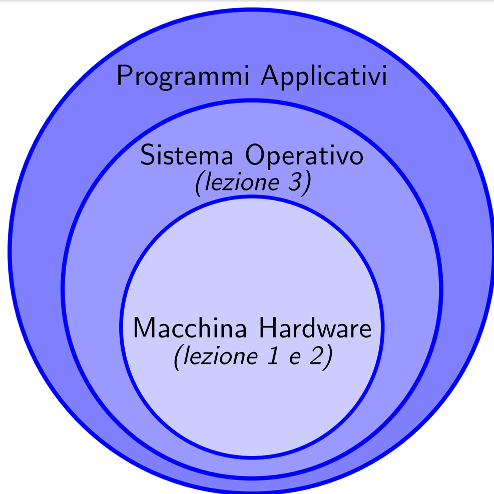
# Informatica e Bioinformatica: Algoritmi

Alessandro Sperduti

20 Marzo 2014

# Programmi Applicativi

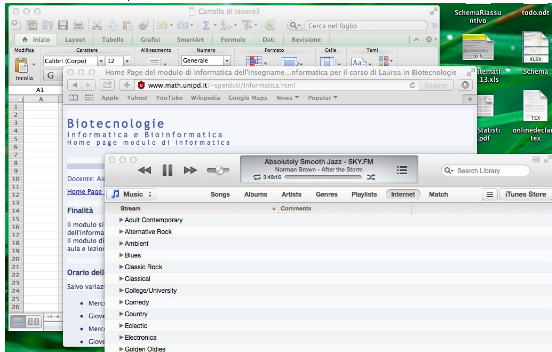



- La macchina hardware permette l'esecuzione di programmi applicativi, che interagiscono con le *risorse* della macchina hardware (CPU, memoria primaria e secondaria, dispositivi di I/O) tramite il sistema operativo.

# Programmi Applicativi

I programmi applicativi

- risolvono i problemi degli utenti:
  - fogli di calcolo, editori di testi, browser, visualizzatore di immagini e/o video, riproduttori audio,...




- sono tipicamente realizzati/definiti tramite un linguaggio di programmazione ad alto livello (C++, Java, Python, ...)
- implementano uno o più *algoritmi*

Cos'è un algoritmo ?

- Definizione informale
  - Insieme di passi da eseguire per risolvere un problema
    - ricetta per cucinare una torta

    - indicazioni stradali per raggiungere una destinazione

Google
Via Paolotti, 63, Padova PD
Via Trieste, 63, 35131 Padova PD
ndro.sperduti@gmail.com
Opzioni percorso

A piedi 600 m, 7 min

Fai attenzione: può includere errori o tratti non adatti ai pedoni

Via Paolotti, 63
Padova PD

1. Procedi in direzione **est** su **Via F. Marzolo** verso **Via Alessandro Cazzato Verolin**

190 m

2. Svolta a **sinistra** e imbocca **Via Giovanni Poleni**

Visualizza mappa completa
Via Trieste, 63
RistorEsu Nord Piovego
Viale Gius
Canale Piovego
Via Leonardo Loredan
Via Leon
Aula Magna di Ingegneria
Dipartimento di Ingegneria Industriale - Sede M
lotti, 63
Via Giovanni Poleni
Via F. Marzolo

    - insieme di passi per calcolare il Massimo Comun Divisore di due numeri interi (`MCD(21,35)=7`)

# Algoritmo

Definizione un pò più formale

- Insieme ordinato (e finito) di passi eseguibili e non ambigui (per risolvere un problema) che giunge (certamente) a terminazione
  - insieme ordinato di passi ⇒ non necessariamente sequenza: possono essere disponibili più *esecutori* (calcolo parallelo)
  - passi eseguibili ⇒ da un esecutore in grado di compiere azioni fattive (e finite)
  - (passi eseguibili) e non ambigui ⇒ l'esecutore deve essere in grado di associare univocamente un passo ad una (o più) azioni
  - che giunge a terminazione ⇒ in modo da trovare una soluzione ad un problema di interesse...

. . . tuttavia, la natura di alcuni problemi potrebbe non richiedere necessariamente la terminazione, ad esempio: funzionamento sistema operativo, video sorveglianza, . . .

