Programmazione

Loriano Storchi

loriano@storchi.org

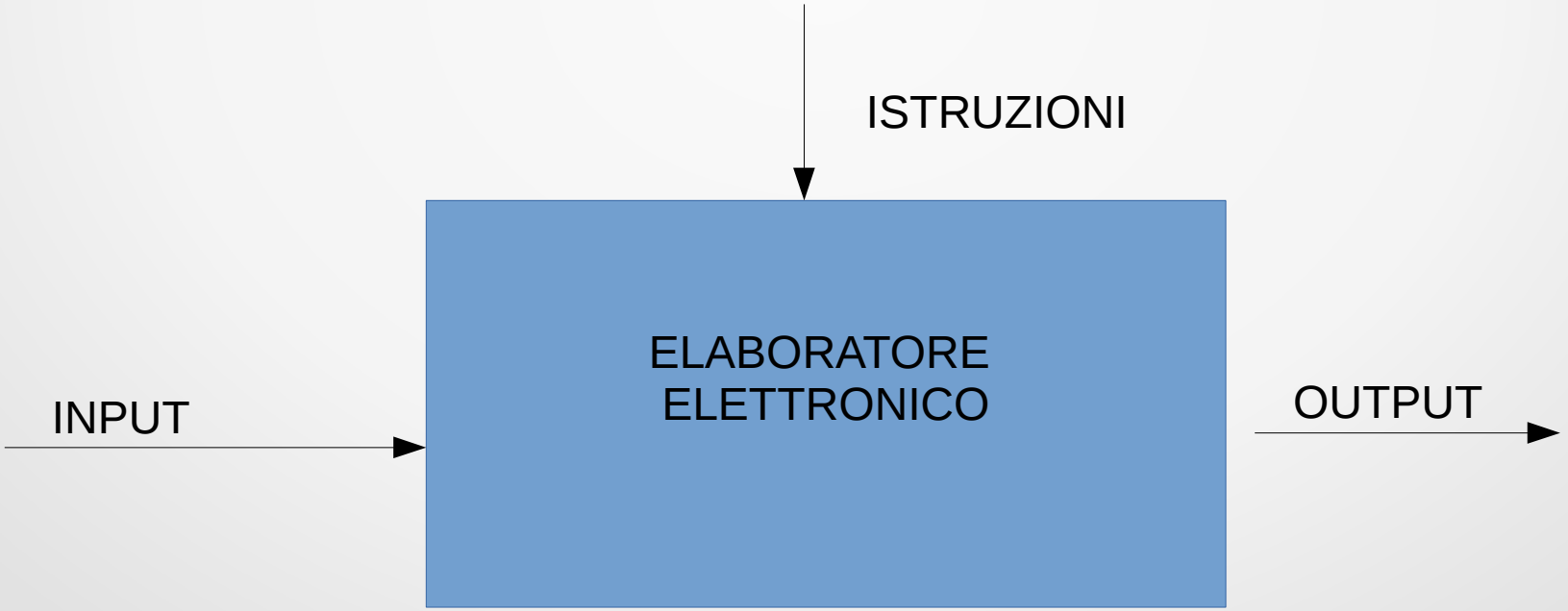
http::://www.storchi.org/

Algoritmi

- Gli algoritmi descrivono il modo in cui trasformare l'informazione. L'informatica si occupa della loro teoria, analisi, progettazione, della loro efficienza realizzazione ed applicazione.
- Un algoritmo è un procedimento formale che risolve un determinato problema attraverso un numero finito di passi. Il termine deriva dalla trascrizione latina del nome del matematico persiano al-Khwarizmi, che è considerato uno dei primi autori ad aver fatto riferimento a questo concetto. L'algoritmo è un concetto fondamentale dell'informatica, anzitutto perché è alla base della nozione teorica di calcolabilità: un problema è calcolabile quando è risolvibile mediante un algoritmo. (Wikipedia)

Programmazione

- Identifica l'attivita' mediante la quale si “istruisce” un calcolatore ed eseguire un particolare insieme di azioni, che agiscono su dati di ingresso (input), allo scopo di risolvere un problema e dunque produrre opportuni dati di uscita (output). Implementazione di un dato algoritmo.



Linguaggi e livelli

Il livello L0 rappresenta il computer reale ed il linguaggio macchina che esso è in grado di eseguire direttamente. Ogni livello superiori rappresenta una macchina astratta. I programmi (istruzioni) di ogni livello superiore devono essere o tradotti in termini di istruzioni di uno dei livelli inferiori, o interpretati da un programma che gira su di una macchina astratta di livello strettamente inferiore.

