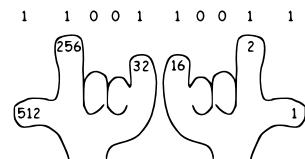




Sistema binario



Introduzione
all'informatica

Michele Tomaiuolo
Ingegneria dell'Informazione, UniPR

Analogico e digitale

- Una grandezza (fisica o astratta) può essere rappresentata in due forme
 - Analogica:** insieme di valori **continuo** (*denso e "senza buchi"*)
 - Digitale** (o numerica): insieme di valori **discreto** (*tutti i punti sono isolati*)



tomamic.github.io/fondinfo

2/33

Approssimazione discreta

- Alcune informazioni sono intrinsecamente discrete
 - Informazioni "artificiali", es. un testo scritto
 - Scala atomica o subatomica ...
- Molte grandezze fisiche hanno forma continua
 - Per loro elaborazione al calcolatore: rappresentazione digitale
 - Approssimazione** del valore analogico
 - Errore dipende dalla precisione della rappresentazione digitale scelta

Codice

- Sistema basato su simboli, che permette la rappresentazione dell'informazione
- Simbolo:** elemento atomico
- Alfabetto:** insieme dei simboli possibili (A)
- Cardinalità** del codice: numero di simboli dell'alfabeto
- Stringa:** sequenza di simboli ($s \in A^*$)
- Linguaggio:** insieme stringhe ben formate ($L \subseteq A^*$)

tomamic.github.io/fondinfo

3/33

tomamic.github.io/fondinfo

4/33

Codice posizionale

- Un numero naturale può essere rappresentato con una notazione posizionale
- $N = c_0 \cdot \text{base}^0 + c_1 \cdot \text{base}^1 + \dots + c_n \cdot \text{base}^n$
 - Es. $587_{10} = 7 \cdot 10^0 + 8 \cdot 10^1 + 5 \cdot 10^2$
- Sistemi di numerazione posizionali di uso comune
 - Decimale (base 10; c: 0-9)
 - Binario (base 2; c: 0-1)
 - Esadecimale (base 16; c: 0-9, A-F)



Codifica dell'informazione

- Codifica: regole di corrispondenza per passare da un certo codice ad un altro
- Corrispondenza biunivoca
 - Tra una stringa di un codice
 - E una stringa di un altro codice
- Ad una certa stringa in un alfabeto ricco di simboli, corrisponde una stringa più lunga in un alfabeto più ridotto



Numeri binari

Codice binario

- Base 2; c: 0-1
- Informazione digitale nei calcolatori rappresentata con una sequenza di 0 e 1
 - Leibniz, ~1700
 - Konrad Zuse, ~1940
- Ogni elemento di una sequenza binaria viene detto **bit**
- Una sequenza di 8 bit viene detta **byte**



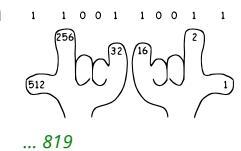
Codifica decimale → binaria

- (1) Dividere il numero decimale per 2
- (2) Assegnare il resto come valore del bit (*loop*)
- Ossia continuare a dividere per 2 il quoziente, finché non si annulla
- Es.: $35_{10} = 00100011_2$

n	n // B	n % B	peso
35	17	1	$1 = 2^0$
17	8	1	$2 = 2^1$
8	4	0	$4 = 2^2$
4	2	0	$8 = 2^3$
2	1	0	$16 = 2^4$
1	0	1	$32 = 2^5$

Numeri naturali

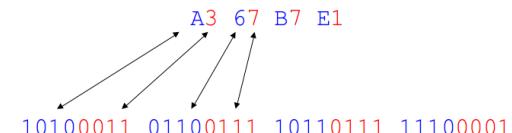
- Rappresentare un numero naturale **N** in forma binaria
- Occorrono **K** bit, t.c. $2^K > N$
- Es. 4 bit per numeri naturali da 0 a 15
- Un calcolatore assegna un numero fisso di bit per diversi tipi di informazione
 - Casi di valori non rappresentabili
 - **Overflow, underflow**



Esadecimale (Hex)

Dec	Bin	Hex	Dec	Bin	Hex	Dec	Bin	Hex
00	0000 0000	00	16	0001 0000	10	32	0010 0000	20
01	0000 0001	01	17	0001 0001	11	33	0010 0001	21
02	0000 0010	02	18	0001 0010	12	34	0010 0010	22
03	0000 0011	03	19	0001 0011	13	35	0010 0011	23
04	0000 0100	04	20	0001 0100	14	36	0010 0100	24
05	0000 0101	05	21	0001 0101	15	37	0010 0101	25
06	0000 0110	06	22	0001 0110	16	38	0010 0110	26
07	0000 0111	07	23	0001 0111	17	39	0010 0111	27
08	0000 1000	08	24	0001 1000	18	40	0010 1000	28
09	0000 1001	09	25	0001 1001	19	41	0010 1001	29
10	0000 1010	0A	26	0001 1010	1A	42	0010 1010	2A
11	0000 1011	0B	27	0001 1011	1B	43	0010 1011	2B
12	0000 1100	0C	28	0001 1100	1C	44	0010 1100	2C
13	0000 1101	0D	29	0001 1101	1D	45	0010 1101	2D
14	0000 1110	0E	30	0001 1110	1E	46	0010 1110	2E
15	0000 1111	0F	31	0001 1111	1F	47	0010 1111	2F

Bin ↔ Hex



- Ogni gruppo di 4 bit: 16 configurazioni diverse ($2^4 = 16$)
- Ciascuna combinazione corrisponde ad uno dei 16 simboli esadecimali

Binary	Hex	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Somma e sottrazione

1 1
0 0 0 1 0 1 0 +
0 0 0 1 0 1 0 1 =

0 0 1 0 1 0 1 1

BINARY

0 10
0 0 0 0 1 1 1 0 -
0 0 0 0 0 1 0 1 =

0 0 0 0 1 0 0 1

BINARY

Attenzione a riporto e prestito (in alto)

Moltiplicazione

1 0 1 1 x
1 1 0 1 =

1 0 1 1 +
0 0 0 0

0 1 0 1 +
1 0 1 1

1 1 0 1 1 1 +
1 0 1 1

1 0 0 0 1 1 1 1

BINARY

tomamic.github.io/fondinfo ☺

13/33

tomamic.github.io/fondinfo ☺

14/33

Divisione

1 0 1 1 0 1 : 1 1
0 0 -----
----- 0 1 1 1 1
1 0 1 -
1 1

1 0 1 -
1 1

1 0 0 -
1 1

1 1 -
1 1

0 0

BINARY

Numeri interi

- Occorre rappresentare anche i numeri negativi
 - Necessario riservare un bit per il segno
 - Ovvero, si dimezza il massimo modulo ammesso
- Modulo e segno**
 - Il primo bit indica il segno
 - 0 positivo, 1 negativo

tomamic.github.io/fondinfo ☺

15/33

tomamic.github.io/fondinfo ☺

16/33

Complemento a due

- Rappresentazione alternativa, *diversa da modulo e segno!*
- Numero negativo, ottenuto dal suo opposto positivo
 - Complemento il numero (cambio gli 1 con 0 e viceversa)
 - Sommo 1
- Anche così, il primo bit indica il segno
 - 0 positivo, 1 negativo
- Attenzione:** bisogna conoscere codifica e num bit
 - Esempi seguenti: ogni intero con segno memorizzato in un singolo byte

Binary	Hex	Decimal	
		US	S
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1

Es. numero intero

- Avendo un byte, +35 è in binario: **00100011**
- Numero -35, in modulo e segno: **10100011**
- Numero -35, in complemento a due: **11011101**

0 0 1 0 0 0 1 1 ~

1 1 0 1 1 1 0 0 +
 1 =

1 1 0 1 1 1 0 1

→: *complemento semplice, bit a bit*

Somma con segno

- Sommare 12 e -35 su 8 bit, modulo e segno
 - Sottrazione tra 35 e 12
 - Cambio di segno
- Stessa operazione, complemento due
 - Semplice somma: **12 + -35 = -23**

0 0 0 0 1 1 0 0 +
1 1 0 1 1 1 0 1 =

1 1 1 0 1 0 0 1

Numeri reali

- Insieme continuo, per grandezze analoghe
 - Rappresentabili solo in modo approssimato
- Parte frazionaria:
 - $F = c_{-1} \cdot base^{-1} + \dots + c_{-n} \cdot base^{-n}$
- Due rappresentazioni *alternative*
 - Virgola fissa:** segno, parte intera, parte decimale
 - Virgola mobile:** segno, mantissa, esponente

Parte frazionaria in binario

- Moltiplicare la parte frazionaria per 2
 - Assegnare la parte intera del risultato come valore del bit (*loop*)
 - Ossia: continuare a moltiplicare per 2 la parte frazionaria del risultato... finché non si annulla

fract	fract*B	int	peso
0,375	0,750	0	2 ⁻¹
0,750	1,500	1	2 ⁻²
0,500	1,000	1	2 ⁻³

 tomamic.github.io/fondinfo 

21/33

Virgola fissa

- Numero espresso come: $r = (i, f)$
 - **i** è la parte intera, n_1 bit
 - **f** è la parte frazionaria, n_2 bit
 - Precisione costante lungo l'asse reale
 - P.es. **f** di 3 bit, valori consecutivi sempre distanziati di 1/8



Virgola mobile

- Numero espresso come: $r = \pm(1+f) \cdot 2^e$
 - **e** è l'esponente intero (o caratteristica), n_1 bit
 - **f** è la parte frazionaria ($0 \leq f < 1$), n_2 bit
 - 2 è la base, $1+f$ è anche detto **mantissa**
 - Precisione variabile lungo l'asse reale; p.es.:
 - $f \in \{0, 1/4, 2/4, 3/4\}$, 2 bit
 - $e \in \{-2, -1, 0, 1\}$, 2 bit



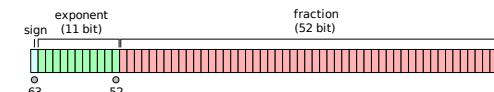
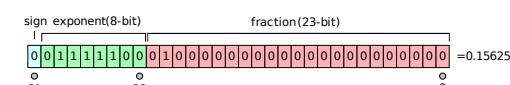
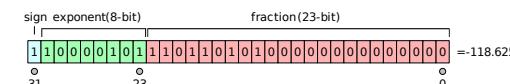
http://www.mathworks.com/company/newsletters/news_notes/pdf/Fall96Cleve.pdf

 tomamic.github.io/fondinfo 

23/33

IEEE 754

- $-118.625 = -1110110.101_2 = -1.110110101_2 \times 2^6$
 - All'esponente, su 8 bit, bisogna sommare 127 ($= 2^8 - 1 - 1$)



 tomamic.github.io/fondinfo 

24/33



Algebra di Boole

- L'algebra di Boole è un formalismo che opera su variabili (dette *variabili booleane*)
- Le variabili booleane possono assumere due soli valori: **vero**, **falso**
- Sulle variabili booleane è possibile definire delle funzioni (dette *funzioni booleane*)
- Anche le funzioni booleane possono assumere solo i due valori **vero** e **falso**

Algebra di Boole

Funzione e tabella di verità

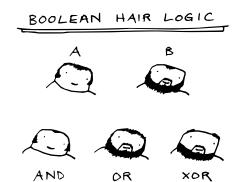
- Una *tabella di verità* permette di definire una *funzione booleana*
- Valore risultante per ciascuna combinazione dei valori in ingresso
- A volte, *specifica incompleta* (certe combinazioni di ingressi non possono verificarsi) → Non è specificato alcun valore

A	B	C	F ₁
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Espressione booleana

- Algebra di Boole: basata su un insieme di operatori
- Possono essere combinati in espressioni
- Altra forma di definizione di funzioni booleane
- Es. $F_2(A, B, C) = A \cdot B + C$

Operatore	Simbolo
And	\cdot (\wedge)
Or	$+$ (\vee)
Not	\neg
Xor	\oplus
Nand	\uparrow
Nor	\downarrow



Operatori principali

A	B	$A \cdot B$	$A + B$	$A \oplus B$	$A \uparrow B$	$A \downarrow B$
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	1	0
1	1	1	1	0	0	0

A	$\neg A$
0	1
1	0

Proprietà degli operatori

Proprietà	Not	
Complemento	$\neg \neg A = A$	
Proprietà	And	Or
Commutativa	$A \cdot B = B \cdot A$	$A + B = B + A$
Associativa	$(A \cdot B) \cdot C = A \cdot (B \cdot C)$	$(A+B)+C = A+(B+C)$
Distributiva	$A + (B \cdot C) = (A+B) \cdot (A+C)$	$A \cdot (B+C) = (A \cdot B) + (A \cdot C)$
Idempotenza	$A \cdot A = A$	$A + A = A$
Identità	$A \cdot 1 = A$	$A + 0 = A$
Del limite	$A \cdot 0 = 0$	$A + 1 = 1$
Assorbimento	$A \cdot (A+B) = A$	$A + (A \cdot B) = A$
Inverso	$A \cdot \neg A = 0$	$A + \neg A = 1$
De Morgan	$\neg(A \cdot B \cdot C \dots) = \neg A + \neg B + \neg C \dots$	$\neg(A+B+C \dots) = \neg A \cdot \neg B \cdot \neg C \dots$

Attenzione a De Morgan: errore comune!

