Paolo 24.** stringa con 28 caratteri (**anche virgola e spazi bianchi sono caratteri**)

Quando una serie di cifre viene memorizzata come una stringa, nella memoria non viene posto il valore binario del numero, ma semplicemente un codice che rappresenta ciascuna cifra. Pertanto se una sequenza di cifre viene memorizzata come stringa non è possibile utilizzarla per eseguire calcoli aritmetici.

Ad esempio, il software Word (o qualsiasi altro software per videoscrittura) non è in grado di svolgere il seguente calcolo:

$$7+5$$

perché il software non memorizza i valori numerici ma semplicemente una sequenza di tre codici, uno per il simbolo "7", uno per il simbolo "+" e uno per il simbolo "5".

Abbiamo visto come si possono memorizzare nelle memorie i numeri interi positivi e negativi e i numeri reali, ossia abbiamo capito come sono stati convertiti, attraverso opportune codifiche, in sequenze di 1 e 0.

Come è possibile memorizzare le stringhe?

Anche esse dovranno essere trasformate in una sequenza di 1 e 0. Vediamo come si fa.

Si utilizza un codice che fa corrispondere ad ogni simbolo un numero intero binario ad 8 bit e quindi può essere memorizzato in un byte della memoria. Tale codice si chiama codice **ASCII** (American Standard Code for Information Interchange) ed è stato ideato dall' **ANSI** (American National Standard Institute) nel 1968.

TABELLE DEL CODICE ASCII

Byte	Cod	Char	Byte	Cod	Char	Byte	Cod	Char	Byte	Cod	Char
00000000	0	Null	00100000	32	SpC	01000000	64	@	01100000	96	`
00000001	1	Start of heading	00100001	33	!	01000001	65	A	01100001	97	a
00000010	2	Start of text	00100010	34	"	01000010	66	B	01100010	98	b
00000011	3	End of text	00100011	35	#	01000011	67	C	01100011	99	c
00000100	4	End of transmit	00100100	36	\$	01000100	68	D	01100100	100	d
00000101	5	Enquiry	00100101	37	%	01000101	69	E	01100101	101	e
00000110	6	Acknowledge	00100110	38	&	01000110	70	F	01100110	102	f
00000111	7	Audible bell	00100111	39	'	01000111	71	G	01100111	103	g
00001000	8	Backspace	00101000	40	(	01001000	72	H	01101000	104	h
00001001	9	Horizontal tab	00101001	41	)	01001001	73	I	01101001	105	i
00001010	10	Line feed	00101010	42	*	01001010	74	J	01101010	106	j
00001011	11	Vertical tab	00101011	43	+	01001011	75	K	01101011	107	k
00001100	12	Form Feed	00101100	44	,	01001100	76	L	01101100	108	l
00001101	13	Carriage return	00101101	45	-	01001101	77	M	01101101	109	m
00001110	14	Shift out	00101110	46	.	01001110	78	N	01101110	110	n
00001111	15	Shift in	00101111	47	/	01001111	79	O	01101111	111	o
00010000	16	Data link escape	00110000	48	0	01010000	80	P	01110000	112	p
00010001	17	Device control 1	00110001	49	1	01010001	81	Q	01110001	113	q
00010010	18	Device control 2	00110010	50	2	01010010	82	R	01110010	114	r
00010011	19	Device control 3	00110011	51	3	01010011	83	S	01110011	115	s
00010100	20	Device control 4	00110100	52	4	01010100	84	T	01110100	116	t
00010101	21	Neg. acknowledge	00110101	53	5	01010101	85	U	01110101	117	u
00010110	22	Synchronous idle	00110110	54	6	01010110	86	V	01110110	118	v
00010111	23	End trans. block	00110111	55	7	01010111	87	W	01110111	119	w
00011000	24	Cancel	00111000	56	8	01011000	88	X	01111000	120	x
00011001	25	End of medium	00111001	57	9	01011001	89	Y	01111001	121	y
00011010	26	Substitution	00111010	58	:	01011010	90	Z	01111010	122	z
00011011	27	Escape	00111011	59	;	01011011	91	[	01111011	123	{
00011100	28	File separator	00111100	60	<	01011100	92	\	01111100	124	
00011101	29	Group separator	00111101	61	=	01011101	93	]	01111101	125	}
00011110	30	Record Separator	00111110	62	>	01011110	94	^	01111110	126	~
00011111	31	Unit separator	00111111	63	?	01011111	95	_	01111111	127	Del