Linguaggio Macchina e ASSEMBLY

Ogni calcolatore e' in grado di interpretare un linguaggio di basso livello detto linguaggio macchina, le istruzioni (OPCODE) sono semplici sequenze di bit che il processore interpreta e seguendo una serie precisa di operazioni. Ogni istruzione a questo livello e' costituita da operazione estremamente basilari.

Per rendere piu' immediata la programmazioni il primo passo e' stato l'introduzione dell'ASSEMBLY, dove il codice binario e' sostituito da istruzioni mnemoniche umanamente piu' facilmente utilizzabili.

Linguaggio Macchina:

```
00000000000000000000001000000
00000000001000000000001000100
00000010000000010000000000000000
00000001000000000000000000111100
```

PSEUDO-ASSEMBLY:

```
z : INT;
x : INT 8;
y : INT 38;
LOAD R0,x;
LOAD R1,y;
ADD R0,R1;
STORE R0,z;
```

LINGUAGGI

- Oggi sono presenti numerosi linguaggi di programmazione. In generale ogni linguaggio risulta piu' o meno adaguato ad uno scopo specifico.
- **Linguaggi naturali**: sono dato spontaneamente sono estremamnte espressivi ma ambigui: **la vecchia porta la sbarra**
- **Linguaggi artificiali**: sono linguaggi che hanno una data precisa di nascita ed una lista di autori. I linguaggi artificiali possono essere **formali** e non-formali.
- **Linguaggi formali**: non ambigui costituiti da un insieme finito di stringhe costruite a partire da un alfabeto finito. E' un linguaggio per cui la forma delle frasi (**sintassi**) ed il significato dell stesse (**semantica**) sono definite in modo preciso non ambiguo. E' dunque possibile definire una procedura algoritmica in grado di verificare la correttezza **grammaticale** delle frasi.

LINGUAGGI

- Per definire un linguaggio rigorosamente occorrono alcuni strumenti di base:
  - **Alfabeto**: insieme dei simboli di base necessari a costituire le parole
  - **Lessico**: insieme delle regole necessarie a acrivere le parole di un linguaggio (vocabolario)
  - **Sintassi** (regole **grammaticali**): insieme di regole che permettono di stabilire se una frase (insieme di parole) e' corretta
  - **Semantica**: definisce il significato di una “frase” sintatticamente corretta ad esempio: **int a[5];** in linguaggio C permette di riservare spazio in memoria necessario a contenere 5 interi

LINGUAGGI

- Abbiamo gia' accennato a Linguaggi artificiali, Linguaggio macchina e Linguaggio di basso livello (ASSEMBLY)
- Linguaggi di alto livello si allontanano dalla logica del processore e sono costruiti per essere semplici, efficienti e leggibili, oltre che indipendenti dalla macchina
- Sono ad oggi presenti tantissimi linguaggi di programmazione, anche se quelli effettivamente utilizzati sono una decina.
- Di seguito riporteremo una classificazione sommaria di tali linguaggi (C, C++, C#, JAVA, PYTHON, FORTRAN, PASCAL, BASIC, Objectice-C e tantissimi altri). E' chiaro che ogni paradigma di programmazione e' piu' o meno adatto ad uno scopo piu' o meno specifico.

CLASSIFICAZIONE DEI LINGUAGGI

LINGUAGGI IMPERATIVI

- La componente fondamentale del programma e' l'istruzione, ed ogni istruzione indica l'operazione che deve essere eseguita. Le singole istruzioni che operato su i dati del programma.
- Le istruzioni vengono eseguite una dopo l'altra
- Ogni programma e' costituito da due parti fondamentali la dichiarazione dei dati e l'algoritmo inteso come sequenza di operazioni
- Ogni istruzione e' un ordine (programmazione dichiarativa il programma e' una serie di affermazioni)
- Di fatto da un punto di vista sintattico molti linguaggi imperativi utilizzano appunto verbi all'imperativo (i.e. PRINT, READ, ….)

