Algoritmi e programmi

Marco Alberti

Programmazione e Laboratorio, A.A. 2016-2017

Dipartimento di Matematica e Informatica - Università di Ferrara
Ultima modifica: 30 settembre 2016

Algoritmi

Sommario

Come si arriva dal problema al processo che lo risolve?

- Problemi e algoritmi
  - Funzione caratteristica
  - Macchina di Turing
  - Algoritmo
- Linguaggi di programmazione
  - Sintassi
  - Semantica
  - Linguaggi di alto e basso livello
  - Caratteristiche
  - Linguaggio C

Problema

Situazione iniziale (problematica):

- Alcune informazioni, che ci interessano, sono ignote
- Altre informazioni, probabilmente utili, sono note

Risolvere il problema significa rendere note le informazioni ignote, partendo da quelle conosciute

Esempio

Qual è il massimo comun divisore di 36 e 60?

- Informazioni note: i due numeri 36 e 60.
- Informazione ignota: l'MCD.

Posso risolvere il problema per qualsiasi coppia di numeri?

Funzione caratteristica di un problema

Dato un problema P, siano

- $\mathcal{I}_P$ l'insieme dei possibili input di P, e
- $\mathcal{O}_P$ l'insieme dei possibili output di P.

La funzione caratteristica di P è

$f_P : \mathcal{I}_P \to \mathcal{O}_P$

$f_P(i) =$ l'output $o \in \mathcal{O}_P$ che rappresenta la soluzione di P per l'input $i \in \mathcal{I}_P$

Esempio

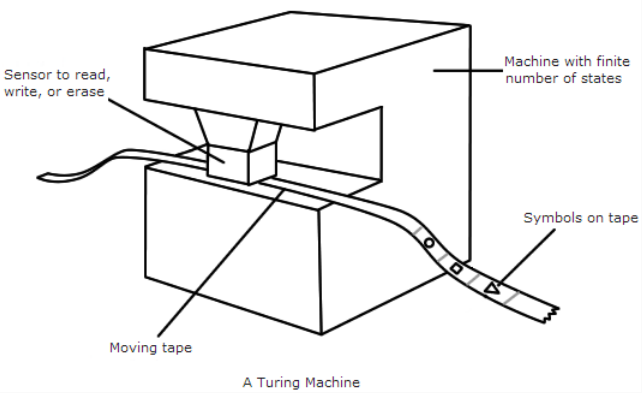
$\mathcal{I}_{MCD} = \mathbb{N}^2$ e $\mathcal{O}_{MCD} = \mathbb{N}$

Risolvere un problema per un certo input equivale a calcolare la sua funzione caratteristica per quell'input.

Macchina di Turing

Modello astratto delle capacità di calcolo di un computer.

Composta da una testina di lettura/scrittura e un nastro infinito mobile.



A Turing Machine

Caratterizzata da:

- Alfabeto di simboli $\mathcal{A}$
- Insieme $\mathcal{S}$ di stati di cui uno iniziale (s_0) e alcuni finali ($\mathcal{F}$).
- Funzione di transizione $\delta : \mathcal{S} \times \mathcal{A} \to \mathcal{S} \times \mathcal{A} \times \{d, 0, s\}$

- Dato lo stato attuale e il simbolo scritto sul nastro, δ determina il prossimo stato, il simbolo da scrivere sul nastro e in quale direzione muovere il nastro
- La macchina parte nello stato s_0 e termina quando lo stato è uno di quelli finali

Tesi di Church-Turing

- Per ogni sequenza iniziale di simboli del nastro, il "passaggio" (se termina) di una macchina di Turing lascerà una sequenza di simboli finale.
- Questo può essere visto come il calcolo di una funzione, che ha come input la configurazione iniziale e come output la configurazione finale.
- La tesi di Church-Turing (comunemente accettata) è che tutte le funzioni calcolabili meccanicamente possano essere calcolate da una macchina di Turing.
- Quindi i problemi che possono essere risolti da un computer sono quelli la cui soluzione è una macchina di Turing: per ogni problema P:w
  P è risolvibile $\leftrightarrow$ f_P è calcolabile $\leftrightarrow$ esiste una macchina di Turing che calcola f_P

Alonzo Church - Alan Turing

- Matematico inglese (1912-1954)
- Macchina di Turing (1936)
- Test di Turing
- Crack del codice Enigma

- Matematico statunitense (1903-1995)
- λ-calculus, fondamento della programmazione funzionale

Macchine di Turing e λ-calculus, proposti in modo indipendente per caratterizzare le funzioni calcolabili algoritmicamente, sono equivalenti.