Forme canoniche

- Somma di Prodotti (SP):** si considerano le righe a 1
 - $F_1(A, B, C) = (\neg A \cdot \neg B \cdot \neg C) + (\neg A \cdot B \cdot C) + (A \cdot \neg B \cdot C) + (A \cdot B \cdot \neg C) + (A \cdot B \cdot C)$
- Prodotto di Somme (PS):** si considerano le righe a 0
 - $F_1(A, B, C) = (A + B + \neg C) \cdot (A + \neg B + C) \cdot (\neg A + B + C)$

A	B	C	F_1	→ Forma canonica...
0	0	0	1	→ SP
0	0	1	0	→ PS
0	1	0	0	→ PS
0	1	1	1	→ SP
1	0	0	0	→ PS
1	0	1	1	→ SP
1	1	0	1	→ SP
1	1	1	1	→ SP

Operazioni bit a bit in Python

```

x, y, z, shift = 0, 0, 0, 0 # some int values
x << shift # x = x * (2^shift)
x >> shift # x = x / (2^shift), con segno
x & y      # AND applicato bit a bit
x | y      # OR applicato bit a bit
x ^ y      # XOR bit a bit
~x         # complemento di ogni bit

z = 0x0B    # hex value (11 dec)
z = 0b1011 # bin value (11 dec)

hex(11)    # '0x0b' (text)
bin(0x0B)  # '0b1011' (text)

```

PYTHON

Da non confondere con operatori logici (and, or, not)

Dati multimediali



<Domande?>

Michele Tomaiuolo
Palazzina 1, int. 5708
Ingegneria dell'Informazione, UniPR

Introduzione
all'informatica

Michele Tomaiuolo
Ingegneria dell'Informazione, UniPR

Caratteri e testo

- Necessaria convenzione per codifica numerica (binaria) dei caratteri
- Codifica **ASCII** (American Standard Code for Information Interchange) a 7 bit
 - *Caratteri alfanumerici*: lettere maiuscole, minuscole, numeri, spazio
 - *Simboli e punteggiatura*: @, #, ...
 - *Caratteri di controllo* (non tutti visualizzabili): TAB, LF, CR, BELL ecc.

Caratteri e testo

Tabella ASCII di base

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	(EOT)	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	(DEVICE 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	(END 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	(TAB SEPARATOR)	60	3C	<	92	5C	\	124	7C	\
29	1D	(GROUP SEPARATOR)	61	3D	=	93	5D	J	125	7D	J
30	1E	(RECORD SEPARATOR)	62	3E	>	94	5E	^	126	7E	=
31	1F	(UNIT SEPARATOR)	63	3F	?	95	5F	-	127	7F	{DEL}

Interruzione di riga

- Unix: LF
 - Multics, Unix etc., Mac OS X, BeOS, Amiga, RISC OS
- Windows: CR+LF
 - Most early OSes, DOS, OS/2, Windows, Symbian
- Vecchi Apple: CR
 - Commodore machines, Apple II family, Mac OS up to version 9

Tabella ASCII estesa

- Caratteri accentati + caratteri per grafici
 - **Code Page 437** per PC (DOS) in Nord America
 - Possibile mischiare testo in inglese e francese (anche se in Francia **CP850**); ma non assieme greco (**CP737**), russo ecc.
- ISO 8859**, estensioni standard per ASCII ad 8 bit
 - ISO 8859-1 (o Latin1): Lingue dell'Europa Occidentale
 - ISO 8859-2: Lingue dell'Europa Orientale
 - ISO 8859-5: Alfabeto cirillico
 - ISO 8859-15: Latin1 con simbolo euro (€)



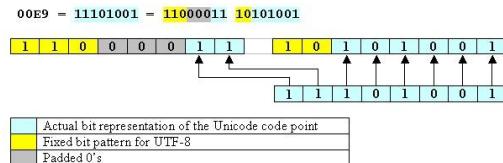
Unicode

- Unicode associa un preciso **code-point** (32 bit) a ciascun simbolo
 - Possibile rappresentare miliardi di simboli
 - Primi 256 code-point = **Latin1**
- Attualmente >30 sistemi di scrittura
 - Rappresentazione di **geroglifici** e caratteri **cuneiformi**
 - Proposta per **Klingon** (da Star Trek... rifiutata!)

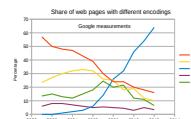


Unicode Transformation Format

- Codifica di un **code-point** in una sequenza di bit (uno o più **code-unit**)
- UTF-32** – code-unit di 32-bit, lunghezza fissa
- UTF-16** – code-unit di 16-bit, lunghezza variabile
- UTF-8** – code-unit di 8-bit, ma lunghezza variabile (1-4 byte), max compatibilità con ASCII



String	abcd学乇
Char Count	7
Character Count	6
UTF-8 Byte Count	11

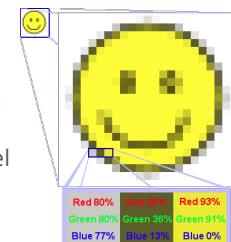


UTF-8

- Se bit più alto a 0:
 - Simbolo ASCII su 7 bit
- Altrimenti:
 - Numero 1 in byte iniziale = numero di byte per **code-point**
 - Byte seguenti cominciano tutti con **10**

Character	Binary code point	Binary UTF-8	Hexadecimal UTF-8
\$	U+0024	010 0100 00100100	24
€	U+00A2	000 1010 0010 11000010 10100010	C2 A2
€	U+20AC	0010 0000 1010 1100 11000010 10000010 10101100	E2 82 AC
ø	U+10348	0 0001 0000 0011 0100 1000 1101 00011001 10001000	F0 90 8D 88

Immagini

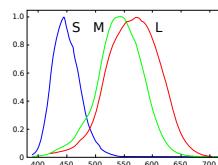


Immagini raster

- Digitalizzazione**: da immagine a sequenza binaria
- Immagine raster** suddivisa in una griglia di punti (**pixel**)
 - Ogni pixel descritto da un codice, che ne individua il colore
- Profondità**: # bit per rappresentare il colore di un pixel
 - 1, 2, 8, 12, 16, 24, 32... bit per pixel (**bpp**)
 - Es. 8 bit per 256 (2^8) possibili colori
 - Colore diretto o indicizzato da una **palette**
- Risoluzione**: # punti per pollice (**dpi**), come in tipografia
 - Spesso (ma non sempre), risoluzione orizzontale uguale a verticale

Modelli di colore

- Occhio sensibile a variaz. luminosità
 - 6 mln di coni, 125 di bastoncelli
- **RGB**: rosso, verde, blu
 - 8 bit: 3 bit × R e G, 2 × B
 - 24 bit: 8 bit × R, G e B
 - 32 bit: canale alpha × opacità
- **YUV**: luminosità, crominanza di R e B
 - Sistema PAL, MPEG
 - TV a colori (compat. B&W)
- **HSB**: tonalità, saturazione e luminosità



Formati di file grafici

- **BMP**: immagine (normalmente) non compressa
- **TIFF, PNG**: comprimono l'immagine, per ridurne l'occupazione, senza deteriorarla (compressione *lossless*)
- **JPEG**: comprime (molto di più), ma deteriora l'immagine (compressione *lossy*)

Formato BMP

FILE INFO HEADER (14)
2 **Tipo** file (= "BM")
4 **Dim.** file (**in byte**)
4 **Riservato**
4 **Offset immagine** (**in byte**)
BITMAP INFO HEADER (40)
4 **Dimensione** struttura
4+4 **Larghezza** e altezza immagine
2 **Piani** (non usato)
2 **# bit per pixel**
4+4 **Compressione** e dim. img (0 senza compressione)
4+4 **Risoluzione** orizz. e vert. (pixel per metro)
4+4 **# colori in palette e # colori importanti**
Pallete (RGBQUAD)
4 **Blue, Green, Red, Riservato**



Es. Redbrick.BMP

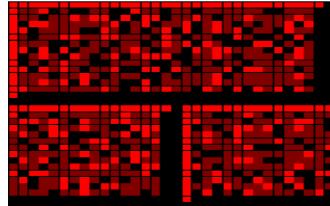
	BM	File size	W×H=32×32	Inizio img	Profondità (bpp)	40
0000	42	4d	76	02	00	00
0010	00	00	20	00	00	01
0020	00	00	00	00	00	04
0030	00	00	00	00	00	00
0040	00	00	00	00	00	00
0050	00	00	80	80	00	00
0060	00	00	ff	ff	ff	00
0070	00	00	ff	ff	ff	00
0080	00	00	00	00	00	00
0090	00	00	00	00	00	00
00a0	01	09	11	01	90	11
00b0	09	11	19	10	11	19
00c0	91	10	91	09	10	09
00d0	91	01	19	00	99	11
00e0	01	11	11	91	10	09
00f0	11	99	10	01	11	11
0100	01	11	11	19	10	09

Es. Redbrick.BMP

Palette-index

```
row 0, scanline 31 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
row 1, scanline 30 00 00 00 00 00 00 00 00 09 00 00 00 00 00 00 00 00 00  
row 2, scanline 29 11 11 01 19 11 01 10 10 09 09 01 09 11 11 01 90  
.  
. .  
row 31, scanline 0 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 90
```

Pixel rectangle

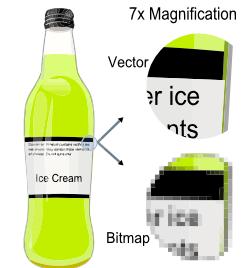


Color-palette

0 Black
1 Dark red
2 Dark green
3 Dark yellow
4 Dark blue
5 Dark magenta
6 Dark cyan
7 Dark gray
8 Light gray
9 Light red
A Light green
B Light yellow
C Light blue
D Light magenta
E Light cyan
F White

Grafica vettoriale

- Immagine: insieme di primitive geometriche
 - Linee, poligoni..., colori, sfumature...
- ▲ Qualità, a varie risoluzioni
- ▲ Compressione dati
- ▲ Gestione modifiche
- ▼ Non intuitiva per alcuni
- ▼ Possibilmente onerosa
- ▼ Risorse non note a priori

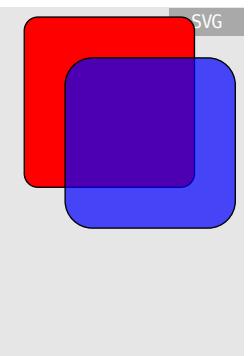


Grafica vettoriale

- **Applicazioni:** editoria (DTP), video-editing, architettura, ingegneria, grafica 3D (CAD), font vettoriali (caratteri scalabili in dimensione senza perdere definizione)
- **Formati esistenti:** PS (PostScript), PDF (Portable Document Format), WMF (Windows MetaFile), DXF (AutoCAD), CDR (CorelDraw), SWF (Flash), SVG (Scalable Vector Graphics, per il web)

Esempio di file SVG

```
<!-- possibly inside an HTML5 file -->  
  
<svg width="800" height="600">  
  
  <rect x="80" y="60" width="250" height="250"  
        rx="20" fill="#ff0000"  
        stroke="#000000" stroke-width="2" />  
  
  <rect x="140" y="120" width="250" height="250"  
        rx="40" fill="#0000ff"  
        stroke="#000000" stroke-width="2"  
        fill-opacity="0.7" />  
  
</svg>
```

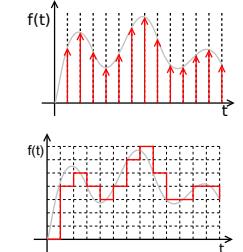


Audio digitale



DII

- Grandezza analogica → discretizzazione
 - **Campionamento** (*sampling*) nel tempo
 - **Quantizzazione** (*quantizing*) nelle ampiezze
 - Qualità CD: 44 kHz, 16bit
 - Spettro udibile: 20-20k Hz, Nyquist-Shannon



Audio digitale

Formato WAV

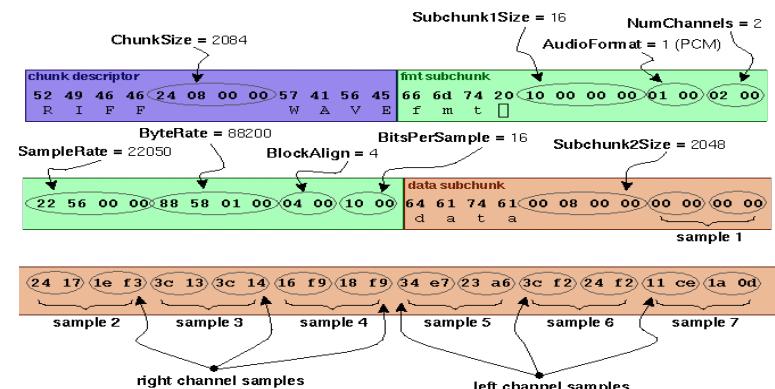
The Canonical WAVE file format			
Endian	File offset (bytes)	Field name	Field Size (bytes)
big	0	ChunkID	4
little	8	ChunkSize	4
big	12	Format	4
big	16	Subchunk1ID	4
little	20	Subchunk1Size	4
little	22	AudioFormat	2
little	24	NumChannels	2
little	28	SampleRate	4
little	32	ByteRate	4
little	34	BlockAlign	2
little	36	BitsPerSample	2
big	40	Subchunk2ID	4
little	44	Subchunk2Size	4
		data	Subchunk2Size

The "RIFF" chunk descriptor
The Format of concern here is "WAVE", which requires two sub-chunks: "fmt" and "data".

The "fmt" sub-chunk
describes the format of the sound information in the data sub-chunk.

The "data" sub-chunk
Indicates the size of the sound information and contains the raw sound data

Esempio di file WAV





Documenti strutturati

Documenti strutturati

- **Struttura logica**

- Determina il *ruolo* delle varie parti del testo
- Titoli, testo, note, etc.