LINGUGGI IMPERATIVI

```
READ *, A, B
C = A + B
PRINT C
```

Quindi una serie di istruzioni leggi A e B, calcola C come somma di A piu’ B ed infine stampa il risultato

PROGRAMMAZIONE PROCEDURALE

- Possiamo considerarlo un sotto-paradigma della programmazione imperativa.
- Viene introdotto il concetto di sotto programma (subroutine) o funzioni.
- Quindi si introduce la possibilita' di creare porzioni di codice sorgente utili ad eseguire funzioni specifiche.
- Questi sottoprogrammi possono ricevere parametri di input e restituire valori di output.

PROGRAMMAZIONE PROCEDURALE

```
SUB EXSUMMA (A, B, C)
    C = A + B
END SUB

FUNCTION SUM (A, B)
    C = A + B
    RETURN C
END FUNCTION
```

```
MAIN
   READ A, B
   PRINT SUM (A,B)
   EXSUM (A,B,C)
   PRINT C
```

Esempio di utilizzo di una subroutine e di una funzione per il calcolo della somma (riusabilita' del codice, librerie di funzioni)

PROGRAMMAZIONE STRUTTURATA

- Possiamo considerarlo un sotto-paradigma della programmazione imperativa.
- In pratica il programmatore e' vincolato ad usare olo strutture di controllo canoniche che non includono le istruzioni di salto incondizionato (GOTO). Dunque la sintassi del linguaggio impedisce l’uso di strutture che non seguono certi vincoli. (non solo)
- L'uso dell'istruzione GOTO porta inevitabilmente ad una scarsa leggibilita' del codice (spaghetti-code)

PROGRAMMAZIONE STRUTTURATA

- Esempio da wikipedia

```
10 dim i
20 i = 0
30 i = i + 1
40 if i <= 10 then goto 70
50 print "Programma terminato."
60 end
70 print i & " al quadrato = " & i * i
80 goto 30
```

```
function square(i)
    square = i * i
end function
dim i
for i = 1 to 10
    print i & " al quadrato = " & square(i)
next
print "Programma terminato."
```

PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

- Possiamo considerarlo un sotto-paradigma della programmazione imperativa.
- Tale paradigma di programmazione permette di definire **Oggetti** Software in grado di interagire gli uni con gli altri.
- L'organizzazione del software sotto forma di oggetti permette un piu' facile riuso dello stesso. Una migliore organizzazione di progetti di grandi dimensioni.
- I Linguaggi OOP prevedono il raggruppamento di parte del codice sorgente in classi, ogni classe comprende dati e metodi (funzioni) che operano sui dati stessi. Le classi sono dei modelli astratti che al momento dell'esecuzione vengono invocate per creare od istanziare oggetti software.
- Un linguaggio orientato agli oggetti permette di implementare tre meccanismi di base utilizzando la sintassi nativa del linguaggio : **incapsulamento, polimorfismo, ereditarieta'**.

PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

ESEMPIO OGGETTI

CLASSE QUADRATO
  ATTRIBUTI
    LATO
    COLORE
  METODI
    REAL OTTIENI_AREA()
    SET_LATO (VAL)

QUADRATO 1
  LATO = 1.0
  COLORE = VERDE

QUADRATO 2
  LATO = 1.5
  COLORE = GIALLO

PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

- **Incapsulamento:** separazione precisa fra implementazione ed interfaccia della classe. Chi usa la classe (oggetto) non deve conoscere il dettaglio implementativo. Utilizza la classe mediante i metodi ed i dati pubblici interagendo con l'oggetto senza appunto consocere il dettaglio dell'implementazione

MAIN

TRIANGOLO T1

T1.COLORE = GRIGIO

T1.SET_LATO(2.0)

PRINT T1.CALCOLA_AREA ()

PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

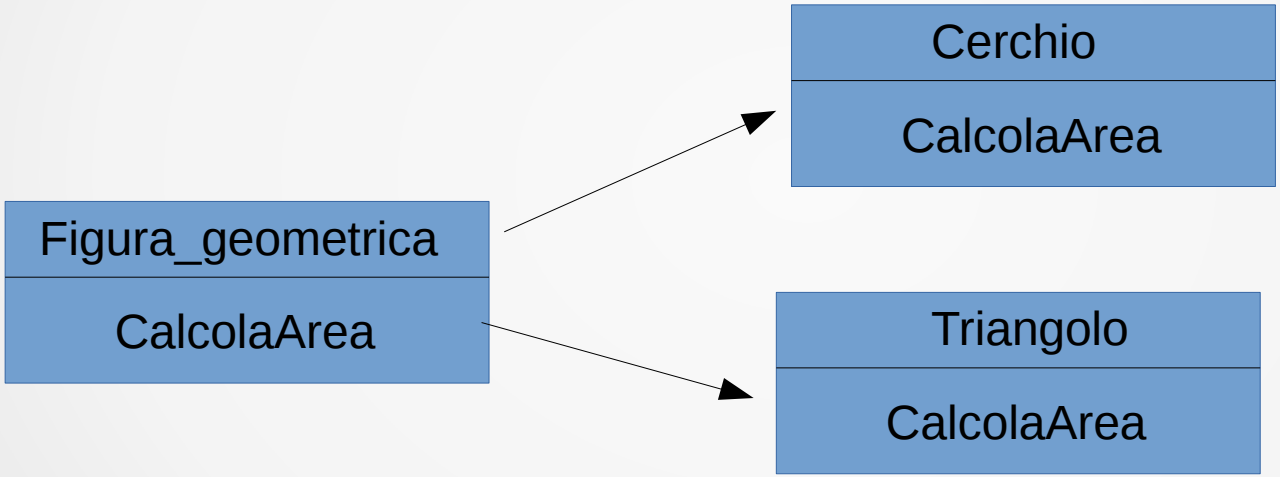
- **Ereditarieta**': una classe puo' ereditare da una classe base ed evolverne o specializzarne le funzionalita'.
- Ad esempio posso immaginare una classe base figura_geometriche da cui classi come trinagolo, cerchio, quadrato … derivano.
- Le classi che derivano da una classe base ereditano tutti i metodi e le proprieta' della classe base, puo' pero' specializzarsi definendo metodi e dati propri.
- Se ad esempio B e' una sottoclasse (oppure piu' in generale un sottotipo) di A, ogni programma/funzione che puo' usare A puo' usare anche B

PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

- **Polimorfismo**: possiamo cercare di esemplificare questo concetto dicendo: molteplici definizione della stessa funzione (overloading), classi e funzioni parametriche rispetto al tipo di dato. Esempio semplice overloading di funzioni.
- Possiamo formalmente distinguere in almeno quattro tipi di polimorfismo: per inclusione, parametrico, overloading, coercion.
- Noi faremo solamente qualche semplice esempio utile a chiarire il concetto generale.

PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

- Immaginiamo la solita classe figura_geometriche da cui avremo fatto derivare due classi cerchio e triangolo



- Quando l'utente richiama il metodo calcola area questo eseguira' una determinata azione (calcolo dell'area nei due casi pur avendo lo stesso nome

PROGRAMMAZIONE FUNZIONALE

- Come dice il nome stesso il flusso di esecuzione assume la forma di valutazione di una serie di valutazioni di funzioni matematiche. Il programma e' quindi un'insieme di funzioni
- Nei linguaggi funzionali puri non esiste il concetto di assegnazione, o di allocazione esplicita della memoria.
- I valori non si trovano cambiando lo stato del programma, non esiste appunto l'assegnazione di valore, ma costruendo il nuovo stato mediante funzioni a partire dallo stato precedente.
- Usanti nell'ambito dell' AI (poco usanti o assenti in ambito industriale)
- Le funzioni possono essere passate come parametri e ritornate come “risultato” da altre funzioni.

LINGUAGGI DICHIARATIVI (O LOGICI)

- Rispetto al paradigma imperativo il programma consiste di una serie di affermazioni e non di ordini.
- Nel programma si specifica il COSA si voglia ottenere non il COME. Il come e' lasciato all'esecutore
- In pratica il programma (o la sua esecuzione) si puo' considerare come la dimostrazione della verita' di un'affermazione.

LINGUAGGI

- Si possono poi classificare i linguaggi anche secondo il tipo di dato **tipizzazione statica** e **tipizzazione dinamica**.
- **Tipizzazione statica :** il programmatore e' costretto a specificare esplicitamente il tipo di ogni elemento sintattico Ad esempio deve specificare il tipo di una variabile ed il linguaggio poi garantira' che quella variabile verra' usata coerentemente con la dichiarazione.
- **Tipizzazione dinamica** : ad esempio in questo caso i dati assumeranno un tipo che varia a runtime in funzione di assegnazioni fatte (vedi Python)
- Possiamo poi distinguere anche fra tipizzazione **debole e forte**

LINGUAGGI

- **Linguaggi o paradigmi di programmazione parallela** (per le moderne architetture) se siete cuirosi potete vedere qui ad esempio http://www.storchi.org/lecturenotes/acr/index.html
- **Linguaggi esoterici** : sono linguaggi voltamente complessi e poco chiari. Popolari solo fra gli utenti piu' abili ed usati al solo scopo di mettere alla prova le capacita' di programmazione (scopo essenzialmente ludico)
- **Scripting**: nati inizialmente per essere usati nelle shell Unix. Sono linguaggi usati per automatizzare compiti ripetitivi e lunghi.

