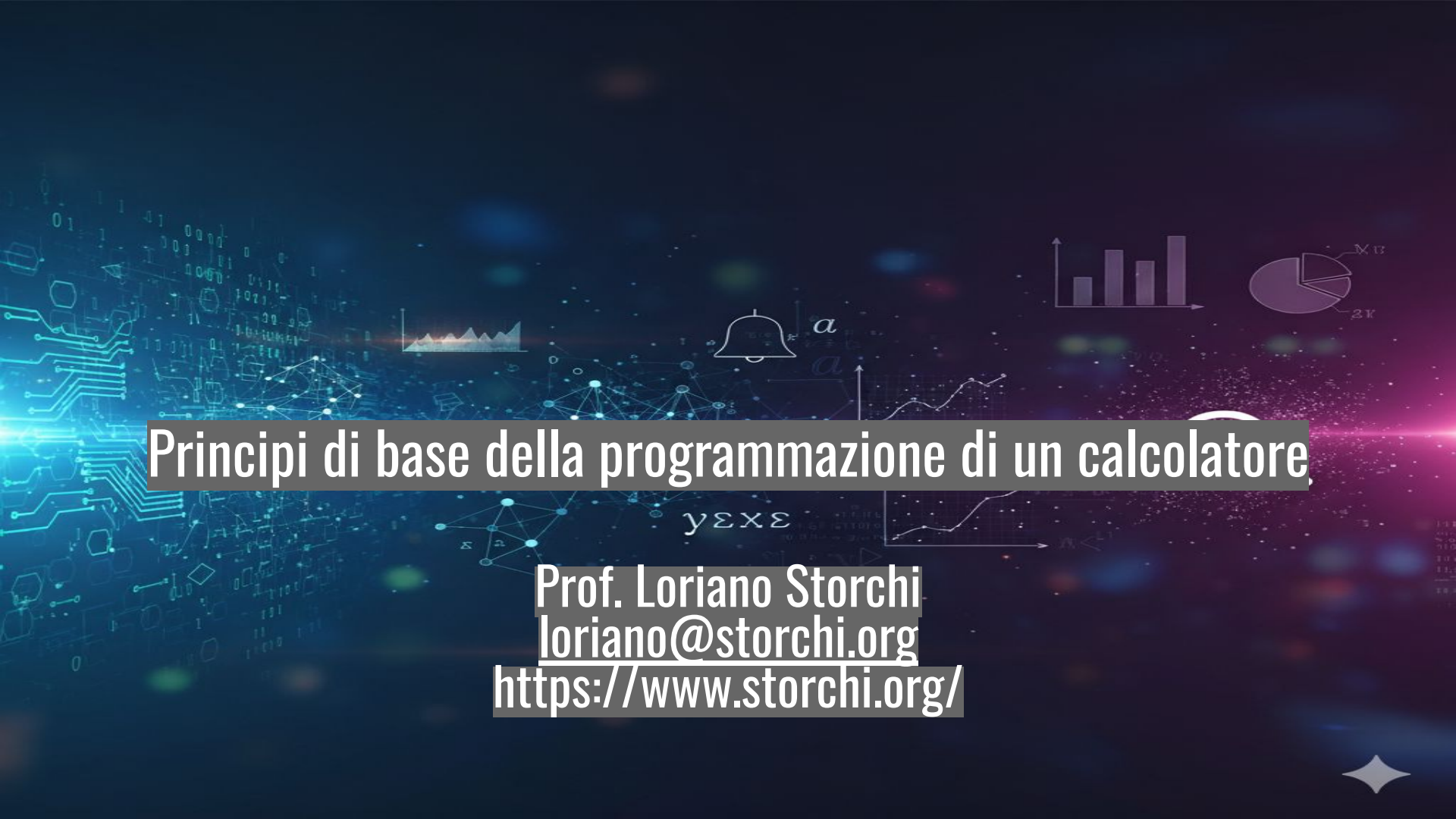


The background is a dark blue and purple gradient with various digital and mathematical motifs. On the left, there are glowing blue circuit lines and binary code (0s and 1s). In the center, there are faint network diagrams with nodes and lines. On the right, there are several mathematical and data-related icons: a bar chart, a pie chart with labels 'x' and 'y', a bell curve with labels 'a' and 'b', and a line graph. The overall aesthetic is high-tech and academic.

