

Algoritmi e programmi

Marco Alberti



Dipartimento
di Matematica
e Informatica



Università
degli Studi
di Ferrara

Programmazione e Laboratorio, A.A. 2020-2021

Ultima modifica: 16 novembre 2020

Attenzione! Questo materiale didattico è per uso personale dello studente ed è coperto da copyright.
Ne sono vietati la riproduzione e il riutilizzo anche parziale, ai sensi e per gli effetti della legge sul diritto d'autore.

Sommario

1 Algoritmi

2 Linguaggi di programmazione

3 La pratica della programmazione strutturata

Sommario

1 Algoritmi

2 Linguaggi di programmazione

3 La pratica della programmazione strutturata

Problema

Situazione iniziale (problematica):

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

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

Esempio

Qual è il massimo comun divisore di 36 e 60? **INPUT**

- Informazioni note: i due numeri 36 e 60.
OUTPUT
- Informazione ignota: l'MCD. **12**

Posso risolvere il problema per qualsiasi coppia di numeri?

Funzione caratteristica di un problema

$MCD(a, b)$

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

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

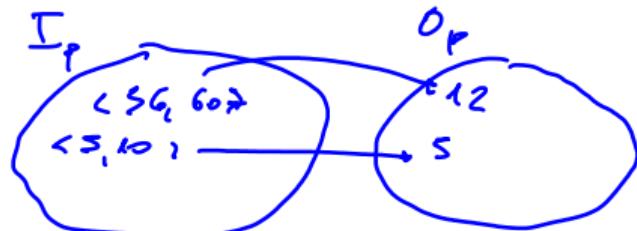
$f_P(i) =$ l'output $o \in \mathcal{O}_P$ che rappresenta la soluzione di P per l'input $i \in \mathcal{I}_P$

Esempio

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

$\mathbb{N} \times \mathbb{N}$

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



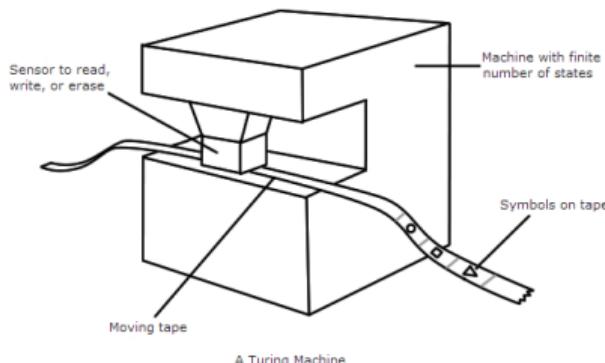
Macchina di Turing

$$s_0 \xrightarrow{\delta} s_s \xrightarrow{\delta} s_r \xrightarrow{\delta} s_{t_0}$$

Modello astratto delle capacità di calcolo di un computer.

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

Caratterizzata da:



- Alfabeto di simboli \mathcal{A} $\{0,1\}$ $\{a,b,c,\dots,e\}$

- Insieme \mathcal{S} di stati di cui uno iniziale (s_0) e alcuni finali (\mathcal{F}). $s_1, s_2, \dots, \textcircled{s}_{t_0}, s_{t_1}, \dots,$

- Funzione di transizione

$$\delta : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S} \times \mathcal{A} \times \{d, 0, s\}$$

↗ **TRANSIZIONE** $\overset{\text{STATO}}{\delta_{\text{STATO}}}$ $\overset{\text{SIMBOLO}}{\delta_{\text{SIMBOLO}}}$ $\overset{\text{NUOVO STATO}}{\delta_{\text{NUOVO STATO}}}$ $\overset{\text{NUOVO SIMBOLO}}{\delta_{\text{NUOVO SIMBOLO}}}$ **DIREZIONE**

- 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

INPUT

- 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
 P è risolvibile $\leftrightarrow f_P$ è calcolabile \leftrightarrow esiste una macchina di Turing che calcola f_P

OUTPUT

Alonzo Church - Alan Turing



- Matematico statunitense (1903-1995)
- λ -calculus, fondamento della programmazione funzionale
- Matematico inglese (1912-1954)
- Macchina di Turing (1936).
- Test di Turing [Int Art](#)
- Crack del codice Enigma

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

Esercizio

Un numero unario (o in base 1) è una sequenza di un numero di cifre 1 pari al numero stesso. Ad esempio, 5 in base 1 è 11111.