Byte	Cod	Char	Byte	Cod	Char	Byte	Cod	Char	Byte	Cod	Char
10000000	128	Ç	10100000	160	á	11000000	192	+	11100000	224	Ô
10000001	129	ü	10100001	161	í	11000001	193	-	11100001	225	ß
10000010	130	é	10100010	162	ó	11000010	194	-	11100010	226	Ô
10000011	131	â	10100011	163	ú	11000011	195	+	11100011	227	Ö
10000100	132	ä	10100100	164	ñ	11000100	196	-	11100100	228	ö
10000101	133	à	10100101	165	Ñ	11000101	197	+	11100101	229	Ö
10000110	134	ä	10100110	166	ª	11000110	198	ä	11100110	230	µ
10000111	135	ç	10100111	167	º	11000111	199	Ä	11100111	231	þ
10001000	136	ê	10101000	168	¿	11001000	200	+	11101000	232	Ð
10001001	137	ë	10101001	169	®	11001001	201	+	11101001	233	Ú
10001010	138	è	10101010	170	¬	11001010	202	-	11101010	234	Û
10001011	139	ï	10101011	171	½	11001011	203	-	11101011	235	Ü
10001100	140	î	10101100	172	¼	11001100	204	ì	11101100	236	ý
10001101	141	ï	10101101	173	í	11001101	205	-	11101101	237	ÿ
10001110	142	Ä	10101110	174	«	11001110	206	+	11101110	238	-
10001111	143	Å	10101111	175	»	11001111	207	å	11101111	239	·
10010000	144	Ê	10110000	176	-	11010000	208	ð	11110000	240	-
10010001	145	æ	10110001	177	-	11010001	209	Ð	11110001	241	±
10010010	146	Æ	10110010	178	-	11010010	210	Ê	11110010	242	-
10010011	147	ô	10110011	179	-	11010011	211	Ë	11110011	243	¼
10010100	148	ö	10110100	180	-	11010100	212	È	11110100	244	¶
10010101	149	ò	10110101	181	À	11010101	213	ì	11110101	245	§
10010110	150	û	10110110	182	Â	11010110	214	í	11110110	246	÷
10010111	151	ù	10110111	183	Ã	11010111	215	î	11110111	247	°
10011000	152	ÿ	10111000	184	©	11011000	216	Ï	11111000	248	°
10011001	153	Ö	10111001	185	-	11011001	217	+	11111001	249	°
10011010	154	Ü	10111010	186	-	11011010	218	+	11111010	250	°
10011011	155	ø	10111011	187	+	11011011	219	-	11111011	251	°
10011100	156	£	10111100	188	+	11011100	220	-	11111100	252	°
10011101	157	Ø	10111101	189	£	11011101	221	-	11111101	253	°
10011110	158	×	10111110	190	¥	11011110	222	Ì	11111110	254	-
10011111	159	f	10111111	191	+	11011111	223	-	11111111	255	-

Perché sono stati utilizzati proprio 8 bit per rappresentare ogni simbolo? Perché non 10 o 4 ???

In realtà la prima versione del codice ASCII era a 7 bit, poi è stato esteso ad 8 bit. Perché? Spieghiamo: con 7 bit è possibile realizzare $2^7 = 128$ combinazioni possibili di 1 e 0. Associando a ciascuna di queste combinazioni un carattere alfanumerico (cioè un simbolo), si è potuto rappresentare 128 simboli, quelli che vediamo nella prima tabella, dove sono mostrati sia i codici in binario sia la loro equivalente “conversione” in decimale (i primi 31 sono caratteri speciali chiamati “di controllo”, sono utilizzati nelle comunicazioni fra computer e non sono visibili sui file di testo).

Quando, negli anni '70, l'utilizzo del PC si è diffuso in numerosi Stati, è nata l'esigenza di poter codificare anche simboli che non erano presenti nell'alfabeto americano, come ad esempio “ç”, o le vocali con la dieresi come “Ö”, “Ü” ecc..