Simulatore di macchine di Turing

http://morphett.info/turing/turing.html

Esercizi

1. Scrivere una macchina di Turing che incrementi un numero *unario*.
2. Scrivere una macchina di Turing che incrementi un numero binario.

Esecutori

Esprimere la soluzione di un problema come macchina di Turing è

- laborioso: le macchine di Turing richiedono lunghe descrizioni per l'esecuzione di operazioni per noi semplici
- incompleto: ci rimane ancora da tradurre la macchina di Turing in un programma per macchina di Von Neumann

Perciò si preferisce esprimere la soluzione per macchine astratte in grado di eseguire operazioni più complesse dette esecutori. La soluzione in termini comprensibili ed eseguibili da un esecutore si chiama algoritmo.

Se l'esecutore scelto è in grado di emulare una qualsiasi macchina di Turing, è potente come qualsiasi computer e quindi non limitativo per l'espressione di algoritmi.

Algoritmo

Dato un esecutore, un algoritmo è una sequenza di istruzioni

- eseguibili dall’esecutore
- non ambigue
- in numero finito

che, partendo dai dati del problema, giunge alla soluzione.

Esempio di algoritmo: MCD

Dati: a e b, numeri naturali

1. se $a = b$ vai al passo 4
2. se $a < b$ allora assegna a b il valore $b - a$, altrimenti assegna ad a il valore $a - b$
3. vai al passo 1
4. il risultato è a

L'esecutore di questo algoritmo deve saper eseguire

- confronti
- operazioni aritmetiche fra interi (sottrazione)
- salti condizionati e incondizionati
- assegnamenti (cioè impostare e cambiare il valore di una variabile)

Algoritmi equivalenti

Due algoritmi sono equivalenti se

1. Accettano gli stessi input;
2. Restituiscono lo stesso output per ogni input;

cioè se risolvono gli stessi problemi allo stesso modo.

Algoritmi equivalenti possono però differire per numero di passi di calcolo e per memoria utilizzata (efficienza).

Esempio

L'MCD di due numeri si può calcolare anche

- calcolando i divisori dei due numeri e prendendo il più grande di quelli in comune
- moltiplicando i fattori primi comuni ai due numeri, ciascuno preso con il minor esponente

Complessità

- La complessità temporale (risp. spaziale) di un algoritmo è il numero di passi (risp. di celle di memoria) di cui necessita per risolvere un problema, in funzione della dimensione dell'input
- La complessità di un problema è la complessità dell'algoritmo più efficiente che risolve quel problema
- Esistono problemi che possono essere risolti, ma la cui complessità è tale che non si arriverebbe alla soluzione in tempo utile, oppure non possono essere implementati sui computer a disposizione per mancanza di memoria (più raro)
- La complessità temporale si considera accettabile se è polinomiale (ma dipende dalle applicazioni)
- Compromesso spazio/tempo

Linguaggi di programmazione

Sintassi

I linguaggi di programmazione sono linguaggi formali.

La sintassi di un linguaggio formale è un insieme di regole che determina se una qualsiasi sequenza di simboli di un alfabeto è un'espressione del linguaggio.

Formalismi per esprimere sintassi:

- (Extended) Backus - Naur Form (EBNF)
- Diagrammi sintattici

Backus-Naur Form: Numero naturale

⟨*naturale*⟩ ::= 0 | ⟨*cifra non nulla*⟩ { ⟨*cifra*⟩ }

Un numero naturale è 0, oppure ("|") una cifra non nulla seguita da zero o più ("{ . . . }") cifre