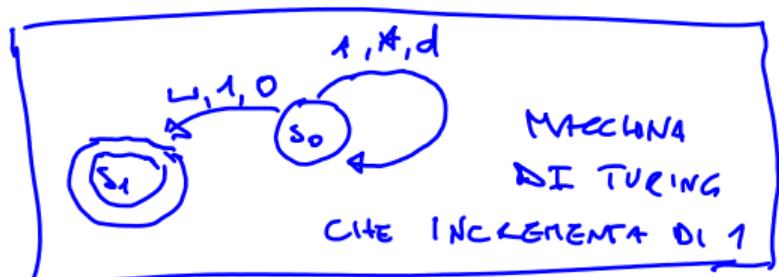
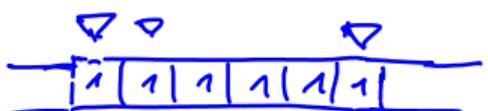
Scrivere una macchina di Turing che incrementi un numero unario.

$$I_p = N \quad O_p = N \quad f_p(11111) = 11111$$

$$f_p(0) = 1$$

$$f_p(1) = 2$$

$$f_p(100) = 101$$



0 1 * r 0

Simulatore di macchine di Turing

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

Esecutori o macchine astratte

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
- nella pratica, non sufficiente: ci rimane ancora da tradurre la macchina di Turing in un programma per macchina di Von Neumann se vogliamo eseguirla su un computer

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

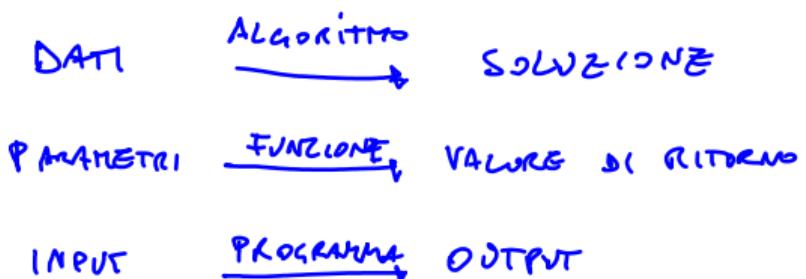
Se l'esecutore scelto è in grado di emulare una qualsiasi macchina di Turing, è potente come qualsiasi computer e quindi non limitativo per l'espressione di algoritmi.

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(36, 60) → 12

INPUT

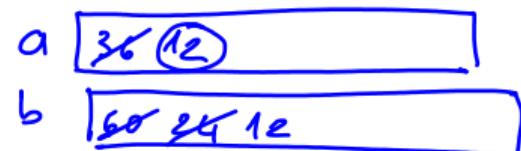
Dati: a e b , numeri naturali

GOTO

- 1 se $a = b$ vai al passo 4 confronto
- 2 se $a < b$ allora assegna a b il valore $b - a$, altrimenti assegna ad a il valore $a - b$ calcolo
- 3 vai al passo 1
- 4 il risultato è a return a,
printf("%d\n", 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

quest'anno 070: $I_{\text{algho}} = \mathbb{N} \times \mathbb{R}$

$$I_{\text{algtre}} = \mathbb{R} \times \mathbb{N}$$

Due algoritmi sono equivalenti se

- ① Accettano gli stessi input; I_p
- ② Restituiscono lo stesso output per ogni input; $\forall i \in I_p \quad a_1(i) = a_2(i)$

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

$$\text{DIV}_{36} = \{1, 2, 3, 4, 6, 9, 12, 18, 36\} \quad \text{max}(\text{DIV}_{36}) = 36$$

$$\text{DIV}_{60} = \{1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60\} \quad \text{max}(\text{DIV}_{60}) = 60$$