# Algoritmo

Un algoritmo ha natura astratta:

- bisogna fare differenza fra un algoritmo e la sua rappresentazione
  - ad esempio, l'algoritmo per convertire le temperature da gradi Celsius (C) a Fahrenheit (F) può essere rappresentato nei seguenti due modi alternativi:
    1. $temperatura_F = \frac{9}{5} temperatura_C + 32$
    2. "moltiplicare la lettura della temperatura in gradi Celsius per $\frac{9}{5}$ e poi aggiungere 32 al prodotto"
- per evitare incomprensioni di comunicazione di algoritmi fra umani/esecutori bisogna fissare una convenzione per rappresentarli $\Rightarrow$ un linguaggio (di programmazione)!
- in effetti, a seconda del livello di astrazione o caratteristiche degli esecutori, si definiscono vari linguaggi per rappresentare algoritmi

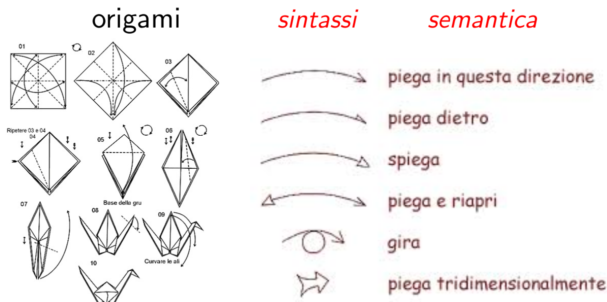
# Primitive

Un linguaggio (di programmazione) è costituito da componenti fondamentali

- chiamate *primitive*
- a partire dalle quali si possono costruire rappresentazioni di algoritmi

Ogni primitiva è costituita da due parti:

- *sintassi*: la rappresentazione simbolica della primitiva
- *semantica*: il significato della primitiva



# Istruzioni mnemoniche

In un linguaggio di programmazione a basso livello, le primitive sono date dalle istruzioni mnemoniche

**Common MIPS instructions.**

Notes: *op*, *funct*, *rd*, *rs*, *rt*, *imm*, *address*, *shamt* refer to fields in the instruction format. The program counter PC is assumed to point to the next instruction (usually 4 + the address of the current instruction). M is the byte-addressed main memory.

| Assembly instruction | Instr. format | *op* *op/funct* | Meaning | Comments |
|---|---|---|---|---|
| `add $rd, $rs, $rt` | R | 0/32 | `$rd = $rs + $rt` | Add contents of two registers |
| `sub $rd, $rs, $rt` | R | 0/34 | `$rd = $rs - $rt` | Subtract contents of two registers |
| `addi $rt, $rs, imm` | I | 8 | `$rt = $rs + imm` | Add signed constant |
| `addu $rd, $rs, $rt` | R | 0/33 | `$rd = $rs + $rt` | Unsigned, no overflow |
| `subu $rd, $rs, $rt` | R | 0/35 | `$rd = $rs - $rt` | Unsigned, no overflow |
| `addiu $rt, $rs, imm` | I | 9 | `$rt = $rs + imm` | Unsigned, no overflow |
| `mfc0 $rt, $rd` | R | 16 | `$rt = $rd` | *rd* = coprocessor register (e.g. epc, cause, status) |
| `mult $rs, $rt` | R | 0/24 | `Hi, Lo = $rs * $rt` | 64 bit signed product in Hi and Lo |
| `multu $rs, $rt` | R | 0/25 | `Hi, Lo = $rs * $rt` | 64 bit unsigned product in Hi and Lo |
| `div $rs, $rt` | R | 0/26 | `Lo = $rs / $rt, Hi = $rs mod $rt` | |
| `divu $rs, $rt` | R | 0/27 | `Lo = $rs / $rt, Hi = $rs mod $rt` (unsigned) | |
| `mfhi $rd` | R | 0/16 | `$rd = Hi` | Get value of Hi |
| `mflo $rd` | R | 0/18 | `$rd = Lo` | Get value of Lo |

... difficili da capire se non si conosce l'architettura di riferimento (in questo caso, MIPS)

# Pseudocodice

Una notazione (linguaggio) meno formale per la rappresentazione di algoritmi è costituito dallo *pseudocodice*

- non costituisce un vero e proprio linguaggio formale
- è utile quando si vogliono esprimere le componenti astratte di un algoritmo
- aiuta nel processo di sviluppo di un algoritmo (senza preoccuparsi del linguaggio di programmazione che poi sarà utilizzato)
- adotta primitive, comuni a molti linguaggi strutturati di programmazione, che si possono raggruppare nelle seguenti classi principali:
  - assegnamento, sequenza, selezione, iterazione, procedura

# Pseudocodice

- *assegnamento*: istruzione che assegna il risultato di un calcolo ad una variabile, che rappresenta un identificativo astratto di una locazione (o insieme di locazioni) in memoria
- *sequenza*: struttura di controllo che permette di eseguire le istruzioni secondo l'ordine in cui sono state scritte
- *selezione*: struttura di controllo che permette di scegliere l'esecuzione di un blocco di istruzioni tra due possibili in base a una condizione
- *iterazione*: struttura di controllo che permette di ripetere l'esecuzione di un blocco di istruzioni in base al valore di una condizione
- *procedura*: struttura che permette di riutilizzare una unità di programma (procedura)