- **Struttura grafica**

- Assegna una resa grafica ai ruoli
- Quindi determina la resa grafica del documento nel suo complesso
- "Stampa" in modo diverso ciò che ha ruolo diverso

- **Word processing:** non tanto *scrivere*, ma *ingegnerizzare informazione*



25/32



24/32

WYSIWYG

- Focus su grafica, si perde di vista la struttura logica
 - Grafica: non con i comandi grafici...
 - Ma definendo gli **stili** delle varie parti di doc, come *ruoli* logici
 - Es. stili di Word/Writer: "*Titolo*", "*Nota in Calce*", "*Intestazione*"
 - Non nomi grafici, ma logici
- In alternativa: editing basato su **comandi** o su **tag**

HyperText Markup Language

- Documenti **strutturati**, standard W3C: <http://www.w3.org/html/>
- HTML dichiara tipi di elementi
 - Paragrafi, titoli, liste, collegamenti ipertestuali, elementi multimediali ecc.
- Tipo di **elemento** descritto da tre parti
 - **Tag di apertura, contenuto, tag di chiusura**
 - Bla bla, `in grassetto.`, normale.
- Molti tag permettono la definizione di **attributi**
 - `UniPR`
 - **id** e **class**: attributi generici per assegnare *ruoli logici*
- Tag semplici non hanno un contenuto
 - ``

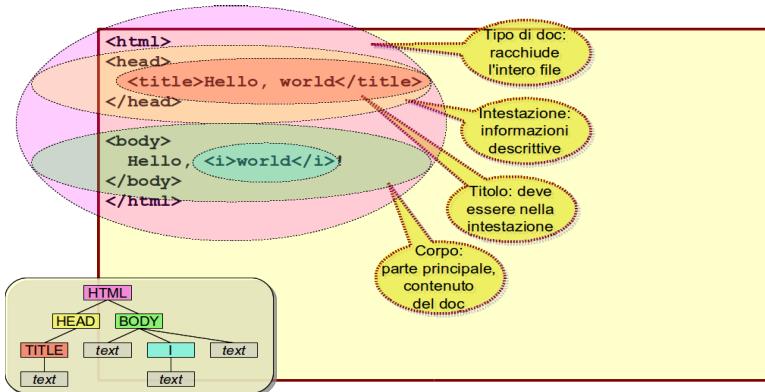


27/32



26/32

Anatomia di una pagina



Tag di formattazione testo

```
<p>Questo è un paragrafo.<br />A-capo ma stesso paragrafo.</p>

<p>Testo <strong>in grassetto</strong>, e poi
<em>in corsivo</em>.</p>

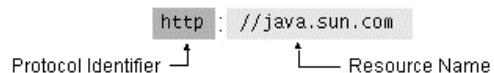
<h1>Il titolo più grande</h1>
...
<h6>Il titolo più piccolo</h6>

<div class="remark">
  Struttura generica di livello blocco,
  con un <span>elemento generico</span> inline.
</div>
```

HTML

Uniform Resource Locator

- Una URL è un riferimento per una risorsa



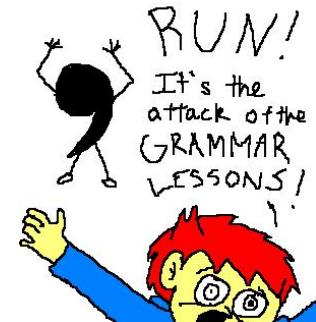
- Il nome della risorsa dipende interamente dal protocollo. Per HTTP include:
 - Nome dell'host su cui risiede la risorsa
 - Numero di porta cui collegarsi (default = 80)
 - Percorso della risorsa sulla macchina
 - Stringa di query (dopo ?)
 - Frammento: id di un elemento all'interno della risorsa (dopo #)
- <http://www.ietf.org:80/rfc/rfc2732.txt>

Html 5

- Nuovi elementi di struttura di Html 5
 - header, main, nav, aside, footer
 - article, section, details, summary
 - figure, figcaption
- Altri nuovi elementi
 - video, audio, canvas, embed
 - mark, menu, command, output, time
 - progress, meter, datalist
- <http://dev.w3.org/html5/html4-differences/>



Linguaggi formali



<Domande?>

Michele Tomaiuolo
Palazzina 1, int. 5708
Ingegneria dell'Informazione, UniPR



Introduzione
all'informatica

Michele Tomaiuolo
Ingegneria dell'Informazione, UniPR

Linguaggi formali

- Presenti in tutte le applicazioni
 - Linguaggi di programmazione
 - Linguaggi di marcatura (es. HTML, Latex)
 - Interazione uomo macchina (es. Google, Zork)
- Fondamentali nel software di sistema
 - Compilatori
 - Interpreti ...
- Paradigmatici nella teoria
 - Molti problemi riconducibili a quello dell'**appartenenza**: una stringa appartiene ad un linguaggio?

Alfabeti e stringhe

- Alfabeto Σ : insieme di simboli
- Stringa s : sequenza di simboli di Σ
 - $s \in \Sigma^*$, insieme di tutte le stringhe
 - ϵ : stringa vuota
 - $|s|$: lunghezza della stringa s
- Linguaggio $L \subseteq \Sigma^*$
 - Sottoinsieme di tutte le stringhe possibili
 - Grammatica: regole formali per definire le "**stringhe ben formate**" di L
- Esempio: numeri romani da 1 a 1000
 - Alfabeto {I, V, X, L, C, D, M} + regole...

Concatenazione di stringhe

- Operazione di concatenazione •
 - Propr. associativa: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$
 - Non commutativa: $x \cdot y \neq y \cdot x$
 - Σ^* chiuso rispetto a \cdot : $\Sigma^* \times \Sigma^* \rightarrow \Sigma^*$
- Potenza
 - $x^n = x \cdot x \cdot x \cdot x \dots$ (n volte)
- Elemento neutro ϵ
 - Stringa vuota, $\forall x \in \Sigma^*, \epsilon \cdot x = x \cdot \epsilon = x$

$\langle \Sigma^*, \cdot, \epsilon \rangle$: monoid

Definizione di linguaggi

- Approccio **algebrico**: linguaggio costruito a partire da linguaggi più elementari, con operazioni su linguaggi
- Approccio **generativo**: grammatica, regole per la generazione di stringhe appartenenti al linguaggio
- Approccio **riconoscitivo**: macchina astratta o algoritmo di riconoscimento, per decidere se una stringa appartiene o no al linguaggio

Espressioni regolari

Operazioni su linguaggi

- L_1 ed L_2 linguaggi su Σ^* (due insiemi di stringhe)
- Unione: $L_1 \cup L_2 = \{x \in \Sigma^* : x \in L_1 \vee x \in L_2\}$
- Intersezione: $L_1 \cap L_2 = \{x \in \Sigma^* : x \in L_1 \wedge x \in L_2\}$
- Complementazione: $\overline{L}_1 = \{x \in \Sigma^* : x \notin L_1\}$
- Concatenazione o prodotto: $L_1 \cdot L_2 = \{x \in \Sigma^* : x = x_1 \cdot x_2, x_1 \in L_1, x_2 \in L_2\}$
- Potenza: $L^n = L \cdot L^{n-1}, n \geq 1; L^0 = \{\epsilon\}$ per convenzione
 - Concatenazione di n stringhe qualsiasi di L
- Stella di Kleene: $L^* = \cup L^n, n = 0 \dots \infty$
 - Concatenazione arbitraria di stringhe di L

L^* : chiusura riflessiva e transitiva di L rispetto a \cdot

Espressioni regolari

- Dato un alfabeto Σ , chiamiamo **espressione regolare** una stringa r sull'alfabeto $\Sigma \cup \{+, *, (,), \cdot, \emptyset\}$ t.c.:
 - $r = \emptyset$: linguaggio vuoto; oppure
 - $r \in \Sigma$: linguaggio con un solo simbolo; oppure
 - $r = s + t$: unione dei linguaggi $L(s), L(t)$; oppure
 - $r = s \cdot t$: concatenazione dei linguaggi $L(s), L(t)$; oppure
 - $r = s^*$: chiusura del linguaggio $L(s)$
 - (con s e t espressioni regolari; simbolo \cdot spesso implicito)
- **Linguaggi regolari**: rappresentabili con espressioni regolari ("regex")

Regex nelle applicazioni

- Concatenazione di caratteri: **goal**
- Unione tra espressioni (opzione): **one|two|three**
- Un car. da un insieme (o no): **[a-z], [^a-zA-Z]**
- Un carattere qualsiasi: **defin.tely**
- Ripetizioni (0+, 1+, 0-1): **go+al, go+al, goo?al**
- Sottoespressione: **(left right)*halt**

```
>>> text = 'Though not quickly, he run the 5th lap steadily.'  
>>> re.findall(r'[a-z]+ly', text)  
['quickly', 'steadily']  
>>> re.sub(r'([0-9])([a-z]+)', r'\1<sup>\2</sup>', text)  
Though not quickly, he run the 5<sup>th</sup> lap steadily.
```

PYTHON

<http://docs.python.org/3/library/re.html> - <http://www.zytrax.com/tech/web/regex.htm>

Grammatiche di Chomsky



Grammatiche di Chomsky

- Grammatica $G = < V_T, V_N, P, S >$
 - V_T : alfabeto finito di simboli **terminali**
 - V_N : ... **non terminali** (variabili, categorie sintattiche)
 - $V = V_T \cup V_N$
 - P : insieme di **produzioni**, relaz. binarie $V^* \rightarrow V_N \rightarrow V^* \times V^*$
 $\langle \alpha, \beta \rangle \in P$ si indica con $\alpha \rightarrow \beta$
 - $S \in V_N$: **assioma**
- $L(G)$: insieme delle stringhe di terminali ottenibili con finite operazioni di riscrittura
 - Applicazione delle regole di produzione, in vario modo

Linguaggio generato da G

- **Derivazione diretta** \Rightarrow : riscrittura di una stringa tramite applicazione di una regola di produzione
- **Derivazione \Rightarrow^*** : chiusura riflessiva e transitiva della derivazione diretta
- **Forma di frase**: stringa x t.c. $x \in V^*$, $S \Rightarrow^* x$
- **Linguaggio generato** da G: forme di frase con soli simboli terminali
 - $L(G) = \{x : x \in V^*, S \Rightarrow^* x\}$
- **Equivalenza** tra G_1 e G_2 : $L(G_1) = L(G_2)$

Grammatiche equivalenti

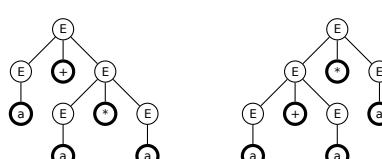
- $G_1 = <\{a, b\}, \{S, A\}, P, S>$, con produzioni:
 - $S \rightarrow b$
 - $S \rightarrow aA$
 - $A \rightarrow aS$
- ... genera il linguaggio $\{a^n b : n \text{ pari}\}$
- Anche G_2 , con produzioni:
 - $S \rightarrow Ab \mid b$
 - $A \rightarrow aAa \mid aa$
- Ed anche G_3 :
 - $S \rightarrow Ab$
 - $A \rightarrow Aaa \mid \epsilon$

| per raggruppare diverse produzioni di uno stesso non-terminale

Esempio di generazione

- $G = <\{a, b, c\}, \{S, B, C\}, P, S>$
 - (1) $S \rightarrow aSBC$
 - (2) $S \rightarrow aBC$
 - (3) $CB \rightarrow BC$
 - (4) $aB \rightarrow ab$
 - (5) $bB \rightarrow bb$
 - (6) $bC \rightarrow bc$
 - (7) $cC \rightarrow cc$
- ... genera il linguaggio $\{a^n b^n c^n : n \geq 1\}$
- Esercizio: provare a generare **aaabbbccc**
 - Soluzione: applicare 1-1-2-3-3-4-5-5-6-7-7

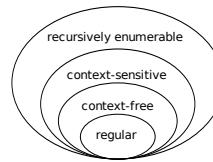
Alberi di derivazione (sintattici)

- Esempio di grammatica **ambigua**: due interpretazioni valide per $a + a * a$
 - $V_T = \{a, +, *, (,)\}; V_N = \{E\}$;
 - $E \rightarrow E+E \mid E^*E \mid (E) \mid a$
- 
- Grammatica non ambigua (con precedenza tra operatori)
 - $E \rightarrow E+T \mid T$
 - $T \rightarrow T*F \mid F$
 - $F \rightarrow (E) \mid a$

Classificazione di Chomsky

- **Tipo 0:** grammatiche **ricorsivam. enumerabili** (RE)
 - $\alpha A\beta \rightarrow \gamma$ (*non limitate*)
- **Tipo 1:** grammatiche **contestuali** (CS)
 - $\alpha A\beta \rightarrow \alpha\gamma\beta$
- **Tipo 2:** grammatiche **non contestuali** (CF)
 - $A \rightarrow \gamma$
- **Tipo 3:** grammatiche **regolari** (REG)
 - $A \rightarrow aB$ oppure $A \rightarrow b$
 - Coincide con classe dei linguaggi definiti da *regex*

$A, B \in V_N; a, b \in V_T; \alpha, \beta, \gamma \in V^*$



DII

Linguaggi non contestuali

Linguaggi non contestuali

- Controllo di *palindromi*, *bilanciamento di parentesi* e varie *simmetrie*
 - Es.: $\{a^n b^n : n \geq 1\}$ gen. da $S \rightarrow aSb \mid ab$ (CF)
 - Ma non: $\{a^n b^n c^n : n \geq 1\}$ (CS) (*)
- **Linguaggi di programmazione** comuni: grammatiche CF
- Definizione con notazione Extended **Backus-Naur Form** (EBNF)
 - {...}: parte ripetibile (0+), [...]: parte opzionale,
 - (...): raggruppamento, |: scelta
 - Terminali tra virgolette

(*) Nell'es. visto, sostituire (3) con: (3a) $CB \rightarrow HB$; (3b) $HB \rightarrow HC$; (3c) $HC \rightarrow BC$

Linguaggi LL(1)

- Sottoclasse dei linguaggi CF
- Ogni produzione relativa a stesso non-terminale (a sx)... genera come primo simbolo un terminale diverso
 - No prefissi comuni, no ricorsione sinistra
- **Recursive descent parser:** analisi sintattica molto semplice ed efficiente
 - Basta "spiare" il simbolo di input successivo, per capire con certezza quale produzione applicare
- **Polish prefix notation**
 - Es.: * + 1 2 - 3 2 \Rightarrow (in forma infissa) (1 + 2) * (3 - 2)

```
expr = number | "+" expr expr | "-" expr expr | "*" expr expr | "/" expr expr  
number = digit {digit}  
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

EBNF

Espressioni infisse

- Grammatica equiv. LL(1), ma senza precedenza tra operatori
- Provare a generare $2 + 3 * 3$