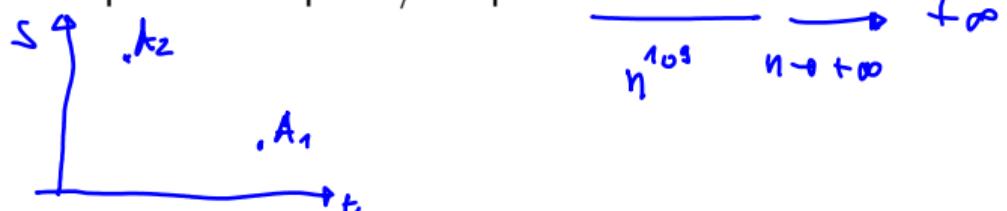
$$36 = 2^2 \cdot 3^2$$

$$60 = 2^2 \cdot 3 \cdot 5$$

$$2^2 \cdot 3^1 = 12$$

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
 n $k n^2 + \dots$ $O(n^2)$ $O(n \log n)$
- 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



Sommario

1 Algoritmi

2 Linguaggi di programmazione

3 La pratica della programmazione strutturata

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.

Ascii



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

- 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

SOLUZIONE → MACCHINA TURING → PROGRAMMA

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

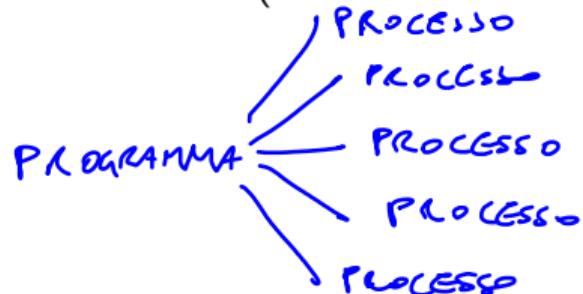
Non esiste un algoritmo in grado di determinare se un programma in un linguaggio Turing-complete terminerà per qualsiasi input (halting problem).

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`.



INCognita

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

Risposta: dipende dall'applicazione.

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

Classificazione di linguaggi di programmazione

Coordinata	Linguaggio C
Tipizzazione statica/dinamica	Statica
Gestione automatica /manuale della memoria (garbage collection)	Non garbage collected
Assegnamento distruttivo	Sì
Impostazione procedurale / dichiarativa (funzionale, logica)	Procedurale.
Organizzazione dei programmi orientata agli oggetti	Non orientato agli oggetti <i>C++</i>

Linguaggio C

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.

Breve storia:

1972 Dennis Ritchie implementa il linguaggio C presso i Laboratori Bell per la programmazione del sistema operativo UNIX

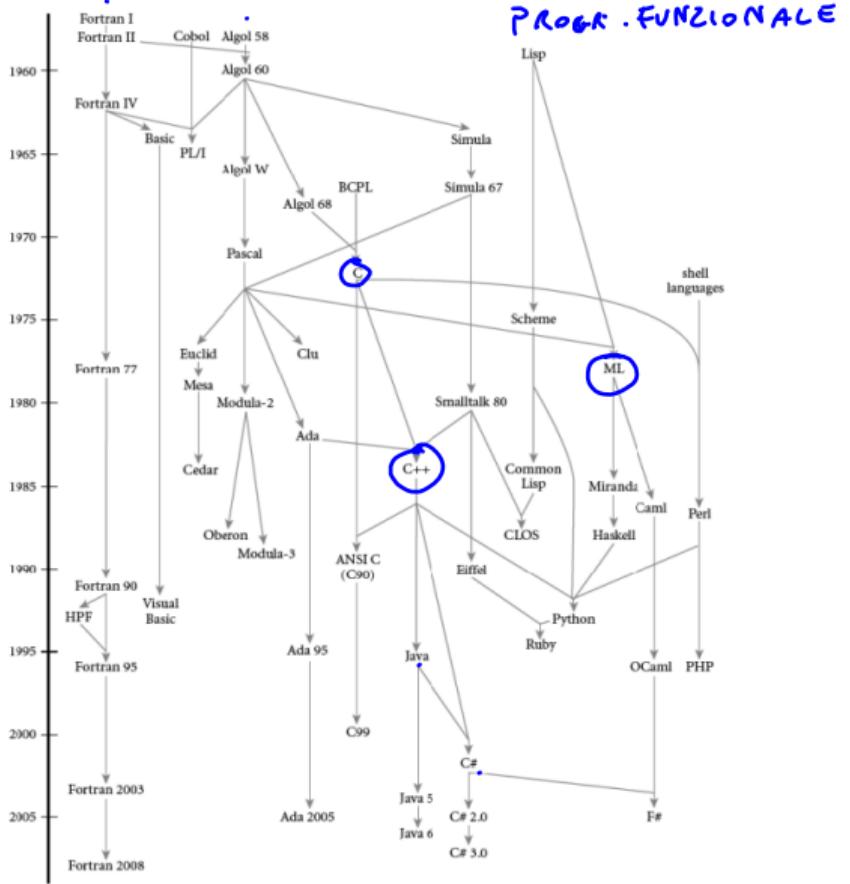
1978 Dopo qualche anno di evoluzione, il libro "Il linguaggio di programmazione C" di Kernighan e Ritchie definisce il C tradizionale (K&R C).