Principi di base della programmazione di un calcolatore

Prof. Lorianò Storchi
loriano@storchi.org
<https://www.storchi.org/>

A small white four-pointed star icon is located in the bottom right corner of the image.



Algoritmi

- Gli algoritmi descrivono come vengono trasformate le informazioni. L'informatica si occupa della loro teoria, analisi, progettazione, efficienza, implementazione e applicazione.
- Un algoritmo è un processo formale che risolve un determinato problema attraverso un numero finito di passaggi. Il termine deriva dalla trascrizione latina del nome del matematico persiano al-Khwarizmi, considerato uno dei primi autori ad aver fatto riferimento a questo concetto. L'algoritmo è un concetto fondamentale dell'informatica, soprattutto perché sta alla base della nozione teorica di calcolabilità: un problema è calcolabile quando può essere risolto mediante un algoritmo. (Wikipedia)



Programmazione: dall'algoritmo all'esecuzione

La programmazione è l'attività con cui si istruisce un computer a eseguire azioni specifiche sui dati di input, per risolvere problemi e produrre risultati. È l'implementazione pratica di un algoritmo.

DATI DI INPUT

INFORMAZIONI
GREZZE

ISTRUZIONI

COMPUTER ELETTRONICO

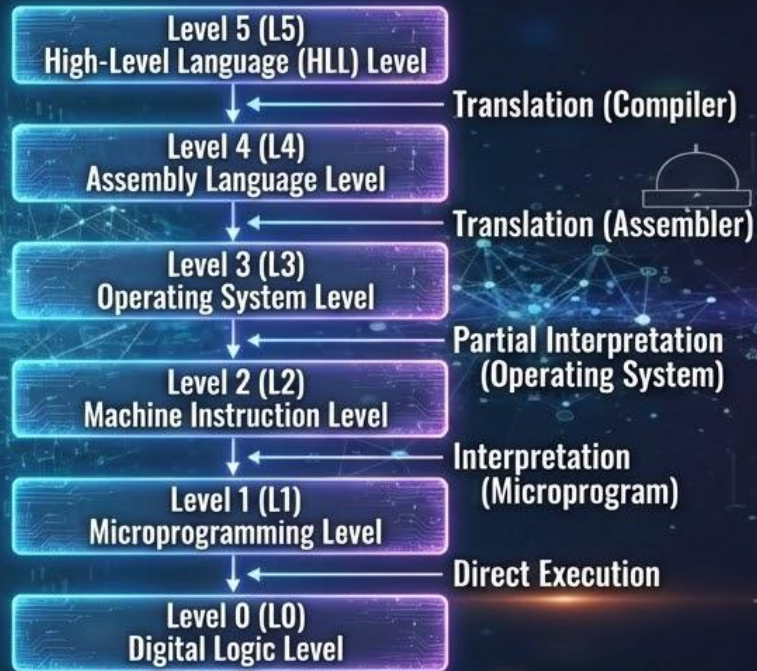
Motore di elaborazione

DATI DI
OUTPUT

SOLUZIONE

"Implementazione di un dato algoritmo per risolvere un problema."

LANGUAGES AND LAYERS



Level L0 represents the actual computer and the machine language it can directly execute. Each higher level represents an abstract machine. The programs (instructions) of each higher level must either be translated into instructions from one of the lower levels, or interpreted by a program running on an abstract machine of a strictly lower level.

MACHINE LANGUAGE AND ASSEMBLY



Machine Language

Every computer is capable of interpreting a low-level language called **machine language**. Instructions (**OPCODEs**) are simple **sequences of bits** that the processor interprets and follows a precise series of operations. Each instruction at this level consists of extremely basic operations.