```
expr = term {("+" | "-" | "*" | "/" ) term}
term = number | "(" expr ")" | "-" term
```

EBNF

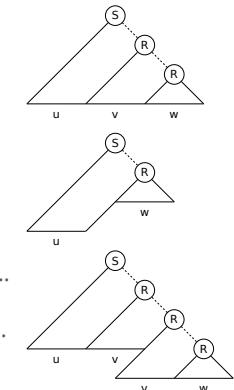
- Grammatica equiv. LL(1), con precedenza tra operatori

```
expr = term {("+" | "-" ) term}
term = factor {("*" | "/" ) factor}
factor = number | "(" expr ")" | "-" term
```

EBNF

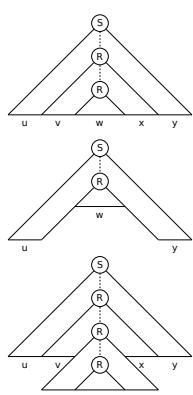
Pumping lemma REG

- Formalmente, $\forall L$ regolare...
 - $\exists k$ t.c. $\forall z \in L, |z| \geq k$
 - $\exists u, v, w, |vwx| \leq k, |vx| \geq 1$ t.c.
 - $z = uvw, uv^i w \in L, \forall i \geq 0$
- In ogni stringa abbastanza lunga,
 - c'è una parte che si può ripetere,
 - generando un'altra stringa di L
- Uno stesso non-terminale, per un input abbastanza lungo, deve comparire più volte nell'albero sintattico ...
 - Un Automa a Stati Finiti (*), per un input abbastanza lungo, torna in uno stato già visitato ...



Pumping lemma CF

- Formalmente, $\forall L$ non contestuale...
 - $\exists k$ t.c. $\forall z \in L, |z| \geq k$
 - $\exists u, v, w, x, y, |vwx| \leq k, |vx| \geq 1$ t.c.
 - $z = uvwxy, uv^i wx^i y \in L, \forall i \geq 0$
- In ogni stringa abbastanza lunga,
 - ci sono due parti che si possono
 - ripetere assieme, restando in L
- Uno stesso non-terminale, per un input abbastanza lungo, deve comparire più volte nell'albero sintattico ...



Corollari dei due pumping lemma

- $\Rightarrow L = \{a^n b^n : n \geq 0\}$ non è REG
 - Si prende $a^m b^m$, con $m > k \Rightarrow |uv| < m$, sono tutte a...
- $\Rightarrow L = \{a^n b^n c^n : n \geq 0\}$ non è CF
 - Si prende $a^m b^m c^m$, con $m > k \Rightarrow |vwx| < m$
 - Se v ed x con più simboli diversi, uv^2wx^2y con simboli mescolati
 - Se v ed x con un solo simbolo, uv^2wx^2y con numero diverso di a, b, c
 - In entrambi i casi la nuova stringa $z' \notin L$



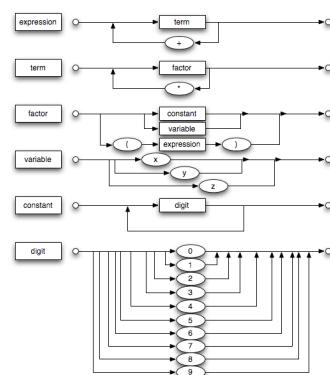
Linguaggi di programmazione

- Notazione formale per definire algoritmi
 - Algoritmo**: sequenza di istruzioni per risolvere un dato problema in un tempo finito
- Ogni linguaggio è caratterizzato da:
 - Sintassi**
 - Semantica**

Linguaggi di programmazione

Sintassi

- Insieme di regole formali per scrivere **frasi ben formate** (programmi) in un certo linguaggio
 - Lessico**: parole riservate, operatori, variabili, costanti ecc. (**token**)
- Grammatiche non contestuali (...) espresse con notazioni formali:
 - Backus-Naur Form
 - Extended BNF
 - Diagrammi sintattici



Semantica

- Attribuisce un **significato** alle frasi (sintatticamente corrette) costruite nel linguaggio
- Una frase può essere sintatticamente corretta e tuttavia non aver alcun significato
 - Soggetto – predicato – complemento
 - "La mela mangia il bambino"
 - "Il bambino mangia la mela"
- Oppure avere un significato diverso da quello previsto...
 - GREEK_PI = 345**

Semantica

• Correttezza sui tipi

- Quali tipi di dato possono essere elaborati?
- Quali operatori applicabili ad ogni dato?
- Quali regole per definire nuovi tipi e operatori?

• Semantica operazionale

- Qual è l'effetto di ogni azione elementare?
- Qual è l'effetto dell'aggregazione delle azioni?
- Cioè, qual è l'effetto dell'esecuzione di un certo programma?

Linguaggi di basso livello

- Più orientati alla macchina che ai problemi da trattare

• Linguaggi macchina

solo operazioni eseguibili direttamente dall'elaboratore

- Op. molto elementari, diverse per ogni processore, in formato binario

• Linguaggi assembly

prima evoluzione, codici binari → mnemonici

```
; Example of IBM PC assembly language
; Accepts a number in register AX;
; subtracts 32 if it is in the range 97-122;
; otherwise leaves it unchanged.

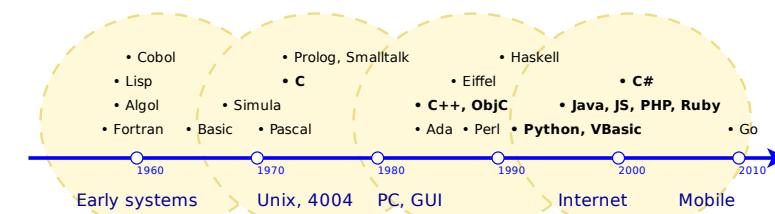
SUB32 PROC           ; procedure begins here
    CMP AX,97          ; compare AX to 97
    JGE DONE            ; if greater, jump to DONE
    CMP AX,122          ; compare AX to 122
    JG DONE             ; if greater, jump to DONE
    SUB AX,32            ; subtract 32 from AX
    RET                 ; return to main program
SUB32 ENDP            ; procedure ends here
```

FIGURE 17. Assembly language

Linguaggi di alto livello

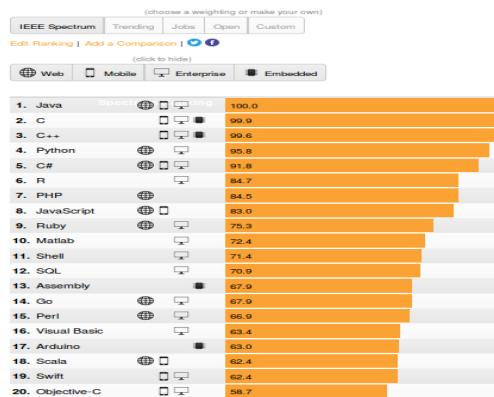
- Introdotti per facilitare la scrittura dei programmi
- Definizione della soluzione in modo intuitivo
- Con una certa **astrazione** rispetto al calcolatore su cui verranno eseguiti
- Devono essere tradotti in linguaggio macchina

Storia dei linguaggi



http://www.oreilly.com/news/graphics/prog_lang_poster.pdf
<http://www.levenez.com/lang/history.html>
http://www.cs.brown.edu/~adff/programming_languages.html

The Top 20 (IEEE Spectrum, 2015)



<http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2015>

tomamic.github.io/fondinfo

32/42

Paradigmi di sviluppo

- Forniscono la filosofia e la metodologia con cui si scrivono i programmi
- Definiscono il concetto (astratto) di computazione
- Ogni linguaggio consente (o sospinge verso) l'adozione di un particolare paradigma
 - Imperativo / procedurale
 - Orientato agli oggetti
 - Scripting (tipizzazione dinamica, principio DRY - Don't Repeat Yourself)
 - Funzionale (funzioni come "cittadini di prima classe")
 - Logico (base di conoscenza + regole di inferenza)

tomamic.github.io/fondinfo

33/42

Linguaggi e paradigmi

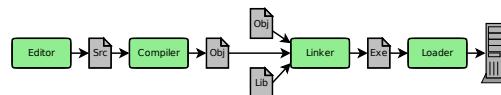
- **Imperativi / procedurali**
 - Cobol, Fortran, Algol, C, Pascal
- **Orientati agli oggetti**
 - Simula, Smalltalk, Eiffel, C++, Delphi, Java, C#, VB.NET
- **Scripting**
 - Basic, Perl, PHP, Javascript, Python
- **Funzionali**
 - Lisp, Scheme, ML, Haskell, Erlang
- **Logici**
 - Prolog...

tomamic.github.io/fondinfo

34/42

Esecuzione dei programmi

- Linguaggio ad alto livello → passi necessari:
 - **Compilazione**, traduzione in linguaggio macchina
 - **Collegamento** con librerie di supporto
 - **Caricamento** in memoria
- Programmi **compilati**: applicati i 3 passi...
 - A tutto il codice; prima dell'esecuzione
- Programmi **interpretati**: applicati i 3 passi...
 - In sequenza, su ogni istruzione; a tempo di esecuzione

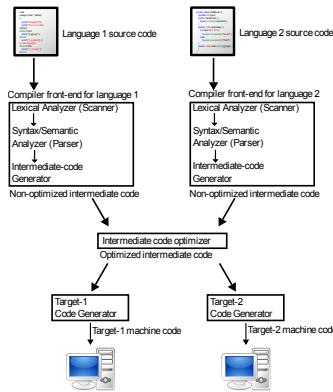


tomamic.github.io/fondinfo

35/42

Compilazione

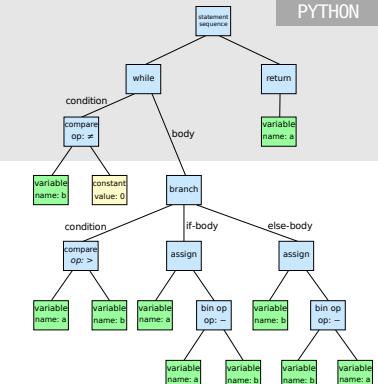
- Traduzione da ling. alto livello a ling. macchina
 - Analisi: lessicale, grammaticale, contestuale
 - **Rappresentazione intermedia**: albero sintattico annotato (**AST**)
 - Generazione codice oggetto
- Codice oggetto: non ancora eseguibile
 - Linker, loader



Albero sintattico