1989 Approvato lo standard ANSI C, indipendente dalla macchina. Esce una versione del testo K&R aggiornata allo standard. 2^a edizione

1999 Standard C99

2011 Standard C11

Albero genealogico (parziale)



Realizzazione (implementazione) di una macchina astratta

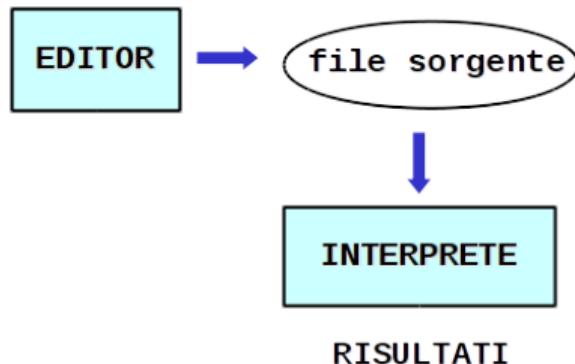
Ogni macchina di Von Neumann è in grado di capire solo il suo linguaggio macchina.
Quindi le istruzioni in codice sorgente del linguaggio di programmazione scelto devono essere trasformate in codice macchina.

Approcci:

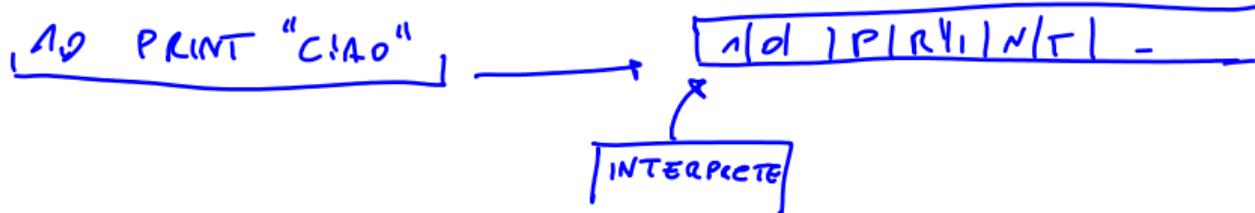
- ① Interpretazione **BASIC**
- ② Compilazione **C**
- ③ Metodi "ibridi"

L'approccio non è legato al linguaggio ma alla sua implementazione (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.c → a.out
gcc programma.c

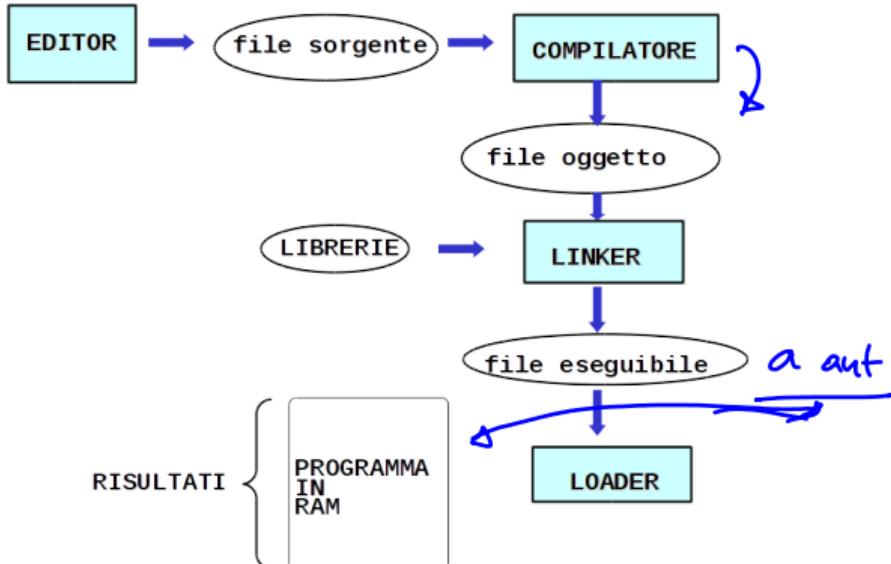
main() {

int a;

a=2;

printf("%d", a);