[illegible]

Assembly Language

To make programming more immediate, the first step was the introduction of **ASSEMBLY**, where binary code is replaced by **mnemonic instructions** that are more easily usable by humans.

PSEUDO-ASSEMBLY

```

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

```

Comprendere i tipi di linguaggio

I linguaggi di programmazione odierni sono diversi e progettati per compiti computazionali specifici.

Linguaggi naturali



Nati spontaneamente ed estremamente espressivi.

"la vecchia porta la sbarra"

Intrinsecamente ambigui, poiché il significato dipende dal contesto.

Linguaggi artificiali



Creati in una data precisa e da autori definiti.

Possono essere formali o non formali in base alla loro struttura.

Linguaggi formali



Stringhe non ambigue costruite a partire da un alfabeto finito.

Definite da sintassi e semantica precise, che consentono la verifica algoritmica.

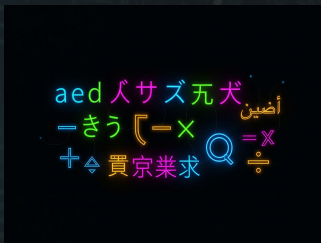
LANGUAGES

- Oggi esistono numerosi linguaggi di programmazione. In generale, ogni linguaggio è più o meno adatto a uno scopo specifico.
- I linguaggi naturali nascono spontaneamente, sono estremamente espressivi ma ambigui: “la vecchia porta la sbarra”.
- I linguaggi artificiali hanno una data di nascita precisa e un insieme di autori. Possono essere formali o non formali.
- I linguaggi formali sono linguaggi non ambigui composti da un insieme finito di stringhe costruite a partire da un alfabeto finito. La forma delle frasi (sintassi) e il loro significato (semantica) sono definiti in modo preciso e non ambiguo. È quindi possibile definire una procedura algoritmica capace di verificarne la correttezza grammaticale.

Definire rigorosamente un linguaggio

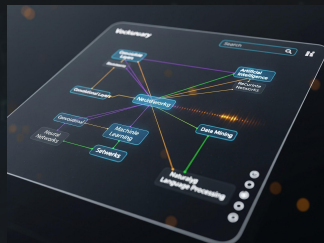
Per definire formalmente un linguaggio sono necessari quattro strumenti concettuali fondamentali:

Alfabeto



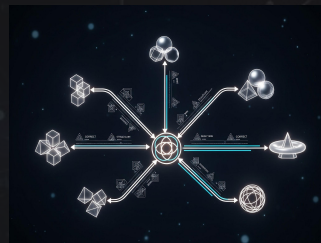
L'insieme dei simboli di base necessari per costruire le unità fondamentali o le parole di un linguaggio.

Lessico



Il vocabolario completo e le regole per la formazione delle parole nel sistema linguistico.

Sintassi



Le regole grammaticali che stabiliscono se una specifica sequenza di parole, cioè una frase, è strutturalmente valida.

Semantica



Definisce il significato effettivo. Esempio: `int a[5];` in C riserva memoria per 5 interi.

LANGUAGES

- Per definire rigorosamente un linguaggio servono alcuni strumenti di base:
 - **Alfabeto:** l'insieme dei simboli di base necessari per comporre le parole.
 - **Lessico:** l'insieme delle regole necessarie per scrivere le parole di un linguaggio, cioè il vocabolario.
 - **Sintassi, o regole grammaticali:** l'insieme delle regole che permette di stabilire se una frase, cioè un insieme di parole, è corretta.
 - **Semantica:** definisce il significato di una “frase” sintatticamente corretta. Per esempio, `int a[5];` in C permette di riservare lo spazio in memoria necessario a contenere 5 interi.



Linguaggi di programmazione ad alto livello

Dalla logica del processore alla leggibilità per le persone e all'indipendenza dalla macchina.

Evoluzione dall'hardware

A differenza dei linguaggi a basso livello, come Assembly e linguaggio macchina, i linguaggi ad alto livello sono progettati per efficienza e leggibilità.