Per poter codificare un numero maggiore di caratteri alfanumerici si è deciso di aggiungere un bit alla codifica ASCII, questo ha consentito di raddoppiare il numero di configurazioni disponibili (da 128 configurazioni con 7 bit a 256 configurazioni con 8 bit) e quindi di utilizzare queste ulteriori 128 configurazioni di bit per nuovi caratteri (quelli della seconda tabella vista sopra). **ASCII a 8 bit chiamato ASCII esteso.**

Esempio: grazie all'estensione ad 8 bit del codice ASCII è possibile codificare e quindi memorizzare correttamente nella memoria anche il nome del calciatore Özil (grazie al codice ASCII 153 per la prima lettera. Con la versione di ASCII a 7 bit il codice per tale lettera non esisteva)

In una semplice memoria ad 8 bit il nome di questo calciatore sarà memorizzato nel seguente modo:

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0	
1	0	0	1	1	0	0	1	Ö (153)
0	1	1	1	1	0	1	0	z (122)
0	1	1	0	1	0	0	1	i (125)
0	1	1	0	1	1	0	0	l (108)

← MEMORIA CON CELLE DA 8 BIT

Esercizi:

1. Qual è il testo codificato nel seguente modo con il codice ASCII ad 8 bit?
01000011 01001001 01000001 01001111 **(CIAO)**
2. Codificare in codice ASCII ad 8 bit il proprio cognome. Quanti byte di memoria occupa?

CODIFICA UNICODE

In seguito all'esigenza di rappresentare anche lettere di altri alfabeti ed altri simboli (cinesi, arabi ad esempio) è stato inventato da una società no profit chiamata **Unicode Consortium**, un altro codice chiamato codice **UNICODE**.

Unicode è una cosa leggermente diversa da ASCII. Unicode è uno standard che ha stabilito due cose:

- Ha raccolto tutti i caratteri ed i simboli presenti nel mondo (TUTTI! anche latino, cinese, arabo, emoticon, ecc..., infatti viene aggiornato all'incirca ogni anno con l'aggiunta di nuovi simboli)
- Ha associato ad essi un nome ed un codice numerico (il codice unicode è indicato generalmente in esadecimale con il prefisso U+, ad esempio "a" ha il codice U+0061 e si chiama *Latin Small Letter A*)

Per la conversione in binario di un carattere Unicode ci sono diverse **codifiche** (chiamate **encoding**) Queste codifiche sono delle "regole" che definiscono con quanti bit deve essere memorizzato un codice unicode nella memoria. I diversi encoding si differenziano per il **set di caratteri** codificato, ossia l'insieme di simboli utilizzati (ci sono che encoding "convertono" in binario solo alcuni caratteri, ad esempio l'encoding BIG 5 è utilizzato per alcune lingue orientali ma non per tutte) e per la modalità di codifica (alcuni encoding utilizzano 1 byte per ogni carattere, altri 2 byte ecc...). La differenza con ASCII sta proprio nel fatto che Unicode non specifica la modalità di conversione in bit dei caratteri, Unicode ha semplicemente raccolto i caratteri e gli ha assegnato un nome ed un codice esadecimale, poi qualunque software house può inventarsi un encoding proprietario per codificare in binario i caratteri Unicode. Nonostante chiunque possa inventarsi il proprio encoding, lo standard Unicode ha ideato, per favorire la standardizzazione e quindi l'interscambio delle informazioni, 3 tipologie di encoding, ciascuna delle quali prevede un proprio "modo" per codificare nella memoria i codici UNICODE. Queste tre codifiche sono chiamate UTF-8, UTF-16, UTF-32.

UTF-8 : è un **encoding** a lunghezza variabile, infatti utilizza 1 byte per codificare i primi 128 caratteri (che coincidono con ASCII), ed un numero maggiore di byte (fino a 4, quindi 32 bit) per codificare gli altri simboli, che sono i meno utilizzati. **Ad esempio per le lettere accentate utilizza due byte.** Tale codifica ha il vantaggio di essere compatibile con ASCII (ma non con ASCII esteso) quindi di poter garantire la

portabilità di documenti anche su software obsoleti o di consentire la lettura, con software nuovi, di documenti prodotti con software vecchi. Tale codifica è inoltre molto efficiente poiché utilizza meno memoria per codificare i caratteri più frequentemente utilizzati (i primi 128 di ASCII) rispetto a quelli meno utilizzati.