Programmazione altri concetti

- Il sorgente viene scritto in file di testo ASCII.Il sorgente esprime l'algoritmo implementato nel linguaggio scelto. Per scrivere il sorgente si possono usare semplici editor di testo (VI, Emacs). Oppure IDE ambineti di sviluppo integrato con altri strumenti, come compilatori , linker e debugger.
- **Compilazione**: il sorgente viene tradotto (dal compilatore) da linguaggio ad alto livello a codice eseguibile. Il vantaggio e' che l'esecuzione e' “veloce” e che il codice viene ottimizzato per la piattaforma specifica. Lo svantaggio e' che si dovra' ri-compilare per ogni diverso sistema operativo o hardware.
- **Linking**: ogni programma generalmente fa uso di una o piu' librerie ed il linker collega assieme librerie e programma di partenza. I linking puo' essere sia statico che dinamico (ad esempio librerie .so in Linux/Unix-like o .dll in windows)

Programmazione altri concetti

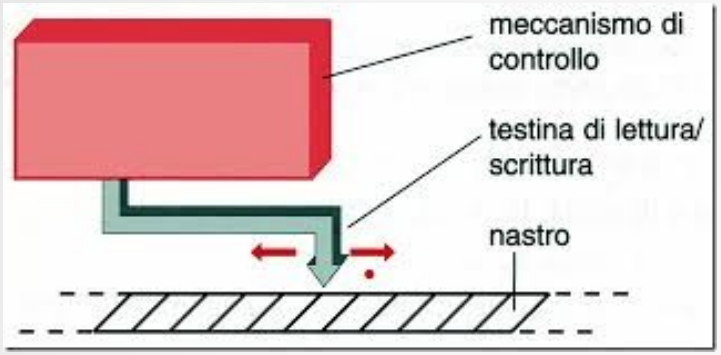
- **Interpretazione**: onde evitare il problema della portabilita' dei programmi si e' ricorso al concetto di interpretate In questo caso il codice sorgente non viene compilato “tradotto” ma eseguito appunto da un'interprete. Questo introduce altri problemi come quello delle prestazioni.
- **Bytecode, P-code**: possiamo definirlo come un approccio intermedio in cui il programma sorgente viene “tradotto” in un codice intermedio che viene interpretato da una macchina virtuale. Questo permette di unire due vantaggi una buona velocita' di esecuzione assieme ad una estrema portabilita' del programma. JIT (Just in Time) al momento dell'esecuzione compilano il codice intermedio in codice macchina.

Programmazione altri concetti

- **Calcolabilita' :** Data una funzione essa e’ detta calcolabile se possiamo trovare un algoritmo (quindi una procedura che esegue meccanicamente un numero finito di passi) che la calcola.
- Tesi di Church-Turing, la classe delle funzioni calcolabili coindicide con la classe delle funzioni calcolabili da una macchina di Turing.
- Tutte le macchine di calcolo (computer) possono essere ricondotte ad una macchina di Turing.

Programmazione altri concetti

- **MdT: macchina di Turing**



Modello deterministico con nastro e istruzioni a 5 campi:

1 - Un nastro lungo a piacere che puo' contenere caratteri oppure spazi vuoti
2 - testina/dispositivo di lettura e e scrittura con cui appunto leggere e scrivere sul nastro ed ovviamente la testina puo' muovere il nastro a desta o sinistra
3 - La macchina ha uno stato interno

La MdT ad ogni passo legge un simbolo ed ed in funzione del suo stato interno puo' cambiare stato e poi scrivere un simbolo nel nastro e poi muovere il nastro a destra o sinitra.
Il comportamento della MdT e' programmato definendo delle regole o quadruple del tipo:

(stato interno, simbolo-letto, nuovo-stato, simbolo-scritto/direzione)

Programmazione altri concetti

- **MdT e problema della terminazione:** dato un certo programma e dato un certo input e' impossibile stabilire se tale programma terminera' o meno. (questo problema e' fortemente legato al teorema della incompletezza di Gödel)