}



- 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



GCC: compilazione e linking separati

```
gcc -c __nome__.c
```

compila (ma non linka) il file sorgente C `__nome__.c`, generando il file **oggetto** `__nome__.o`

```
gcc __nome__.o -l__lib__
```

linka il file oggetto `__nome__.o` con la libreria `lib__lib__`, generando il file eseguibile `a.out` (è possibile specificare il nome `__eseguibile__` con l'opzione `-o __eseguibile__`).

Esempio

```
gcc -c hello.c produce hello.o
```

```
gcc -o hello hello.o -lc produce hello, unendo il codice binario di hello.o a  
quello della libreria standard C (nota: spesso -lc si può omettere).
```

L'utilità di compilazione e linking separati sarà più chiara quando scriveremo programmi composti da più file sorgente.

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:

SORGENTE → BYTCode

- 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

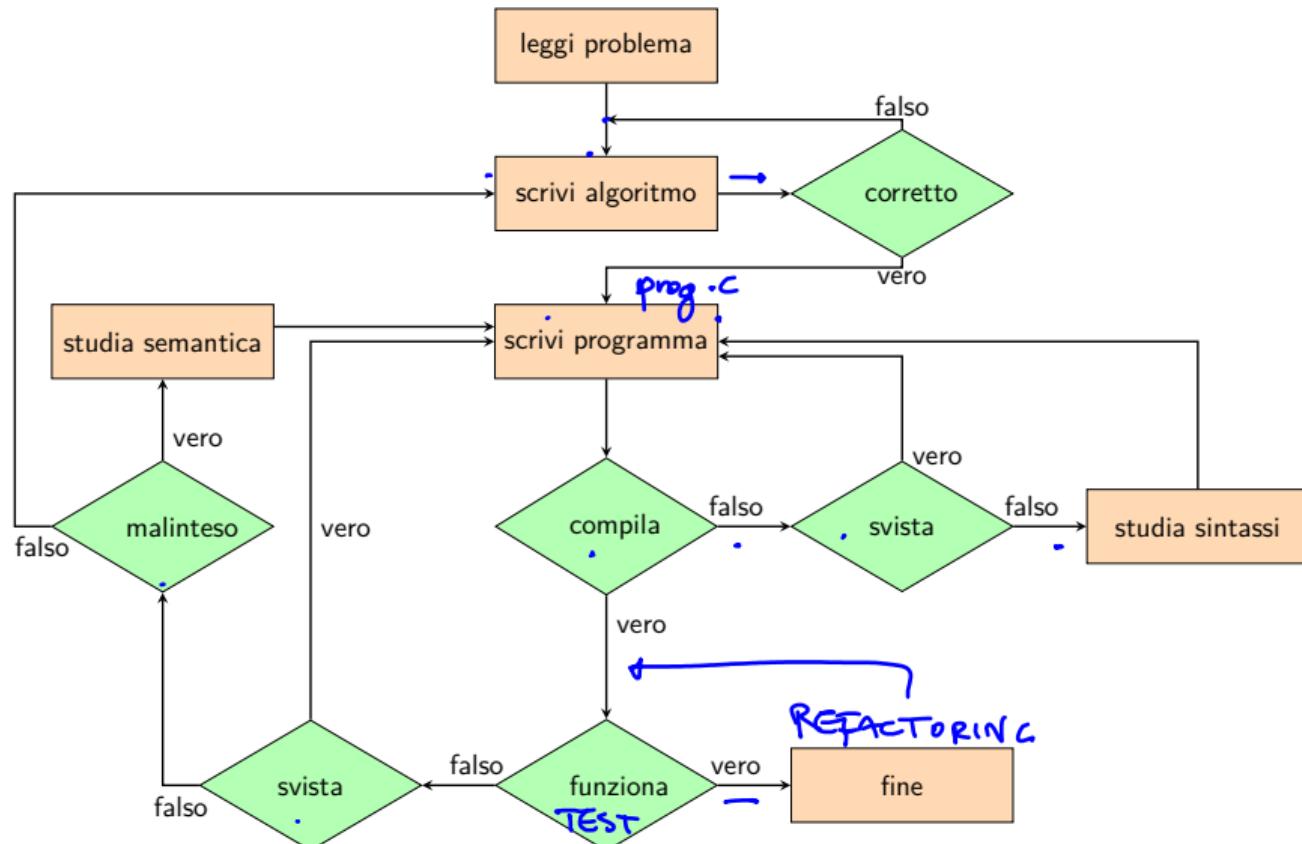
Sommario

1 Algoritmi

2 Linguaggi di programmazione

3 La pratica della programmazione strutturata

Procedimento di programmazione



Primitive, Composizione, Astrazione

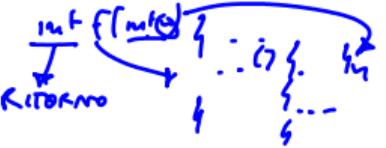
Ogni linguaggio di programmazione fornisce, per ognuno degli aspetti di un programma che consente di esprimere,

- primitive: gli elementi fondamentali
- strumenti di composizione: per costruire elementi più complessi a partire da elementi più semplici
- strumenti di astrazione: per usare un elemento complesso come se fosse uno semplice

	Calcolo	Controllo	Dati
Primitive	costanti, variabili 5	istruzione semplice	scalari
Composizione	operatori $S+6*(a+7)$	blocco, selezione, ciclo, salto	tipi strutturati (array, struct)
Astrazione	funzioni	funzioni	typedef

```
quad(3)      int quad(int n){  
    return n*n;  
}  
void stampa(float f){  
    while(..)  
        Stampa(3.5);  
}
```

Astrazione procedurale FUNZIONI