⟨*cifra non nulla*⟩ ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

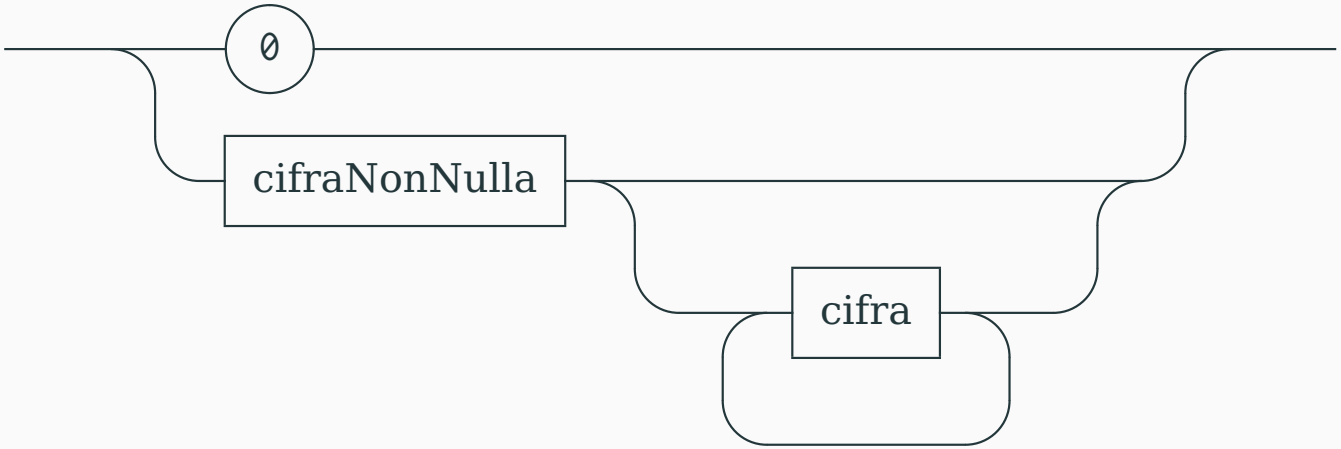
Una cifra non nulla è 1, oppure 2, . . . , oppure 9

⟨*cifra*⟩ ::= 0 | ⟨*cifra non nulla*⟩

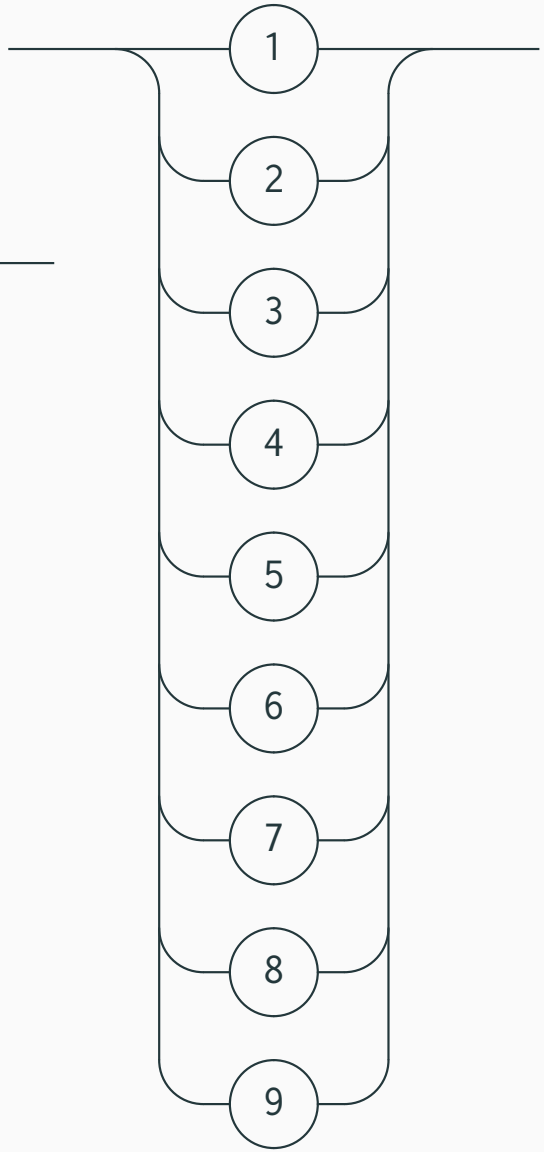
Una cifra è 0 oppure una cifra non nulla

Diagrammi sintattici: Numero naturale

naturale



cifra



cifraNonNulla

Esercizio

Quali dei seguenti sono riconosciuti dalla grammatica?

- "0"
- "23"
- "023"
- "ventitre"
- "3.0"

Semantica

Definisce il significato di ogni programma nel linguaggio di programmazione considerato.

Possibili semantiche:

- Operazionale: definisce quali sono le operazioni eseguite dall’esecutore per ogni programma
- Denotazionale: definisce il significato del programma come funzione
- Assiomatica: definisce proprietà logiche che legano l’input, il programma e l’output

Algoritmi e programmi

- La semantica di un linguaggio di programmazione definisce un esecutore.
- Un programma in un linguaggio di programmazione è un algoritmo per l'esecutore definito dalla semantica del linguaggio di programmazione.
- Normalmente gli algoritmi sono espressi per un esecutore astratto la cui semantica è facilmente riproducibile nei più diffusi linguaggi di programmazione imperativi, per facilitare il passaggio da algoritmo a programma.