UTF-16: è un **encoding** a lunghezza variabile. Utilizza 2 byte (16 bit) per i primi 65536 caratteri UNICODE, per i successivi (meno frequenti) utilizza 4 byte (32 bit). Il vantaggio che i caratteri più frequentemente utilizzati occupino sempre due byte consiste nella più facile gestione delle stringhe da parte del software che utilizza questa codifica. UTF-16 è molto diffuso (utilizzato dal SO Windows) poiché infatti rappresenta un buon compromesso fra la facilità di gestione dei caratteri da parte di un software e lo spazio occupato in memoria. Per accedere all'elenco dei caratteri Utilizzati dal SO Windows vi è un'apposita utility chiamata "Mappa caratteri".

UTF-32: e' un **encoding** a lunghezza fissa. Ogni carattere viene infatti codificato utilizzando 4 byte (32 bit). Ha lo svantaggio di occupare più memoria ma consente una gestione delle stringhe più semplice nei software che utilizzano tale codifica.

Riassumendo si può dire che, **limitandoci a considerare i primi 128 caratteri ASCII**, ossia quelli più utilizzati:

UTF-8 : per ogni carattere occupa 1 o 2 byte (per i caratteri da 128 a 255 di ASCII occupa due byte)

UTF-16 : per ogni carattere ASCII occupa 2 byte

UTF-32 : per ogni carattere occupa 4 byte

ESEMPI CON I SOFTWARE

Per far comparire un qualsiasi carattere ASCII/UNICODE in un programma di videoscrittura (**WORD**) è sufficiente tenere premuto il tasto Alt e digitare (con il tastierino numerico) il codice relativo al carattere **convertito in decimale**.

Ad esempio per la A (codice unicode U+0041 → Valore decimale 65) A → alt 65

Ad esempio per la Σ (codice unicode U+01A9 → Valore decimale 425) Σ → alt + 425

Ad esempio per 😊 (codice unicode U+1F608 → Valore decimale 128520) 😊 → alt + 128520

(Sito che a partire dal codice unicode restituisce il nome del carattere: <https://unicode-search.net/>)

Esempi con NOTEPAD:

- Scrivere con word la seguente stringa “ciao Σ”, poi copiare la stringa su Notepad e salvarla con codifica ANSI (che è ASCII). Il file occuperà 6 byte ma riaprendolo si vedrà che il carattere Σ non viene riconosciuto.

Scrivere in word la seguente stringa “ciao 🎵” (🎵 è U+226B Beamed Eighth Note, $266B_{(16)}=9835_{(10)}$ quindi ALT+9835. Devi copiarlo da Word o da mappa caratteri ed incollarlo in Notepad) e salvarlo nelle diverse codifiche. Poiché tale simbolo occupa 3 byte in UTF 8, lo spazio occupato nelle varie codifiche sarà il seguente:

- ANSI: 6 byte (ma il carattere 🎵 non verrà memorizzato)
- UTF 8 : 8 byte (5 per i caratteri *normali* di “ciao “ + 3 per il carattere 🎵) + eventualmente altri 2 byte per la codifica (solo in alcuni casi)
- UTF 16 (ossia Unicode o Big Endian Unicode): 14 byte (2 per la codifica + 2 per ciascun carattere)

Esercizi:

1. Quanti byte occupa la stringa “Buongiorno Piero” codificata con UTF-8? (1x16=16 byte uno per lettera, compreso lo spazio)
2. Quanti byte occupa la stringa “Buongiorno Piero” codificata con UTF-16? (2x16=32 byte)
3. Quanti byte occupa la stringa “Questo è Piero” codificata con ASCII? (1x14 =14 byte)
4. Quanti byte occupa la stringa “Questo è Piero” codificata con UTF-8? (Poiché il carattere “è” occupa due byte e tutti gli altri un byte : $1 \times 13 + 2 \times 1 = 15$ byte)
5. Quanti byte occupa la stringa “Questo è Piero” codificata con UTF-32? (4x14= 56 byte)