```
while b != 0:  
    if a > b:  
        a = a - b  
    else:  
        b = b - a  
return a
```

Algoritmo di Euclide per MCD



Collegamento

- Il **linker** collega diversi moduli oggetto
 - Simboli irrisolti → riferimenti esterni
 - Il collegamento può essere statico o dinamico
- **Collegamento statico**
 - Libreria inclusa nel file oggetto, eseguibile stand-alone
 - Dimensioni maggiori, ma possibile includere solo funzionalità utilizzate
- **Collegamento dinamico**
 - Librerie condivise da diverse applicazioni
 - Installazione ed aggiornamento unici
 - Caricate in memoria una sola volta

Caricamento

- Il **loader** carica in memoria un programma rilocabile
 - Risolti tutti gli indirizzi relativi (variabili, salti ecc.)
 - Caricati eventuali programmi di supporto
- **Rilocazione statica**: indirizzi logici trasformati in indirizzi assoluti
- **Rilocazione dinamica**: indirizzi logici mantenuti nel programma in esecuzione
 - Programma compilato: indirizzamento relativo
 - Tramite **registro base**: locazione in memoria del codice, dei dati e dello stack (reg. CS, DS e SS su x86)
 - **Memory Management Unit** in S.O.

Codice gestito

- Compilazione in **codice intermedio**
 - Bytecode (Java), Common Intermediate Lang. (.NET), ...
 - Python: compilato per una macchina virtuale (file .pyc), ma in modo trasparente
- Esecuzione su una **macchina virtuale**, che gestisce la memoria (garbage collection)
 - Java Virtual Machine, Common Language Runtime, ...
 - Spesso compilazione "al volo" (*Just In Time*) in codice nativo
- **Garbage collection**
 - Restituzione automatica della memoria
 - Per oggetti/dati che non servono più
 - Possibile anche per codice nativo: linguaggio *Go*, "*smart pointers*" in C++, estensioni...

 tomamic.github.io/fondinfo 

40/42

Garbage collection

- Vantaggi
 - Non è possibile dimenticare di liberare la memoria (*memory leak*)
 - Non è possibile liberare della memoria che dovrà essere utilizzata in seguito (*dangling pointer*)
- Svantaggi
 - Decide autonomamente quando liberare la memoria
 - Liberare e compattare mem. richiede del calcolo
- Diversi algoritmi
 - *Reference counting*: idea di base, ma cicli...
 - *Mark & sweep*: parte da riferimenti locali/globali, marca oggetti raggiungibili
 - *Generational garbage collection*: controlla spesso oggetti recenti



 tomamic.github.io/fondinfo 

41/42

<Domande?>

Michele Tomaiuolo
Palazzina 1, int. 5708
Ingegneria dell'Informazione, UniPR



Introduzione
all'informatica

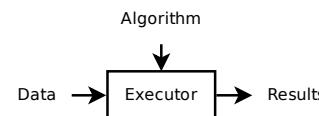
Michele Tomaiuolo
Ingegneria dell'Informazione, UniPR

Automi e calcolo



Automi e calcolo

- **Automa**: *macchina astratta*
- Riceve dall'esterno i dati ed una descrizione dell'algoritmo richiesto
- Interpreta un linguaggio, detto linguaggio macchina dell'automa
- Vincolo su numero di componenti, finito
- Vincolo su alfabeti di ingresso e di uscita, composti da un numero finito di simboli



Riconoscimento di linguaggi

- Data una stringa x , stabilire se essa appartiene ad L (problema dell'*appartenenza*, o *membership*)
- L. tipo 3: riconosciuti da *macchine a stati finiti (FSM)*
 - Es.: $\{a^n b : n \geq 0\}$, gen. da $S \rightarrow aS \mid b$
- L. tipo 2: *automi a pila non deterministici (NPDA)*
 - Es.: $\{a^n b^n : n \geq 1\}$ gen. da $S \rightarrow aSb \mid ab$
- L. tipo 1: *automi limitati linearmente (LBA)*
 - Es.: $\{a^n b^n c^n : n \geq 1\}$
- L. tipo 0: riconosciuti da *macchine di Turing (TM)*
 - Però semi-decidibili: se $x \notin L$, il processo può non terminare!

Macchina a stati finiti (FSM)



Macchina a stati finiti (FSM)

- $M = <\Sigma, Q, \delta, q_0, F>$
- $\Sigma = \{\sigma_1, \dots, \sigma_n\}$: alfabeto di input
- $Q = \{q_0, \dots, q_n\}$: insieme finito non vuoto di stati
- $F \subseteq Q$: insieme di stati finali
- $q_0 \in Q$: stato iniziale
- $\delta: Q \times \Sigma \rightarrow Q$: funzione di transizione
 - In base allo stato e al simbolo di input attuali ... determina lo stato successivo

Linguaggio riconosciuto da FSM

- Funzione di transiz. estesa a stringhe: $\delta: Q \times \Sigma^* \rightarrow P(Q)$
 - $\delta(q, \epsilon) = q$
 - $\delta(q, ax) = \delta(\delta(q, a), x)$, con $a \in \Sigma$, $x \in \Sigma^*$
- Linguaggio riconosciuto da una macchina M:
 - $L(M) = \{x \in \Sigma^* : \delta(q_0, x) \in F\}$
- FSM riconoscono tutti e soli i *linguaggi regolari*
- Rappresentazione della funzione di transizione
 - Tavella di transizione*
 - Diagramma degli stati*

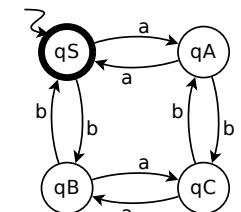
Esempio di FSM

- $M = <\{a, b\}, \{q_S, q_A, q_B, q_C\}, \delta, q_S, \{q_S\}>$

δ	a	b
q_S	q_A	q_B
q_A	q_S	q_C
q_B	q_C	q_S
q_C	q_B	q_A

- Grammatica equivalente:

$$\begin{array}{l} S \rightarrow aA \mid bB \mid \epsilon \\ A \rightarrow aS \mid bC \\ B \rightarrow bS \mid aC \\ C \rightarrow aB \mid bA \end{array}$$



Stringhe con a in numero pari e b in numero pari

FSM non deterministica

- $M = <\Sigma, Q, \delta_N, q_0, F>$
- $\Sigma = \{\sigma_1, \dots, \sigma_n\}$: alfabeto di input
- $Q = \{q_0, \dots, q_m\}$: insieme finito non vuoto di stati
- $F \subseteq Q$: insieme di stati finali
- $q_0 \in Q$: stato iniziale
- $\delta_N: Q \times \Sigma \rightarrow P(Q)$: funzione di transizione, determina insieme di stati successivi
 - $P(Q)$ è l'insieme delle parti di Q , ossia l'insieme di tutti i possibili sottoinsiemi di Q

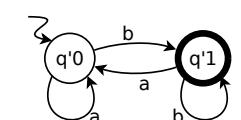
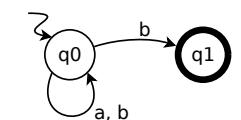
Esempio di NFSM

- $M = <\{a, b\}, \{q_0, q_1\}, \delta, q_0, \{q_1\}>$

δ	a	b
$\{q_0\}$	$\{q_0\}$	$\{q_0, q_1\}$
$\{q_1\}$	\emptyset	\emptyset

- $\{q_0\} \xrightarrow{a} \{q'_0, q_1\} \xrightarrow{b} \{q'_1\}$

δ	a	b
$\{q_0\}$	$\{q_0\}$	$\{q_0, q_1\}$
$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0, q_1\}$



Accetta qualsiasi stringa terminante con b

Equivalenza FSM / NFSM

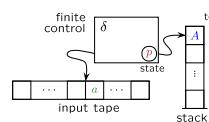
- Per ogni stato / ingresso, definiti più stati successivi
- Non-determinismo: calcolo = albero di computazioni autonome, anziché traiettoria in uno spazio di stati
- Nel casi di FSM, non-determinismo non aggiunge potere computazionale
 - FSM / NFSM: formalismi equivalenti
 - FSM è un caso particolare di NFSM
 - Viceversa, ogni elemento di $P(Q)$ di NFSM diventa uno stato di FSM
 - $P(Q)$ contiene $2^{|Q|}$ elementi



Automa a pila (PDA)

Automa a pila (PDA)

- Simile a FSM, ma dotato di memoria infinita, organizzata a pila
 - Si può accedere solo alla cima della pila
 - Lettura del simbolo in cima
 - Sostituzione simbolo in cima con nuova stringa (anche ϵ)
- In forma non-deterministica, permette di riconoscere i linguaggi non contestuali
- In forma deterministica, riconosce solo i linguaggi non contestuali deterministicici (sottoclasse)
 - Base dei comuni linguaggi di programmazione



Definizione di PDA

- $M = \langle \Sigma, \Gamma, z_0, Q, q_0, F, \delta \rangle$
- $\Sigma = \{\sigma_1, \dots, \sigma_n\}$: alfabeto di input
- $\Gamma = \{z_0, \dots, z_m\}$: simboli della pila
- $z_0 \in \Gamma$: simbolo di pila iniziale
- $Q = \{q_0, \dots, q_k\}$: insieme finito non vuoto di stati
- $q_0 \in Q$: stato iniziale; $F \subseteq Q$: insieme di stati finali
- $\delta: Q \times \Sigma \times \Gamma \rightarrow Q \times \Gamma^*$: funzione di transizione
 - In base a stato, simbolo di input, simbolo in cima a pila ...
 - Determina stato successivo e simboli inseriti nella pila
 - Per rimuovere il simbolo in cima alla pila, si scrive ϵ

Esempio di PDA

- Automa a pila M che riconosce $L = \{a^n b^n, n \geq 1\}$
- $M = \langle \{a, b\}, \{Z_0, A_0, A\}, Z_0, \{q_0, q_1, q_2\}, q_0, \{q_2\}, \delta \rangle$
 - A serve a ricordare la presenza delle a
 - q_0 : si memorizzano le a
 - q_1 : si confrontano le b con quanto memorizzato
 - q_2 : stato finale

δ	Z_0, a	Z_0, b	A_0, a	A_0, b	A, a	A, b
q_0	A_0, q_0		AA_0, q_0	ϵ, q_2	AA, q_0	ϵ, q_1
q_1				ϵ, q_2		ϵ, q_1
q_2						

PDA non deterministico (NPDA)

- $A = \langle \Sigma, \Gamma, z_0, Q, q_0, F, \delta_N \rangle$
- $\delta_N: Q \times \Sigma \times \Gamma \rightarrow P(Q \times \Gamma^*)$: funzione di transizione, determina gli stati e i simboli di pila successivi
 - Es. $\delta(p, a, A) = \{(q, BA), (r, \epsilon)\}$ (due transizioni)
 - Simbolo A in cima alla pila sostituito dalla stringa di caratteri BA , nuovo stato interno q
 - Simbolo A in cima alla rimosso (sostituito da ϵ), nuovo stato interno r
- NPDA: maggiore potere computazionale di PDA
 - $L = \{a^n b^n\} \cup \{a^n b^{2n}\}, n \geq 0$
 - $S \rightarrow A|B, A \rightarrow aAb|\epsilon, B \rightarrow aBbb|\epsilon$



Macchina di Turing (TM)



Macchina di Turing (TM)

- Automa con testina di scrittura/lettura su nastro bidirezionale “illimitato”
- Ad ogni passo:
 - Si trova in un certo stato
 - Legge un simbolo dal nastro
- In base alla funzione di transizione (deterministica):
 - Scrive un simbolo sul nastro
 - Sposta la testina di una posizione
 - Cambia lo stato
- Può simulare ogni altro modello di calcolo noto!

TM deterministica

- $M = \langle \Sigma, Q, q_0, F, \delta \rangle$
- $\Sigma = \{\sigma_1, \dots, \sigma_n, b\}$: alfabeto di input (+ **blank**)
- $Q = \{q_0, \dots, q_m\}$: insieme finito non vuoto di stati
- $q_0 \in Q$: stato iniziale
- $F \subseteq Q$: insieme di stati finali
- $\delta: Q \times \Sigma \rightarrow Q \times \Sigma \times \{d, s, i\}$: funzione di transizione
 - Determina lo stato successivo, il simbolo successivo, lo spostamento della testina

TM non deterministica (NTM)

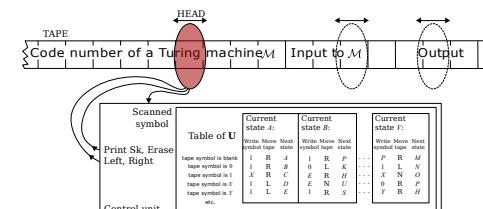
- $M = \langle \Sigma, Q, q_0, F, \delta_N \rangle$
- $\delta_N: Q \times \Sigma \rightarrow P(Q \times \Sigma \times \{d, s, i\})$: funzione di transizione
 - Determina le configurazioni successive (nuovo stato, simbolo scritto, spostamento della testina)
- Grado di non-determinismo: massimo numero di figli di un nodo dell'albero di computazione
- NTM: **stessa potenza computazionale di TM**
 - Data NTM, M , con grado di non-determinismo n ...
 - $\exists M'$, una equivalente MT, in grado di simulare M
- Ma (finora...) NTM **più efficiente**
 - k passi di M richiedono k' ($\propto kn^k$, asintot.) passi di M'

Macchina di Turing universale (UTM)

- UTM realizza la computazione:
 - $(q_{0U}, D_M \# x) \xrightarrow{*} (\alpha, q_{fU}, \beta) \dots$
 - q_{0U} , stato iniziale; q_{fU} , stato finale
 - $D_M \# x$: input; D_M : descrizione di M (funz. di transizione)
- $\Leftrightarrow M$ realizza la computazione: $(q_0, x) \xrightarrow{*} (\alpha, q_f, \beta)$
 - q_0 , stato iniziale; q_f , stato finale; x , input
- Regole di transizione di $M \rightarrow$ Sequenza di quintuple
 - $D_M = d_1 \# d_2 \# \dots \# d_n$
 - $q_i \# \sigma_j \# q_h \# \sigma_k \# t_l \Leftrightarrow \delta(q_i, \sigma_j) = (q_h, \sigma_k, t_l)$

Funzionamento della UTM

- UTM **interpreta** un arbitrario programma su nastro
 - Dato lo stesso input, UTM produce lo stesso output di M
 - Per ogni simbolo letto in x (input di M), scorre la lista di regole, per scegliere la giusta transizione





Calcolabilità

Calcolabilità

- **Tesi di Church-Turing:** problema intuitivamente calcolabile → esiste TM in grado di calcolarlo
 - TM: formalismo non meno potente di ogni altro modello di calcolo proposto finora
 - Funzioni ricorsive, lambda-calcolo, macchine a registri...
- Ma esistono **problemi irrisolvibili** (nel caso generale)
 - Attenzione: non si dice niente sulla singola istanza!
- **Teorema di incompletezza di Gödel**
 - ∀ formalizzazione della matematica che assiomatizza \mathbb{N}
 - → ∃ proposizione né dimostrabile né confutabile



tomamic.github.io/fondinfo

23/44



tomamic.github.io/fondinfo

22/44

Paradossi classici

- Paradosso del mentitore
 - Questa frase è falsa
 - Epimenide di Creta afferma: "*I cretesi sono bugiardi*"
- Paradosso del barbiere
 - Se un barbiere fa la barba a tutti e soli coloro che non si fanno la barba da soli, chi sbarba il barbiere?
- Paradosso di Russell
 - Consideriamo insiemi di insiemi e supponiamo che un insieme possa contenere se stesso
 - Sia T l'insieme di tutti gli insiemi che non contengono se stessi; possiamo stabilire se T contiene T ?

Problema della terminazione

- Predicato della terminazione, **non calcolabile**
 - $h(D_M, x) = 1$, se M con input x termina
 - $h(D_M, x) = 0$, se M con input x non termina
- Costruiamo per assurdo H , TM che calcola h
- Costruiamo quindi G
 - $g(D_M) = 0$, se $h(D_M, D_M) = 0$
 - Indefinito, altrimenti (ossia G cicla all'infinito, se $h = 1$)
- Ma è assurdo:
 - $g(D_G)$ è indefinita, se $g(D_G) = 0$ (definita)
 - $g(D_G) = 0$ (definita) se $g(D_G)$ è indefinita

M: macchina di Turing; D_M : rappresentazione di M come stringa



tomamic.github.io/fondinfo

25/44



tomamic.github.io/fondinfo

24/44

Più informalmente...

- Funz. g definita in `paradox.py` (Python è *Turing completo*)