# Pseudocodice

- *assegnamento*:

$$nome \leftarrow espressione$$

esempio:

FondiRimasti ← BilancioConti + BilancioRisparmi

- *sequenza*:

istruzione1
istruzione2
...
istruzioneN

esempio:

BilancioConti ← EntrateConti - UsciteConti
BilancioRisparmi ← EntrateRisparmi - PrelievoRisparmi
FondiRimasti ← BilancioConti + BilancioRisparmi

# Pseudocodice

- *selezione*:

$$\textbf{if } (\textit{condizione}) \textbf{ then } (\textit{azione})$$
$$\textbf{else } (\textit{azione})$$

esempio:

**if** (l'anno è bisestile)
  **then** (TotaleGiornaliero $\leftarrow$ Totale/366)
  **else** (TotaleGiornaliero $\leftarrow$ Totale/365)

il *ramo* **else** è opzionale:

esempio:

**if** (vendite diminuite) **then** (diminuisci prezzo del 5%)

# Pseudocodice

- *iterazione*:

  **while** (*condizione*) **do** (*azione*)

  esempio:

  **while** (rimangono biglietti da vendere) **do** (vendi biglietti)

- *procedura*:

  **procedure** *nome*

  *corpo procedura*

  esempio:

```
procedure Saluti
    Conta ← 3
    while (Conta > 0) do
        (Stampa il messaggio "Saluti"
         Conta ← Conta - 1)
```

Proviamo a fare qualche esercizio!

# Alcune soluzioni

Scrivere una funzione `ordina_dec` che data una lista numerica, passata come parametro, ne restituisca una ordinata dal valore più grande al valore più piccolo

```
def massimo(lista):
    lunghezza = len(lista)
    if (lunghezza > 0):
        max = lista[0]
        posizione = 1
        while (posizione < lunghezza):
            if (lista[posizione] > max):
                max = lista[posizione]
            posizione = posizione + 1
    return max
```

```
def ordina_dec(lista):
    nuova_lista = [ ]
    while (len(lista) > 0):
        max = massimo(lista)
        nuova_lista.append(max)
        lista.remove(max)
    return nuova_lista
```

# Alcune soluzioni

Scrivere una funzione `ordina` che dati come parametri una lista numerica e un carattere, restituisca la lista ordinata dal valore pi grande al valore più piccolo se il carattere è uguale a 'd', altrimenti restituisca la lista ordinata dal valore più piccolo al valore più grande

```
def max_o_min(lista,ord):
    lunghezza = len(lista)
    if (lunghezza > 0):
        m = lista[0]
        posizione = 1
        while (posizione < lunghezza):
            if (ord == 0) :
            # se vero m = minimo
                if (lista[posizione] < m):
                    m = lista[posizione]
            else:
            # altrimenti m = massimo
                if (lista[posizione] > m):
                    m = lista[posizione]
            posizione = posizione + 1
    return m
```

```
def ordina(lista,ordine):
    nuova_lista = [ ]
    while (len(lista) > 0):
        if (ordine == 'd'):
            m = max_o_min(lista,1)
        else:
            m = max_o_min(lista,0)
        nuova_lista.append(m)
        lista.remove(m)
    return nuova_lista
```

# Facciamo un pò di conti..

Ma quante operazioni esegue `ordina` per ordinare $n$ numeri ?
Risposta:

- ogni chiamata della funzione `max_o_min()` deve esaminare tutti gli elementi della lista passata come parametro
  - la prima chiamata di `max_o_min()` lavora sulla lista iniziale di $n$ numeri;
  - la seconda sulla stessa lista dove è stato rimosso il massimo/minimo (quindi contiene $n-1$ elementi);
  - la terza chiamata sulla lista iniziale dove sono stati rimossi 2 elementi (quindi contiene $n-2$ elementi);
  - e così via fino ad avere la lista con un solo elemento;
- quindi in totale, tutte le chiamate di `max_o_min()` esaminano un numero di elementi pari a

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

# Facciamo un pò di conti..

Ma quante operazioni esegue `ordina` per ordinare $n$ numeri ?
Risposta:

- in totale, tutte le chiamate di `max_o_min()` esaminano un numero di elementi pari a