- Il corpo di una funzione è un blocco di codice, potenzialmente molto complesso (può contenere ogni tipo di struttura di controllo, chiamate ad altre funzioni, etc.).
- Definire una funzione significa individuare una funzionalità di calcolo, darle un nome e stabilire da cosa dipende (parametri) e che cosa produce (valore ed effetto).
- Come per le espressioni, caratterizzare una funzione con valore ed effetto di qualsiasi chiamata è l'unica cosa che serve sapere per usarla. Si può dimenticare (**astrazione**) le istruzioni che la compongono (**procedurale**).
- Ad esempio, definita la funzione **potenza** come in precedenza, possiamo usarla secondo la nostra intuizione matematica senza dover pensare che è implementata con un ciclo.

$\text{int potenza(int base, int esponente)}$
 $\text{potenza}(3,5) \rightarrow 3^5$

L'astrazione procedurale nella pratica

int f (int u) {
 ...
}

main () {
 f(...);
}



- Una volta stabilite interfaccia, valore ed effetto di una funzione, l'implementazione può essere modificata (per correzione di bug, miglioramento di efficienza) senza conseguenze per chi usa la funzione.
- Il programmatore ha meno aspetti da tenere presente contemporaneamente: sviluppare la funzione e usarla sono fasi diverse.
- Senza astrazione procedurale, esprimere funzionalità di calcolo non banali può rapidamente diventare ingestibile.
- L'astrazione procedurale però è un processo non banale, che richiede intuito ed esperienza, ed è a sua volta soggetto ad errori.

top-down e bottom-up

Ma come si decide come strutturare il programma in funzioni? Ci sono due metodi:

- ↓ • top-down: decidere a priori quali funzioni sono necessarie;
- ↑ • bottom-up: individuare le funzionalità che si ripetono durante la scrittura del programma, e separarle dal programma principale isolandole in funzioni.

Esempio

Voglio scrivere un programma che richieda all'utente due numeri interi a e b e stampi a video il numero intero $a^b - b^a$.