Programmazione altri concetti

- **Complessita' algoritmica :** e' la misura' della difficolta' di un calcolo (algoritmo + input)
- La bonta' di un algoritmo si valuta in funzione del tempo e dello spazio necessario alla sua esecuzione, in genrale quindi in funzione delle risorse richieste.
- Chiaramente il tempo di esecuzione e' funzione del tipo di input oltre che del tipo di hardware utilizzato, non ha quindi senso classificare gli algoritmi in funzione del numero di secondo richiesti alla sua esecuzione.
- Il tempo di calcolo si esprime dunque come il numero di operazioni elementari in funzione della dimensione N dei dati di input

Programmazione altri concetti

- Esempio calcolo dell'efficienza di un algoritmi in cui cerchiamo il minimo **m** all'interno di un insieme di **N** numeri {x1, x2, …, xN}
- Immaginiamo di affrontare il problema come segue:
  - Scelgo x1 come possibile minimo
  - lo confronto con x2, poi x3 e cosi' via
  - Se trovo un xi piu' piccolo continuo i confronti con quello cosi' come fatto con x1
  - Al termine avro' trovato il minimo
- Per fare tutto avro' fatto N confronti quindi l'efficienza dell'algoritmo e' direttamente proporzionale alle dimensioni N dell'input

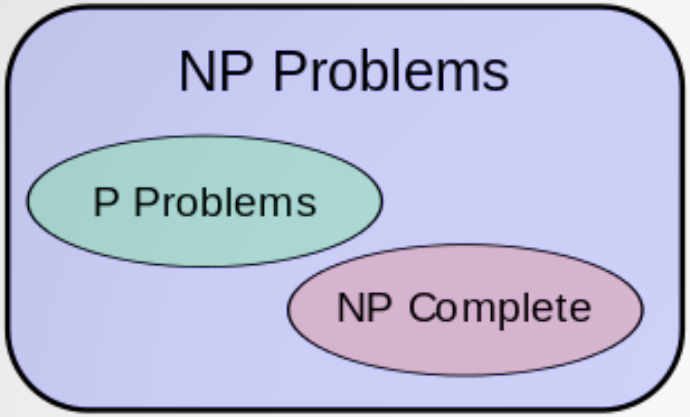
Programmazione altri concetti

- L’efficienza di un dato algoritmo e’ quindi esprimbile come una funzione f(N), quindi funzione della variabile N che rappresenta la dimensione dei dati di input.
  - Tale funzione esprime dunque il numero di operazioni elementari necessarie per risolvere il problema mediante l’algoritmo dato in funzione della dimensione dell’input
  - Rappresenta quindi la complessita’ computazione
  - Dato N un algoritmo A e’ piu’ efficiente di un altro B se al crescere di N $f_A(N)$ s’ minore o uguale ad $f_B(N)$
- Il tempo di esecuzione di un programma quindi dipende dalla complessita’ dell’algoritmo, dalla dimensione di N ed ovviamente dalla “ velocita’ ” della macchina sul quale e’ eseguito

Programmazione altri concetti

- Per suddividere gli algoritmi in classi di complessita' si usa il segunte criterio:
- Una funzione f(N) si dice che e' di ordine g(N) e si indica con **f(N) = O(g(N))** se esiste una costante K tale che , a meno che per un numero finito di valori di N sia sempre vera la seguente disuguaglianza: **f(N) ≤ K*g(N)**. Si puo' anche scrivere **f(n) / g(N) ≤ K**
- Ad esempio 2*N + 5 = O(N) infatti 2*N + 5 ≤ 7*N per ogni N maggiore di zero

Programmazione altri concetti



Classi di complessita’ P e NP , se P sia uguale ad NP o meno ad oggi e’ un problema ancora aperto (problema da un milione di dollari Clay Math Institute)

Classe P problemi risolvibili in una macchina di Turing deterministica in tempi polinomiali.

Classe NP problemi per i quali non e’ noto un algoritmo con complessita’ polinomiale. Ma sono verificabili invece velocemente

Problemi NP-completi il modo piu’ semplice di descriverla se uno dei problemi NP-completi e’ facile allora lo sono tutti visto che posso “convertire” la risoluzione di uno nell’altro. Allo stesso modo se uno e’ difficile sono tutti difficili.

ESEMPIO: la fattorizzazione in numeri primi di un numero intero problema NP (molto probabilmente non e’ NP-completo ancora non possiamo dirlo) dato c trovare i fattori primo a e b tali che a * b = n