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

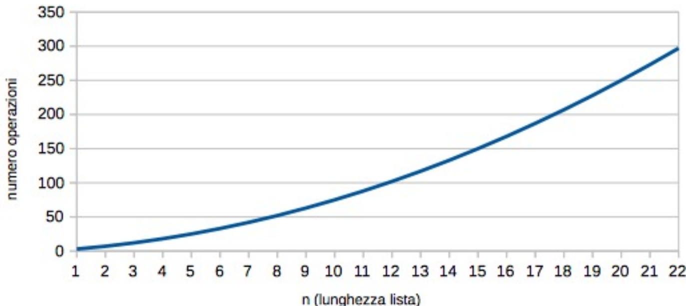
- l'`append` alla fine della lista di un elemento è eseguito $n$ volte
- la `remove` di un elemento è eseguito $n$ volte (assumiamo che ogni rimozione costi 1 operazione)

Quindi in totale abbiamo, in prima approssimazione, un numero totale di operazioni pari a

$$\underbrace{\frac{1}{2}(n^2 + n)}_{\texttt{max_o_min()}} + \underbrace{n}_{\texttt{append}} + \underbrace{n}_{\texttt{remove}} = \frac{1}{2}n^2 + \frac{5}{2}n$$

# Facciamo un pò di conti..

Un numero di operazioni pari a $\frac{1}{2}n^2 + \frac{5}{2}n$ significa che al crescere del numero di elementi della lista, il tempo impiegato per restituire una soluzione cresce in ragione *quadratica*




**Si può fare meglio ?**

# Si può fare meglio ?

Osservazione: unire due liste già ordinate (ad esempio, in modo decrescente) in modo da ottenere una lista ordinata può essere fatto efficientemente

1. chiamiamo le due liste `lista1` e `lista2` (già ordinate in modo decrescente) e `lista_unione` la lista risultante dall'unione
2. inizialmente poniamo `lista_unione` essere vuota
3. confrontiamo fra loro i primi elementi di `lista1` e `lista2`
4. eseguiamo l'`append` dell'elemento più grande a `lista_unione`, e rimuoviamolo dalla propria lista di appartenenza
5. ripetiamo dal punto 3 fino a quando una delle due liste è vuota
6. eseguiamo l'`append` della lista non vuota a `lista_unione`

# Si può fare meglio ?

Quante operazioni abbiamo dovuto eseguire ?

Risposta:

- assumiamo che `lista1` e `lista2` siano entrambe lunghe $m$ (quindi il numero totale di dati sarà $n = 2m$)
- il numero massimo di confronti da eseguire (passo 3) sarà $2m$
- il numero massimo di operazioni di `append` (passo 4) sarà $2m$
- il numero massimo di operazioni di `remove` (passo 4) sarà $2m$
- pertanto, in prima approssimazione, il numero totale di operazioni da eseguire sarà al più

$$\underbrace{2m}_{\text{confronti}} + \underbrace{2m}_{\texttt{append}} + \underbrace{2m}_{\texttt{remove}} = 6m = 3n$$

Quindi riusciamo a fare l'unione (ordinata) in tempo che cresce in ragione *lineare* rispetto al numero di dati!

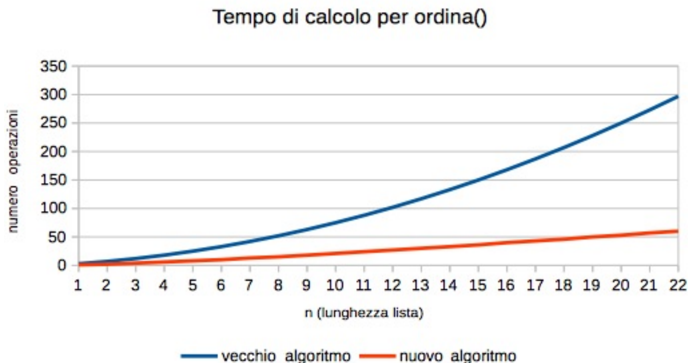
# Nuovo algoritmo!

Idea: ordiniamo gli elementi della lista a coppie e poi uniamo progressivamente le liste già ordinate

Il calcolo del numero di operazioni eseguite è più complesso e quindi vediamo solo il risultato finale: è proporzionale a $n \log(n)$

# Confronto

Il nuovo algoritmo è molto più efficiente!




In gergo si dice che la *complessità algoritmica in tempo* del vecchio algoritmo è maggiore di quella del nuovo algoritmo.

# Difficoltà dei problemi

Si dice che un problema è *difficile* da risolvere se non si conosce un algoritmo che trovi una soluzione in tempo proporzionale ad un polinomio della quantità di dati da elaborare ($n$)

- Pertanto il problema dell'ordinamento è un problema *facile*
- Molti problemi del mondo reale sono problemi difficili: si conoscono solo algoritmi che impiegano tempo esponenziale rispetto ad $n$