3 2

INPUT

3 2

OUTPUT

$$3^2 - 2^3 = 1$$

Procedimento top-down: fase 1

Riconosco che il problema richiede due volte il calcolo della potenza; non so ancora come farlo, ma so che interfaccia deve avere (int potenza (int base, int esponente)) e come usarlo:

080_algoritmi_e_programmi/diff-potenze-finale.c

```
1 int main() {
2     int a, b;
3     scanf("%d%d", &a, &b);
4     printf("%d\n", potenza(a, b) - potenza(b, a));
5     return 0;
6 }
```

Procedimento top-down: fase 2

Ora che ho stabilito che cosa deve fare, scrivo la funzione **potenza**:

080_algoritmi_e_programmi/diff-potenze-finale.c

```
1 int potenza(int base, int esp) {  
2     int cont, prod = 1;  
3     for (cont = 0; cont < esp; cont++)  
4         prod *= base;  
5     return prod;  
6 }
```

2 3

ACCUMULATORE

Procedimento bottom-up: fase 1

Comincio a scrivere il codice per calcolare a^b :

080_algoritmi_e_programmi/diff-potenze-1.c

```
1 #include <stdio.h>
2
3 int main() {
4     int a, b, p1 = 1, cont;
5     scanf("%d%d", &a, &b);
6     for (cont = 0; cont < b; cont++)
7         p1 *= a;    P1 → Pb
8     }
9 }
```

Ora però mi rendo conto che per calcolare b^a dovrei riscrivere lo stesso codice, solo scambiando i ruoli di a e b : decido di operare un'astrazione procedurale delegando il calcolo delle potenze a una funzione apposita, che chiamerò due volte dal **main**.

Procedimento bottom-up: fase 2

Ho separato il calcolo della potenza dall'algoritmo principale.

080_algoritmi_e_programmi/diff-potenze-finale.c

```
1 #include <stdio.h>
2
3 int potenza(int base, int esp) {
4     int cont, prod = 1;
5     for (cont = 0; cont < esp; cont++)
6         prod *= base;
7     return prod;
8 }
9
10 int main() {
11     int a, b;
12     scanf("%d%d", &a, &b);
13     printf("%d\n", potenza(a, b) - potenza(b, a));
14     return 0;
15 }
```

Meglio top-down o bottom-up?

- Il risultato finale è lo stesso.
- Se si riesce immediatamente a individuare le funzionalità di calcolo condivise, il procedimento top-down è più diretto e porta a programmi più leggibili e strutturati più chiaramente.
- Ma la presenza di funzionalità di calcolo ripetute in diverse parti dell'algoritmo non è sempre evidente: in questo caso, il procedimento top-down può portare ad astrazioni errate, che complicano il codice anziché semplificarlo. In questo caso, è meglio procedere bottom-up, a patto di non fermarsi alla prima versione funzionante del programma.
- Quasi sempre lo sviluppo di programmi non banali comporta l'uso di entrambi i metodi.

Dichiarazioni di funzioni

- Spesso la lettura dei programmi strutturati in molte funzioni sarebbe più immediata se la funzione **main**, che contiene l'algoritmo principale, comparisse per prima.
- Nella slide 38 ci limitassimo a spostare la funzione **potenza** dopo la funzione **main**, il compilatore rifiuterebbe il programma, perché **main** conterebbe una chiamata di una funzione di prototipo sconosciuto, di cui il compilatore non potrebbe verificare la correttezza.
- E' però possibile premettere al **main** solo la **dichiarazione** (cioè il protototipo seguito da **;**) della funzione, lasciando la definizione dopo il **main**: in questo modo il compilatore può verificare la correttezza della chiamata.
- Ad esempio, **<stdio.h>** ^{usr/include/stdio.h} contiene, fra le altre cose, i prototipi delle funzioni **printf** e **scanf**; è per questo che è necessario includerlo per usare queste funzioni.
- I prototipi aiutano anche la lettura del programma, perché chiariscono il tipo dei parametri e del valore di ritorno.

Esempio di dichiarazione e definizione di funzione

080_algoritmi_e_programmi/diff-potenze-prototipi.c

```
1 #include <stdio.h>
2
3 int potenza(int base, int esp); DICHIARAZIONE
4
5 int main() {
6     int a, b;
7     scanf("%d%d", &a, &b);
8     printf("%d\n", potenza(a, b) - potenza(b, a));
9     return 0;
10 }
11
12 int potenza(int base, int esp) {
13     int cont, prod = 1; DEFINIZIONE
14     for (cont = 0; cont < esp; cont++)
15         prod *= base;
16     return prod;
17 }
```