Turing Completeness

Un linguaggio di programmazione è Turing-complete se la sua semantica permette di implementare una qualsiasi macchina di Turing.

Un linguaggio Turing-complete può essere usato per risolvere qualsiasi problema che ammetta soluzione. Quasi tutti i linguaggi di programmazione sono Turing complete.

Ciò non significa che siano equivalenti in termini pratici.

Halting problem

Non esiste un algoritmo in grado di determinare se un programma in un linguaggio Turing-complete terminerà per qualsiasi input (halting problem).

Esistono linguaggi appositamente non-Turing-complete, per problemi che non richiedono tanta potenza, per i cui programmi si può dimostrare la terminazione.

Processo

Processo è la sequenza delle operazioni eseguite da una macchina fisica o astratta per eseguire un programma o una parte di esso.

Da non confondere (anche se correlato) con i processi di sistema operativo.

Livello dei linguaggi di programmazione

Un linguaggio di programmazione è di livello tanto più alto quanto più è vicino a quello del dominio dell'applicazione.

Esempio

Il seguente codice in linguaggio C

```
c = a + b;
```

è di livello più alto (in quanto più vicino alla notazione matematica) rispetto al corrispondente codice Assembly x86:

```
mov     edx, DWORD PTR [rbp-8]
mov     eax, DWORD PTR [rbp-12]
add     eax, edx
mov     DWORD PTR [rbp-4], eax
```

Livello più alto

Linguaggio Prolog (usato per applicazioni di rappresentazione della conoscenza e ragionamento automatico)

Il programma descrive il dominio in termini logici (`:-` significa se):

```
antenato(X,Y) :- genitore(X,Y).
antenato(X,Y) :- genitore(X,Z), antenato(Z,Y).

genitore(alice, bruno).
genitore(bruno, carlo).
```

Alla domanda `?- antenato(X,carlo)` il sistema fornisce le due risposte `X=alice` e `X=bruno`.

Alto livello è meglio?

Vantaggi dei linguaggi di alto livello:

- possono portare a riduzione costi e tempi di realizzazione delle soluzioni
- programmi leggibili anche da non esperti di programmazione

Svantaggi:

- richiedono semantiche più complesse per colmare il divario fra il dominio dell'applicazione e la macchina
- non sempre consentono il controllo desiderato sul processo (possono risultare meno efficienti)
- difficile individuare linguaggi che siano davvero vicini ai domini applicativi e allo stesso tempo computabili

Risposta: dipende dall'applicazione.

Caratteristiche di linguaggi di programmazione

- Tipizzazione statica/dinamica
- Gestione automatica /manuale della memoria (garbage collection)
- Immutabilità
- Impostazione procedurale / dichiarativa (funzionale, logica)
- Organizzazione del codice strutturata / orientata agli oggetti

Linguaggio C

- Statico
- Non garbage collected
- Mutabile
- Procedurale
- Strutturato

Considerato di basso livello fra i linguaggi di alto livello.

Ottimo per applicazioni che richiedono controllo del processo (software di sistema, calcolo numerico, multimedia fra le tante).