```
from absurd import h
def g(file):
    if h(file, file):
        while True: pass
    else:
        return False
g('paradox.py')
```

PYTHON



- Altri problemi indecidibili (corollari)
 - Correttezza: il programma calcola la funzione desiderata?
 - Chiamata: una procedura (o istruzione) sarà eseguita?
 - Equivalenza, ambiguità di grammatiche CF ...

tomamic.github.io/fondinfo

26/44

Macchine a registri

tomamic.github.io/fondinfo

27/44

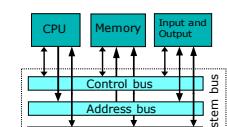
Calcolatore

- Macchina programmabile
 - Memorizza ed elabora automaticamente...
 - Attraverso istruzioni di un programma...
 - Informazioni in formato digitale (I/O)
- Diversi modello di calcolo
 - Definizione di operazioni elementari e concetto stesso di algoritmo
 - Definizione di problema risolvibile / irrisolvibile (calcolabilità): dipende dal modello di calcolo
- Macchina di Turing
 - Modello teorico (A. Turing, 1936)
 - Tesi di Church-Turing: problema intuitivamente calcolabile → esiste TM in grado di calcolarlo



Macchina di von Neumann

- 1941, Z3 (Berlino)
 - A relais, programma su nastro esterno
- 1946, ENIAC (Philadelphia)
 - Balistica, meteorologia, reazioni nucleari
 - Programmazione con cavi
- 1948, Manchester "baby"
 - Programma in memoria
- 1951, sistema IAS (Princeton)
 - Dati e programmi in memoria centrale



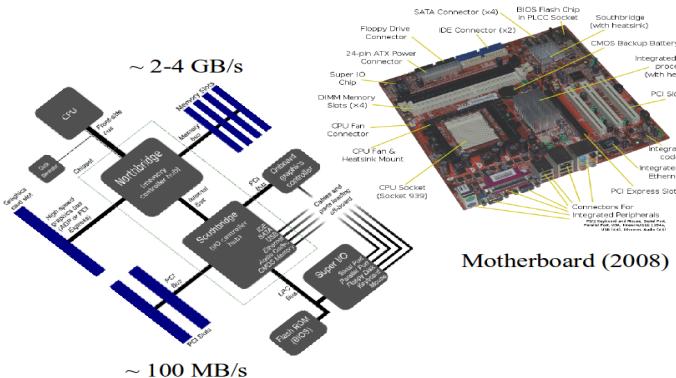
tomamic.github.io/fondinfo

28/44

tomamic.github.io/fondinfo

29/44

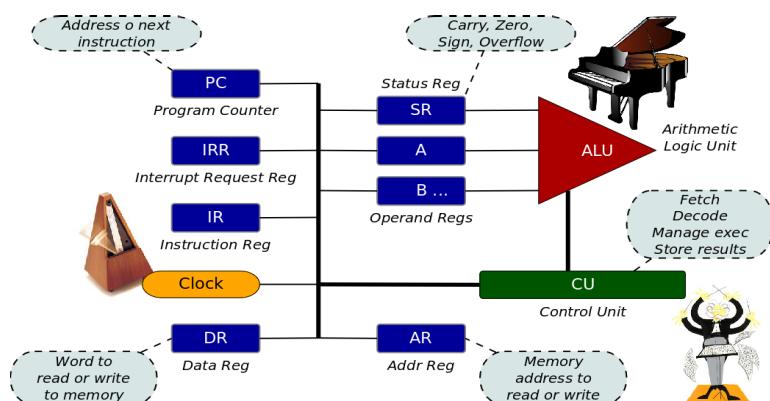
Northbridge e southbridge



Central Processing Unit

- CPU: "cervello" del calcolatore
 - Esegue i programmi
 - Comanda le altre parti del calcolatore
- Composta da due parti:
 - **Control Unit (CU)**: interpreta le istruzioni, comanda le altre parti della CPU, controlla il flusso tra CPU e memoria
 - **Arithmetic Logical Unit (ALU)**: esegue le operazioni aritmetiche e logiche, esegue i confronti tra dati

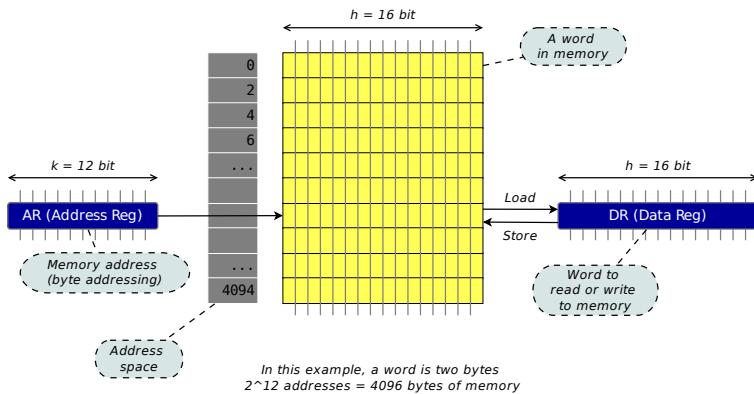
Architettura CPU



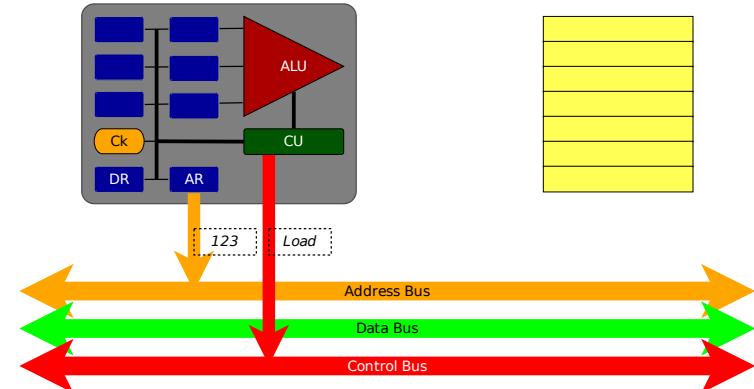
Ciclo principale della CPU

- **Caricamento**: CU preleva l'istruzione dalla locazione di memoria indicata dal registro PC (Program Counter) e la memorizza in IR (Instruction Register)
- **Decodifica**: CU interpreta l'istruzione, legge eventualmente dalla memoria i dati necessari
- **Esecuzione**: CU comanda le parti
- **Memorizzazione**: risultati memorizzati nella memoria centrale o in registri della CPU

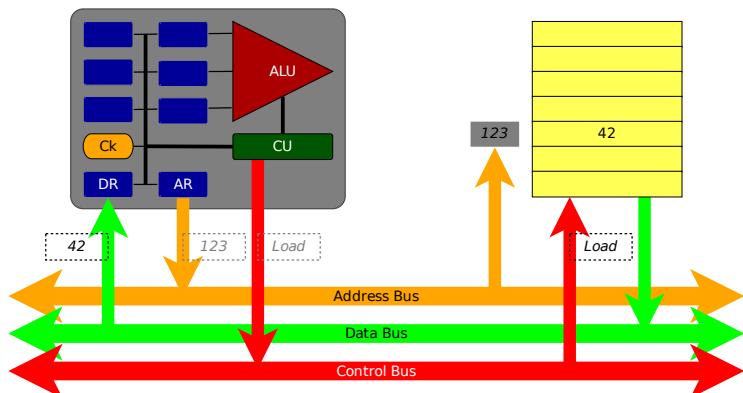
Lettura dalla memoria



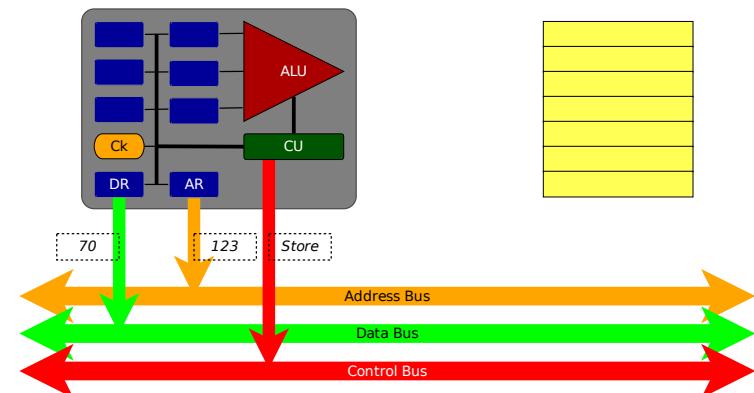
Sequenza di lettura



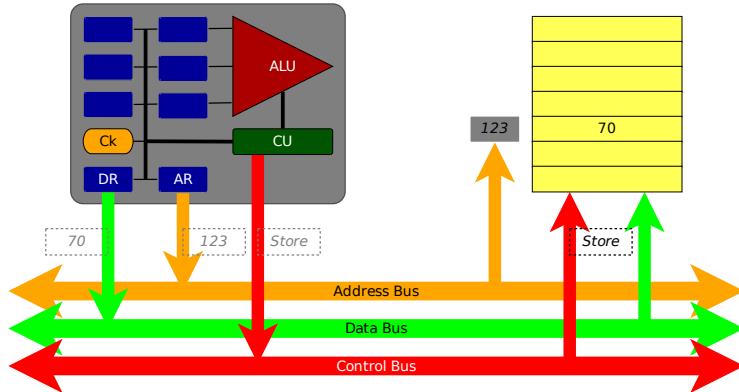
Sequenza di lettura



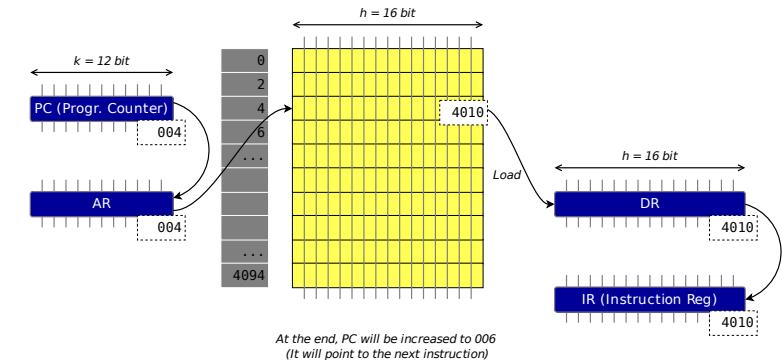
Sequenza di scrittura



Sequenza di scrittura



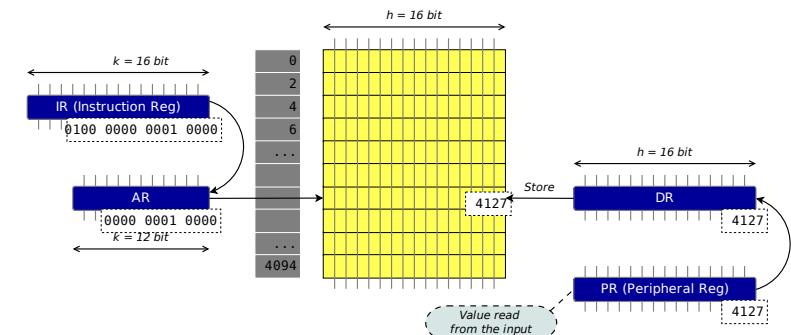
Fetch istruzioni



Interpretazione istruzioni

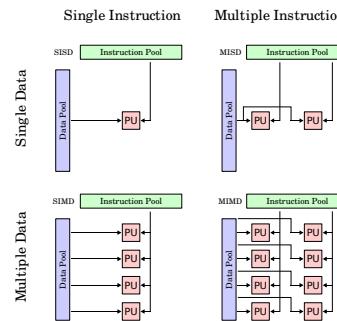
- Al termine della fase di *fetch*, IR contiene l'istruzione da eseguire
 - Codice operativo + operando**
 - Linguaggio macchina*: il significato dipende dalla CPU
- Nell'esempio: **4 010** (16) = **0100** 0000 0001 0000 (2)
 - Codice operativo = **0100**
 - Es. Leggi una parola dal registro delle periferiche...
 - E memorizzala in un indirizzo di memoria (operando)

Esecuzione istruzioni



Classificazione di Flynn

- **Parallelismo:** migliori prestazioni, a parità di velocità sulla singola istruzione
- **SISD:** una operazione alla volta; macchine tradizionali a singolo processore e core
- **MISD:** insolite, per tolleranza ai guasti; sistemi eterogenei, sugli stessi dati, devono dare gli stessi risultati
- **SIMD:** operazioni naturalmente parallelizzabili; unità di calcolo vettoriale e GPU
- **MIMD:** istruzioni diverse su dati diversi; architetture con più core o processori autonomi, sistemi distribuiti



Assembler

- **Linguaggio macchina:** definisce il set di istruzioni comprensibile dalla CPU
 - **CISC:** Complex Instruction Set Computing
 - **RISC:** Reduced instruction set Computing
- Assembler: traduce da **linguaggio assembly** (mnemonico) a linguaggio macchina (mapping 1~1)
- Es. **Assembly x86** → macchina (istruzioni di varia lunghezza)

MOV AH, 11 → 1011 0 100 00001011

ASSEMBLY

- 4 bit di op-code (**1011**), tipo di istruzione
- Bit **w** (**0**): operazione a 8 o 16 bit, (0 o 1 resp.)
- 3 bit per registro destinazione (**100**)
- 8 bit di dato per operando: $11_{(10)} = 00001011_{(2)}$

<Domande?>

Michele Tomaiuolo
Palazzina 1, int. 5708
Ingegneria dell'Informazione, UniPR



Introduzione
all'informatica

Michele Tomaiuolo
Ingegneria dell'Informazione, UniPR

Complessità computazionale



Problemi e complessità

- Problemi *non risolvibili*
 - Es. Questa frase è falsa
 - Incompletezza Gödel; indecidibilità terminazione
- Risolvibili
 - *Non trattabili* (costo "esponenziale")
 - Trattabili (costo accettabile, "polinomiale")
- **Calcolabilità:** classificare risolvibili e non risolvibili
- **Complessità:** "facili" e "difficili"

Ricerca lineare

```
def linear_search(v: list, value) -> int:  
    '''v: not necessarily sorted'''  
  
    for i in range(len(v)):  
        if v[i] == value:  
            return i  
  
    return -1
```

PYTHON

tomamic.github.io/fondinfo

2/30

tomamic.github.io/fondinfo

3/30

Ricerca binaria