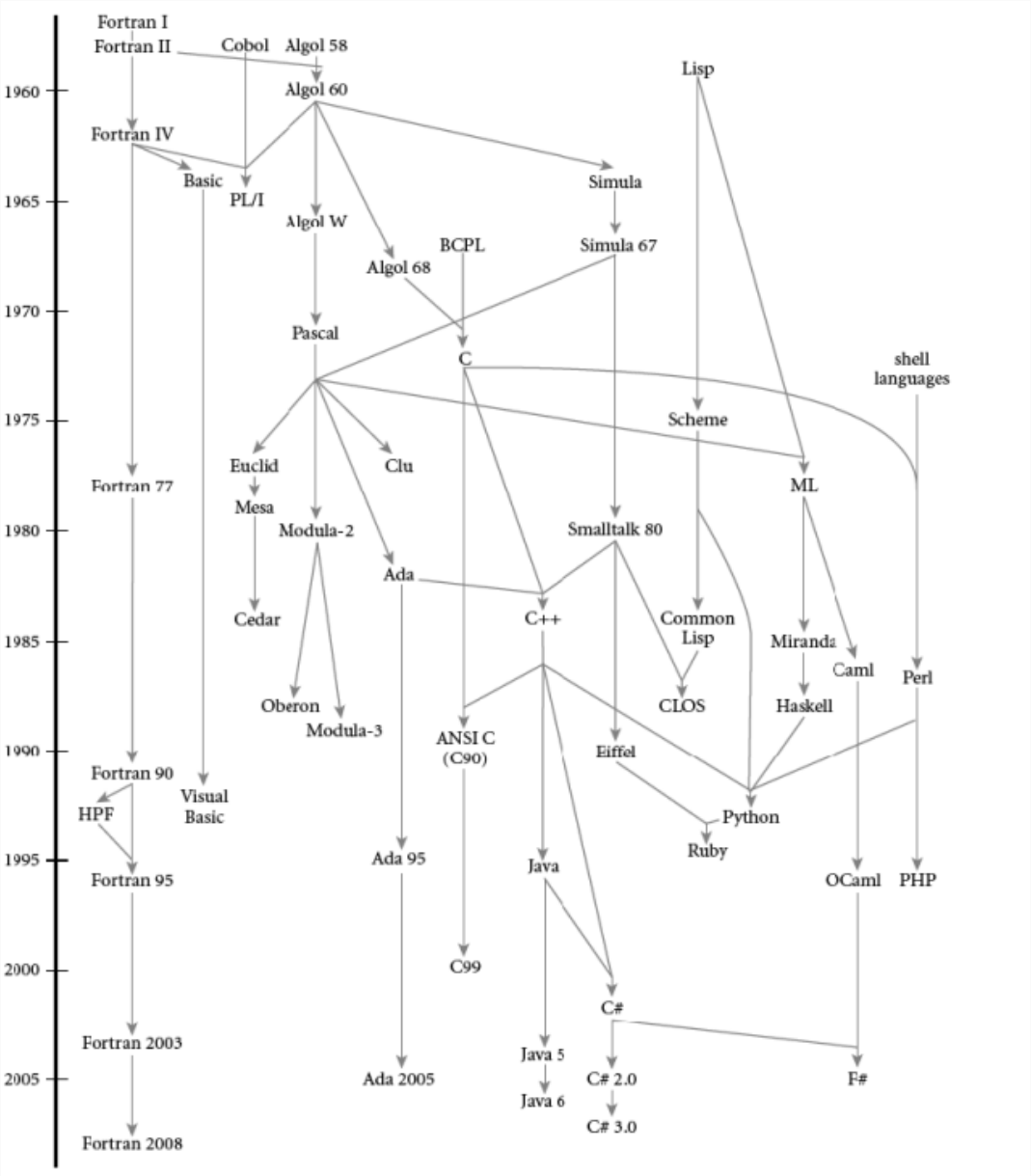
Tra i linguaggi più usati (insieme a Java).

Disponibile per praticamente qualsiasi piattaforma hardware e *portabile* se si segue lo standard ANSI C.

Storia del linguaggio C

1972	Dennis Ritchie implementa il linguaggio C presso i Laboratori Bell per la programmazione del sistema operativo UNIX
1978	Dopo qualche anno di evoluzione, il libro "Il linguaggio di programmazione C" di Kernighan e Ritchie definisce il C tradizionale (K&R C).
1989	Approvato lo standard ANSI C, indipendente dalla macchina. Esce una versione del testo K&R aggiornata allo standard.
1999	Standard C99
2011	Standard C11

Albero genealogico (parziale)



Realizzazione (implementazione) di una macchina astratta

Ogni macchina di Von Neumann è in grado di capire solo il suo linguaggio macchina.