```
def binary_search(v: list, value) -> int:  
    '''v: sorted list'''  
  
    begin, end = 0, len(v)  
    while begin < end:  
        middle = (begin + end) // 2  
        if v[middle] > value:  
            end = middle  
        elif v[middle] < value:  
            begin = middle  
        else:  
            return middle  
  
    return -1
```

PYTHON

tomamic.github.io/fondinfo

4/30

Costo di un algoritmo

- **Spazio**, memoria richiesta
- **Tempo**, necessario all'esecuzione
 - Di solito si contano i cicli, in funzione di n
 - O i confronti/scambi tra elementi dell'array
 - Array in memoria centrale, accesso lento
 - Altre variabili nei registri del processore
 - Test e misure empiriche

tomamic.github.io/fondinfo

5/30

Confronto tra algoritmi

- Caso peggiore negli algoritmi di ricerca: elemento non presente
- Ricerca lineare: n confronti
- Ricerca binaria: $\lceil \log_2(n) \rceil$ confronti
 - A ogni iterazione l'insieme è dimezzato
 - Quante volte n dev'essere diviso per 2, per arrivare ad 1?
 - $2^k \geq n \rightarrow k \geq \log_2(n)$

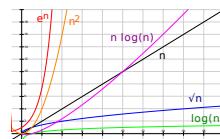
Def. di complessità

- Una funzione $f(n)$ ha **ordine** $O(g(n))$ sse:
 - Esistono due costanti positive c ed m , tali che
 - $|f(n)| \leq c|g(n)| \forall n > m$
- Un algoritmo ha una **complessità** $O(g(n))$ sse:
 - Il tempo di calcolo $t(n)$, sufficiente per eseguire l'algoritmo con ogni istanza(*) di dimensione n , ha ordine $O(g(n))$

(*) *Istanza: insieme di dati su cui è definito il problema; quindi per la complessità conta il caso peggiore*

Analisi asintotica

- Per n abbastanza grande, a meno di una costante moltiplicativa, $f(n)$ non supera in modulo $g(n)$
- Comportamento dell'algoritmo al limite, per dimensione delle istanze tendente all'infinito
- Es. $n = 1\,000\,000$
 - Ricerca lineare: 1'000'000 cicli
 - Ricerca binaria: 20 cicli



Complessità intrinseca

- Limite inferiore di complessità di un problema
- Una funzione $f(n)$ è $\Omega(g(n))$ sse
 - Esistono due costanti positive c e m tali che
 - $|f(n)| \geq c|g(n)| \forall n > m$
- Un problema ha una **delimitazione inferiore** alla complessità $\Omega(g(n))$ sse
 - Per ogni algoritmo risolutore...
 - \exists una istanza (caso peggiore)...
 - per cui il tempo di calcolo $t(n)$ è $\Omega(g(n))$

Algoritmo ottimale

- Algoritmo che risolve un problema P, con le due seguenti condizioni:
 - Costo di esecuzione $O(g(n))$
 - P ha una delimitazione inferiore $\Omega(g(n))$
- Es. L'algoritmo della ricerca binaria è ottimale
 - È dimostrato che $\log_2(n)$ è la complessità intrinseca della ricerca
 - Ma ricerca lineare funziona anche per liste non ordinate!



10/30

Algoritmi di ordinamento

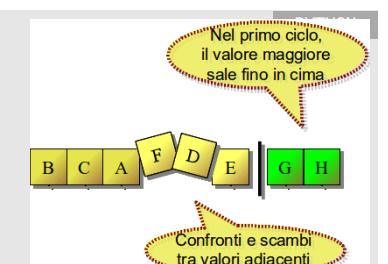
Algoritmi di ordinamento

- Ricerca binaria: importante avere dati ordinati
 - Ordinateur, ordenador
- Algoritmi di ordinamento più semplici hanno complessità n^2
 - Confronto tra ciascun elemento e gli altri
- Algoritmi di ordinamento *divide et impera*
 - Complessità $n \cdot \log_2(n)$
 - Complessità intrinseca

Bubble sort

```
def swap(v: list, i: int, j: int):
    v[i], v[j] = v[j], v[i]

def bubble_sort(v: list):
    end = len(v) - 1
    while end > 0:
        for i in range(end):
            if v[i] > v[i + 1]:
                swap(v, i, i + 1)
        end -= 1
```



Analisi Bubble Sort

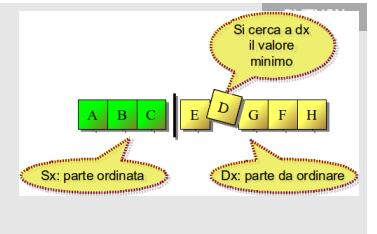
- Gli elementi maggiori salgono rapidamente, “*come bollicine di champagne*”
- Caso peggiore: lista rovesciata
 - Numero di confronti e scambi: $n^2/2$
 - $(n-1)+(n-2)+\dots+2+1 = n(n-1)/2 = n^2/2 - n/2 \approx n^2/2$
 - (Applicata la formula di Gauss per la somma dei primi numeri)
 - Complessità n^2
- Anche in media, circa stessi valori

Selection Sort

```
def selection_sort(v: list):
    for i in range(len(v) - 1):
        min_pos = i

        for j in range(i + 1, len(v)):
            if v[j] < v[min_pos]:
                min_pos = j

        swap(v, pos_min, i)
```



Analisi Selection Sort

- Ad ogni ciclo principale, si seleziona il valore minore
- Caso peggiore: lista rovesciata
 - Numero di confronti $n \cdot (n-1)/2$; complessità n^2
 - Numero di scambi: $n-1$ scambi
- Anche in media, circa stessi valori

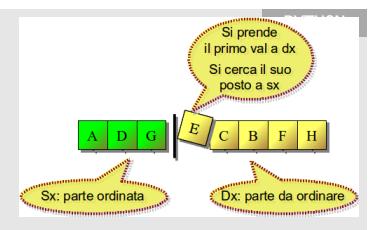
Numero di confronti: $(n - 1) + (n - 2) + (n - 3) + \dots + 0$
Si applica Gauss

Insertion sort

```
def insertion_sort(v: list):
    for i in range(1, n):
        value = v[i]

        for j in range(i - 1, -1, -1):
            if v[j] <= value: break
            v[j + 1] = v[j]

        v[j + 1] = value
```

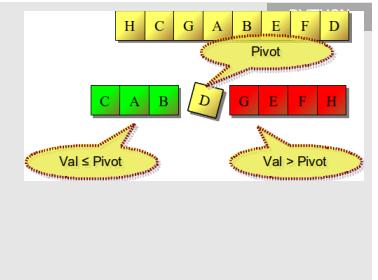


Analisi Insertion Sort

- La prima parte è ordinata, vi si inserisce un elemento alla volta, più facile trovare il posto
- Caso peggiore: lista rovesciata
 - Cicli: $1+2+\dots+(n-1) = n \cdot (n-1)/2$; compl: $O(n^2)$
- In media si scorre solo 1/2 della prima parte
 - In media $n^2/4$ confronti e $n^2/4$ scambi
- Ottimizzazioni
 - Ricerca binaria in parte ordinata, ma scambi
 - Inserimento a coppie, o gruppi

Quick Sort

```
def quick_sort(v: list, begin=0, end=len(v)):  
    if end - begin > 1:  
        pivot = v[end - 1]  
        j = begin  
        for i in range(begin, end - 1):  
            if v[i] < pivot:  
                swap(v, i, j)  
                j += 1  
        swap(v, end - 1, j)  
        quick_sort(v, begin, j)  
        quick_sort(v, j + 1, end)
```

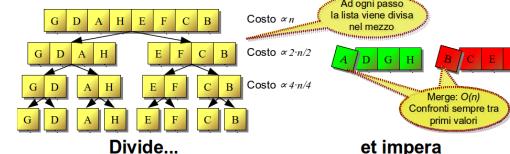


Analisi Quick Sort

- Dato un insieme, sceglie un valore **pivot**
- Crea due sottoinsiemi: $x \leq \text{pivot}$, $x > \text{pivot}$
- Stesso algoritmo sui 2 insiemi (ricorsione)
- Caso peggiore: lista rovesciata, n^2
 - Dipende da scelta pivot, ma esiste sempre
- Caso medio: $n \cdot \log_2(n)$
 - $t(n) = \alpha \cdot n + 2 \cdot t(n/2)$
 - Sostituzione k volte: $t(n) = \alpha \cdot k \cdot n + 2^k t(n/2^k) \dots$

Merge Sort

```
def merge_sort(v, begin=0, end=len(v)):  
    '''In v, sort elements in range(begin, end)'''  
    if end - begin > 1:  
        middle = (begin + end) / 2  
        merge_sort(v, begin, middle)  
        merge_sort(v, middle, end)  
        merge(v, begin, middle, end)
```



Merge, con appoggio

```
def merge(v, begin, middle, end):
    '''Merge two sorted portions of a single list'''
    i1, i2, n = begin, middle, end - begin
    result = []

    for k in range(n):
        if i1 < middle and (i2 >= end or v[i1] <= v[i2]):
            result.append(v[i1])
            i1 += 1
        else:
            result.append(v[i2])
            i2 += 1