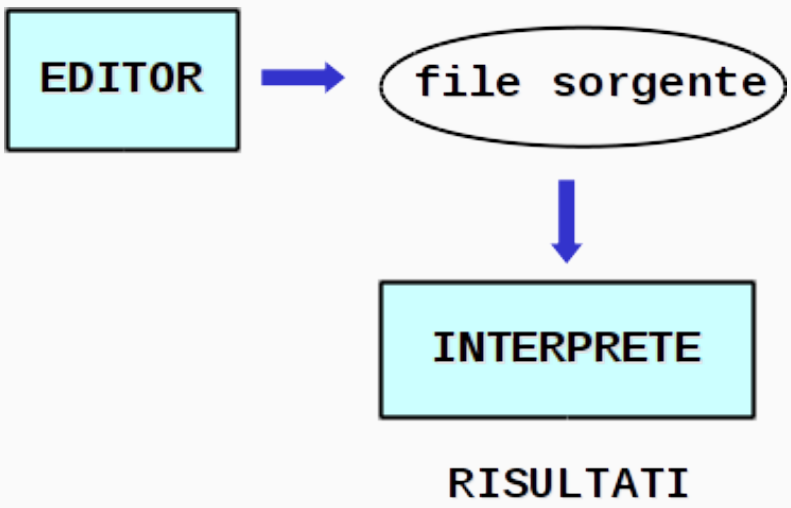
Quindi le istruzioni in codice sorgente del linguaggio di programmazione scelto devono essere trasformate in codice macchina.

Approcci:

1. Interpretazione
2. Compilazione
3. Metodi "ibridi"

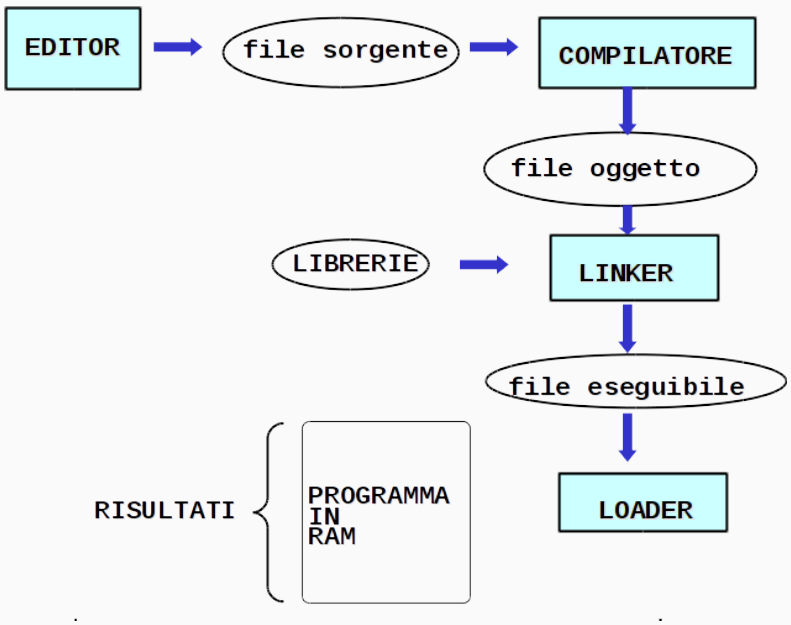
L’approccio non è legato al linguaggio ma alla sua intepretazione (lo stesso linguaggio può avere implementazioni interpretate o compilate).

Interpretazione



- Programma salvato come file nel codice sorgente.
- Caricato come codice sorgente in memoria
- Le istruzioni sono decodificate ed eseguite una alla volta

Compilazione



- Programma salvato come codice sorgente.
- Compilato (tradotto in codice macchina)
- Linking statico o dinamico
- Caricato in RAM (insieme alle librerie in caso di linking dinamico) come codice macchina
- Eseguito direttamente

Metodi "ibridi"

Bytecode: formato "intermedio" fra sorgente e codice macchina: si può vedere come

- codice macchina per una macchina astratta
- codice sorgente più facile da interpretare

Possibili utilizzi:

- Compilazione in bytecode (Ahead Of Time), caricamento bytecode, interpretazione (Visual Basic 6)
- Compilazione in bytecode AOT, caricamento bytecode, interpretazione o compilazione Just In Time in codice macchina (Java Virtual Machine, Common Language Runtime di .NET)
- Caricamento del sorgente, compilazione in bytecode, interpretazione o compilazione JIT (Python)

Conclusione: come programmare?

1. Identificare il problema, cioè l’insieme degli input e le proprietà desiderate degli output
2. Trovare un algoritmo, espresso per un esecutore adeguato (cioè indipendente dal linguaggio di programmazione, ma con funzionalità facilmente realizzabili nel linguaggio di programmazione prescelto)
3. Convincersi della correttezza e dell’efficienza dell’algoritmo
4. Codificare l’algoritmo nel linguaggio di programmazione
5. Se necessario, tornare a un passo precedente

Appendice: come non programmare

1. Copiare e incollare un programma che risolveva un problema grossomodo simile
2. Modificarlo a tentativi finché non sembra che funzioni (almeno per qualche input)