    for k in range(n):
        cards[begin + k] = result[k]
```

PYTHON

Analisi Merge Sort

- Simile a Quick Sort, ma non si sceglie pivot
- La fusione ha complessità lineare
- Caso peggiore, caso medio: $n \cdot \log_2(n)$
- **Spazio:** la fusione richiede altra memoria: n
 - Ma si può evitare il costo con spostamenti *in place*
- Accessi sequenziali, buon uso *cache*
- Integraz. con Insertion Sort (Python, Java7)

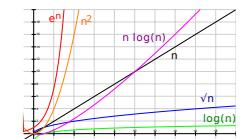
Classi di complessità



DII

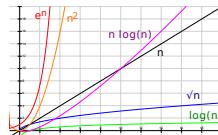
Classi di complessità

- Costante: numero op. non dipende da n , dim. istanza
- Sotto-lineare: n^k , $k < 1$; $\log(n)$, ricerca binaria
- Lineare: numero op. $\propto n$, ricerca lineare
- Sovra-lineare: $n \cdot \log(n)$, merge sort
- Polinomiale: n^k , $k \geq 2$, insertion sort
- **Algoritmo efficiente:** fino a classe polinomiale
- **Problema trattabile:** \exists algoritmo efficiente



Complessità esponenziale

- Complessità esponenziale: k^n
 - Es. elenco sottinsiemi, strategia perfetta per scacchi
- Complessità super-esponenziale: $n!$, n^n , ...
 - Es. elenco permutazioni
- **Problemi intrattabili**
 - \nexists algoritmo efficiente
 - Soluzioni non esatte/ottime, euristiche
 - Ma minimi locali...



Problemi P ed NP

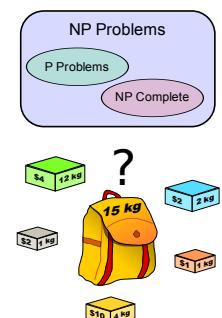
- Problemi **P**: \exists algoritmo *deterministico polinomiale*
- **NP**: \exists algoritmo *non-deterministico polinomiale*
 - Su macchine deterministiche: non noto algoritmo polinomiale per la **ricerca** di una soluzione...
 - Ma algoritmo polinomiale per la **verifica** di una soluzione
- Esempio: fattorizzazione di grandi numeri
 - Ricerca: quali sono i fattori primi di un numero di n cifre?
 - Verifica: è vero che x è divisore di y ?
- **Non è dimostrato che $P \neq NP$, né che $P = NP$**
 - Millennium Prize Problems: 1M\$
 - Se $P = NP$, trovare i fattori primi di un numero o verificarli: stessa classe di complessità

Complessità dei linguaggi

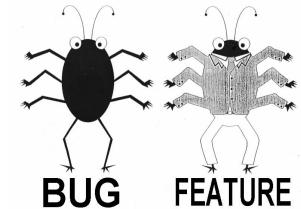
- *Linguaggi di classe P / NP*: stringa x riconosciuta in tempo polinomiale rispetto a $|x|$...
 - **P**: da una macchina di Turing deterministica (*DTM*)
 - **NP**: da una macchina di Turing non-deterministica (*NTM*)
 - Sappiamo che $P \subseteq NP$ (DTM: caso particolare di NTM)
- *Linguaggi di classe EXP*: stringa x riconosciuta in tempo esponenziale rispetto a $|x|$ da una DTM
 - NTM: simulata da DTM in tempo esponenziale
 - Quindi $NP \subseteq EXP$
- $P \subseteq NP \subseteq EXP$

Problemi NP-completi

- Ogni problema NP può essere ricondotto ad un problema **NP-completo** con algoritmo deterministico *polinomiale*
 - *Lower-bound* deterministico esponenziale per uno dei problemi NP-completi? $\Rightarrow P \neq NP$
 - Oppure, *soluzione* con algoritmo deterministico polinomiale? $\Rightarrow P = NP$
- Esempio: *SAT*
 - Data una formula booleana PdS , è soddisfacibile?
 - \exists combinazione di input che dà risultato vero?
- Esempio: *Knapsack*
 - \exists combinazione di elementi che realizza utilità $\geq V$, con peso $\leq W$?



Qualità del software



<Domande?>

Michele Tomaiuolo
Palazzina 1, int. 5708
Ingegneria dell'Informazione, UniPR

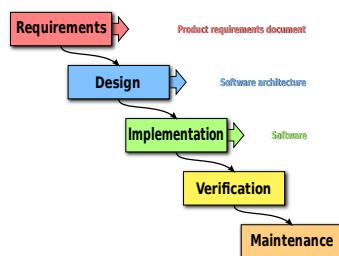


Introduzione
all'informatica

Michele Tomaiuolo
Ingegneria dell'Informazione, UniPR

Ciclo di vita del software

- **Analisi**
 - Modello, requisiti, fattibilità
- **Progetto e implementazione**
 - Componenti architetturali... dettaglio classi
- **Collaudo**
 - Rispetto requisiti, qualità sw
- **Rilascio e manutenzione**
 - 40%-80% del costo totale (DoD, HP)
 - Non noti o non colti correttamente i requisiti
 - Cambiano le condizioni operative ...



[Winston W. Royce, 1970](#) - [Robert L. Glass, 2001](#)

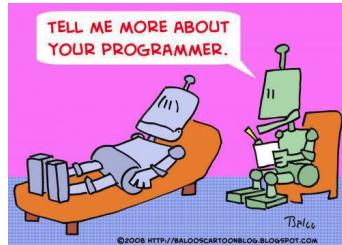
Evoluzione di un sistema sw

- Evoluzione ineliminabile per molti sistemi
 - Prestazioni, qualità, funzionalità (manutenzione *perfettiva*, ~60%)
 - Anomalie ed errori (manutenzione *correttiva*, ~20%)
 - Mutamenti dell'ambiente (manutenzione *adattativa*, ~20%)
- Sviluppo iterativo e metodologie agili
 - Rilascio frequente ed incrementale
 - <http://agilemanifesto.org/>



Qualità del software

- Le qualità su cui si basa la valutazione di un sistema software possono essere:
 - **Interne**, riguardano le caratteristiche legate al **processo** di sviluppo e non sono direttamente visibili agli utenti
 - **Esterne**, riguardano le funzionalità fornite dal **prodotto** sw e sono direttamente visibili agli utenti
- Le categorie sono legate:
 - *Product quality is process quality*



Qualità esterne

- **Correttezza e affidabilità**: il sistema rispetta le specifiche, l'utente può affidarsi al programma
- **Robustezza**: il sistema si comporta in modo ragionevole anche fuori dalle specifiche
- **Efficienza**: usa bene le risorse di calcolo
- **Scalabilità**: migliori prestazioni con più risorse
- **Sicurezza**: riservatezza, autenticazione, autorizzazione, accounting
- **Facilità d'uso**: interfaccia utente permette di interagire in modo naturale

Qualità interne

- **Verificabilità**: sistema basato su modello formale
- **Riusabilità**: parti per costruire nuovi sistemi
- **Manutenibilità**: riparabilità, evolvibilità (nuove specifiche), adattabilità (cambiamenti ambiente)
- **Interoperabilità**: capacità di co-operare con altri sistemi, anche di altri produttori
- **Portabilità**: adatto a più piattaforme hw/sw
- **Comprendibilità**: codice leggibile, documentato
- **Modularità**: interazione tra componenti coesi

Specifiche

- Rispetto a cosa valutiamo **correttezza** o **affidabilità** di un programma?
- Idea del programmatore
 - Non formulata, non documentata
 - Incompleta, mutevole, facilmente dimenticata
- Specifiche (formali o informali)
 - Formulate, scritte, studiate e condivise
→ Parte del progetto e del programma
 - Spec. assiomatiche: espressioni logiche o asserzioni
→ **Precondizioni**, **postcondizioni** e **invarianti**



Pre- e post-condizioni

- **Precondizioni**
 - Stabiliscono se è possibile chiamare un metodo
 - Prerequisiti per l'attivazione
- **Postcondizioni**
 - Stabiliscono se il metodo restituisce il valore atteso, cioè se produce l'effetto desiderato
 - ... In relazione ai parametri (che soddisfano le precondizioni)
 - Definiscono il significato del metodo
- **Divisione delle responsabilità** tra moduli
 - Errore del codice *chiamante (client)* se precondizioni non soddisfatte
 - Errore del codice *chiamato (server)*, se postcondizioni non soddisfatte

Responsabilità e contratti

- **Precondizioni + postcondizioni = contratto**
 - ... tra modulo chiamante e modulo chiamato
- Infrazione di un contratto: problema serio
 - Errore rispetto alle specifiche
 - Eccezione e/o terminazione
- No **divisione responsabilità** → sovrapposizioni
 - Tutti i moduli assumono molte responsabilità
 - Programmazione difensiva: tutte le parti del programma controllano tutte le condizioni
 - Grosso programma → ancora più grosso

Esempio di contratto

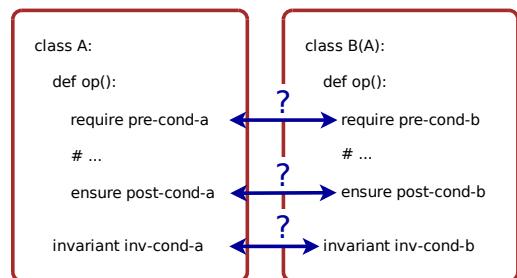
- ```
def sqrt(x: float) -> float
```
- Precondizioni:  $x \geq 0$
  - Postcondizioni:  $\text{abs}(\text{result} * \text{result} - x) \leq 0.00001$
  - Codice chiamante
    - Obblighi: deve passare un numero non negativo
    - Benefici: riceve la radice del numero
  - Codice chiamato
    - Obblighi: restituisce un numero  $r$  tale che  $r * r \approx x$
    - Benefici: può assumere che  $x$  non è negativo

PYTHON

## Invariante di classe

- Vincolo che deve valere per ogni stato stabile di un oggetto, durante tutto il suo ciclo di vita
- Rafforzamento generale di pre- e post-condizioni
- "Criterio di sanità" dell'oggetto
- Deve essere soddisfatto dal costruttore
- Deve essere mantenuto dai metodi pubblici
- Ma non necessariamente da metodi privati o protetti

## Ereditarietà e contratti



- Che relazione c'è tra le asserzioni di una classe e quelle dei suoi discendenti?

## Principio di sostituibilità

- Polimorfismo: possibile esecuzione metodo di una sottoclasse, anziché della classe base
  - I metodi delle sottoclassi possono ridefinire i metodi delle classi base... ma non arbitrariamente
- I contratti della sottoclasse devono *rispettare i contratti della classe base* ("sottocontratti")
  - Precondizioni: non devono essere più forti
  - Postcondizioni: non devono essere più deboli
  - Invarianti di classe: non devono essere più deboli

“Require no more, promise no less”

## Design by contract

- Paradigma proposto nel linguaggio *Eiffel* (Betrand *Meyer*, 1986)
- Uso di asserzioni in varie fasi di sviluppo
  - Progetto: approccio pragmatico alle specifiche
  - Implementazione: guida per la programmazione
  - Documentazione: interfacce con info aggiuntive
  - Collaudo: DbC delimita i casi da testare (per affidabilità)
  - Manutenzione: DbC fa emergere prima gli errori
  - Uso finale: sollevate eccezioni se violazioni

## Asserzioni Python

- Espressioni booleane, simili a prediciati matematici
- Esprimono proprietà semantiche di classi e metodi
- Utili per collaudo e debugging, ma anche documentazione
- Violazione → **AssertionError** (e normalmente *abort*, terminazione programma)

`assert age > 0`

PYTHON

## Asserzioni e contratti

- Asserzioni in genere utili per:
  - Precondizioni, postcondizioni, invarianti di classe
  - Invarianti interne e di controllo del flusso
- Argomenti di metodi pubblici sbagliati → eccezione
  - `ValueError` o `TypeError`
  - Di solito, asservimenti usate per debug...

## Pre- e post-condizioni

```
def sqrt(x: float) -> float:
 ...
 Precondition: x >= 0
 Postcondition: abs(result * result - x) <= 0.00001
 ...
 if x < 0 raise ValueError("sqrt: arg < 0")
 # ...

 assert abs(result * result - x) <= 0.00001
 return result
```

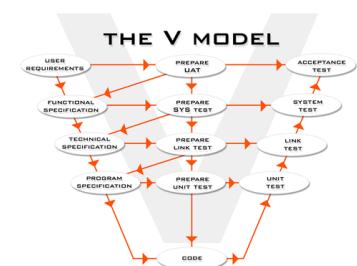
PYTHON

## Verifica e validazione



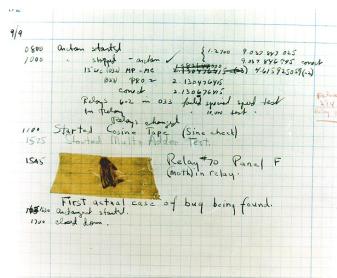
## Verifica e validazione

- Mostrare che il sistema...
  - È conforme alle specifiche
  - Soddisfa i bisogni dell'utente
- Comprende revisione e collaudo del sistema
- **Test case**, derivati dalle specifiche



# Costo dei bug

- Scovare bug non è un compito facile, e nemmeno una esperienza eccitante...
    - Costoso: non è insolito dedicare al testing il 40% del tempo e delle risorse di un progetto
  - **Far emergere** bug in prime fasi dello sviluppo!
    - B. Boehm: se trovare e correggere un problema in fase di specifica dei requisiti costa 1\$...
    - 5\$ in progetto, \$10 in programmazione,
    - \$20 in unit testing, fino a \$200 dopo capite



## Prove formali

- Dimostrazione matematica di un programma: alternativa (~ accademica) al testing
    - Annotazione del programma con asserzioni matematiche: comportamento atteso
    - Proprietà valide per i vari costrutti del programma
  - Prova che post-condizioni verificate, se:
    - Precondizioni verificate
    - Programma termina
  - Dimostrazioni automatiche
    - Se a mano → errori (più che nel programma?)



# Revisione del software

- Analisi del codice (o pseudocodice) per capirne le caratteristiche e le funzionalità
  - Code walk-through**
    - Selezione porzioni di codice e valori di input
    - Simulazione su carta comportamento del sistema
  - Code inspection**, più formale e focalizzato
    - Uso di variabili non inizializzate
    - Loop infiniti
    - Letture di porzioni di memoria non allocata
    - Rilascio improprio della memoria

## Testing

*“Le operazioni di testing possono individuare la presenza di errori nel software ma non ne possono dimostrare la correttezza. (E. Dijkstra)”*

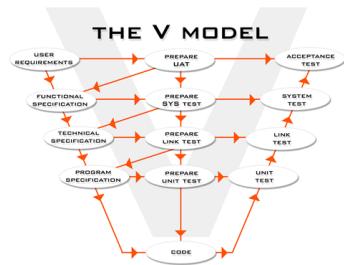
*“Eseguire un programma con l'intento di trovare errori. (Glen Myers, “The art of Software Testing”)”*

- Verificare sistema in un insieme abbastanza ampio di casi... → plausibile comportamento analogo anche nelle restanti situazioni



## Classificazione dei test

- Tipi di test
  - **White box (in the small)**
  - **Black box (in the large)**
- Livelli di test
  - *Unit test*
  - *Integration test*
  - *System test*
- Ripetizione di test
  - *Regression test*



## Testabilità

- Qualità software che facilitano rilevazione errori
  - **Osservabilità** – Disponibili i risultati dei test
  - **Controllabilità** – Possibilità di impostare ingressi e stato del programma prima di eseguire un test
  - **Decomponibilità** – Programma diviso in parti che possono essere testate individualmente
  - **Comprendibilità** – Si capisce il comportamento corretto (desiderato) del programma
- → Sviluppo per testabilità



## White-box testing

### White-box testing

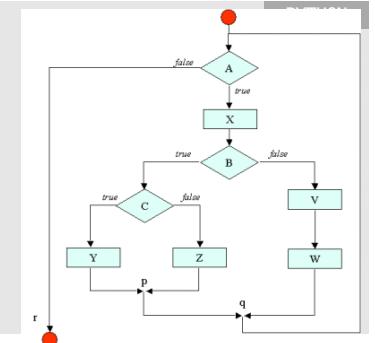
- Test basati sulla conoscenza della struttura interna del codice
- Un errore non può essere scoperto se la parte di codice che lo contiene non viene mai eseguita
- **Statement test**
  - Insieme di test T tali che, eseguendo su tutti i casi di T il programma P, ogni istruzione di P venga eseguita almeno una volta (test utopia?)
  - **Branch test** (copertura delle decisioni)
  - **Branch & condition test** (... condizioni)

## Basic path testing

- Scelto insieme minimo di percorsi per coprire tutte le istruzioni e condizioni (*white box*)
  - Tracciare diagramma di flusso
  - Astrarre il diagramma in un grafo di flusso
  - Complessità ciclomatica  $n$  = metrica di test
  - Trovare  $n$  casi di test che seguono ciascun cammino indipendente
- Cammino: sequenza di comandi, da inizio a fine
- Cammino indipendente: aggiunge almeno una nuova istruzione rispetto ai cammini già identificati

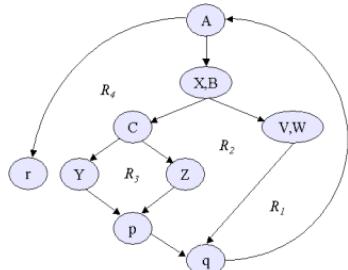
## Diagramma di flusso

```
def f():
 // entry
 while a:
 x()
 if b:
 if c: y()
 else: z()
 else:
 v()
 w()
 # q
 # exit: r
```



## Grafo di flusso

- Piccola astrazione rispetto a diagramma di flusso
- Complessità ciclomatica**, dalla teoria dei grafici:
  - Numero di possibili cammini indipendenti,  $2^n - 1$
  - Numero di regioni del grafo di flusso,  $R_1, R_2, \dots, R_s$
  - Numero di nodi predicato + 1
- A, r
- A, X, B, C, Y, p, q, A, r
- A, X, B, C, Z, p, q, A, r
- A, X, B, V, W, q, A, r



## Black box testing

## Black box testing

- Sistema = scatola nera; si verificano le corrispondenze di input e output
  - White-box testing: impossibile per grandi sistemi
  - Test case scelti in base alle specifiche dei requisiti
- Desiderata: trovare errori...
  - Funzionali: otteniamo i risultati attesi per dati input di un metodo?
  - Interfaccia: dati passati correttamente tra i metodi?
  - Efficienza: il metodo è abbastanza veloce?

## Partizioni d'equivalenza

- Partizionamento ingressi in **classi di equivalenza**
  - Irrealistico testare tutti i possibili ingressi (es. `sqrt`)
  - Ipotesi: sufficiente testare un solo caso per classe
  - Si includono casi limite e valori non validi
  - Precondizioni: riducono il numero di casi di test

```
def swap_elements(v: list, i: int, j: int):
 ...
 Exchange element i and j in list v
 v: empty, one element, more elements
 i, j: one or both indexes out of range... or both in range: i < j, i > j, i = j
 ...
...
```

PYTHON

## Regression testing

- Scopo: trovare errori di regressione
  - Errori in un programma che prima era corretto, ed è stato modificato di recente
  - Un errore di regressione è un errore che prima non c'era
- Dopo la modifica di una parte **P** nel programma **Q**
  - Testare che la parte **P** funzioni correttamente
  - Testare che l'intero programma **Q** non sia stato danneggiato dalla modifica



## <Domande?>

Michele Tomaiuolo  
Palazzina 1, int. 5708  
Ingegneria dell'Informazione, UniPR