Colmano, attraverso l'astrazione, la distanza tra la logica umana e l'esecuzione binaria.

Classificazione moderna

Esistono migliaia di linguaggi, ma solo una decina domina il settore per paradigmi specifici.

Linguaggi comuni:

PYTHON

JAVA

C++

C#

FORTRAN

PASCAL

BASIC

OBJECTIVE-C

LANGUAGES

- Abbiamo già menzionato i linguaggi artificiali, il linguaggio macchina e i linguaggi a basso livello, come Assembly.
- I linguaggi ad alto livello si allontanano dalla logica del processore e sono costruiti per essere semplici, efficienti, leggibili e indipendenti dalla macchina.
- Oggi sono disponibili molti linguaggi di programmazione, anche se solo una decina viene effettivamente usata. Di seguito proponiamo una breve classificazione di questi linguaggi: C, C++, C#, Java, Python, Fortran, Pascal, Basic, Objective-C e molti altri. È evidente che ogni paradigma di programmazione è più o meno adatto a uno scopo specifico.

CLASSIFICAZIONE DEI LINGUAGGI



CLASSIFICAZIONE DEI LINGUAGGI

- Imperativo: esegui questa azione, poi quella e modifica questo stato.
 - Procedurale: raggruppa le azioni in funzioni riutilizzabili.
 - Orientato agli oggetti: riunisce azioni e stato negli oggetti.
- Dichiarativo: ecco la verità, ora fornisci la risposta.
 - Funzionale: la verità è definita da funzioni matematiche.
 - Logico: la verità è definita da fatti e regole logiche.



IMPERATIVO



La programmazione imperativa è un paradigma che usa una sequenza di comandi passo dopo passo per definire esplicitamente come un programma modifica il proprio stato al fine di ottenere il risultato desiderato.



LINGUAGGI IMPERATIVI

- Il componente fondamentale del programma è l'istruzione e ogni istruzione indica l'operazione da eseguire. Le singole istruzioni operano sui dati del programma.
- Le istruzioni vengono eseguite una dopo l'altra.
- Ogni programma è costituito da due parti fondamentali: la dichiarazione dei dati e l'algoritmo, inteso come sequenza di operazioni.
- Ogni istruzione è un ordine, mentre nella programmazione dichiarativa il programma è una serie di affermazioni.
- Dal punto di vista sintattico, molti linguaggi imperativi usano verbi all'imperativo, come PRINT e READ.



LINGUAGGI IMPERATIVI

Quindi una

```
1 PROGRAM SimpleAddition
2   ! 1. Initialize state / Declare variables
3   ! We tell the computer to reserve memory for three real (decimal) numbers.
4   REAL :: A, B, C
5   ! Prompt the user so they know what to do
6   PRINT *, "Please enter two numbers separated by a space:"
7   ! 2. Read the input and change the state of A and B
8   READ *, A, B
9   ! 3. Explicit calculation and mutation
10  ! We calculate A + B and store the result in the memory location for C
11  C = A + B
12  ! 4. Output the final state
13  PRINT *, "The sum (C) is: ", C
14 END PROGRAM SimpleAddition
```

il risultato.



PROGRAMMAZIONE PROCEDURALE

- Può essere considerata un sottoparadigma della programmazione imperativa.
- Introduce il concetto di sottoprogrammi o funzioni.
- Questo consente di creare porzioni di codice sorgente utili per svolgere funzioni specifiche.
- Questi sottoprogrammi possono ricevere parametri di input e restituire valori di output.



PROGRAMMAZIONE PROCEDURALE



functions.c

THE PROCEDURES

```
// Reusable block 1
float calculateTax(float amount) {
    return amount * 0.20;
}

// Reusable block 2
void printReceipt(float base, float tax) {
    float total = base + tax;
    printf("Total: $%f", total);
}
```



main.c

THE COORDINATOR

```
int main() {
    float cartValue = 100.0;

    // Call Procedure 1
    float myTax = calculateTax(cartValue);

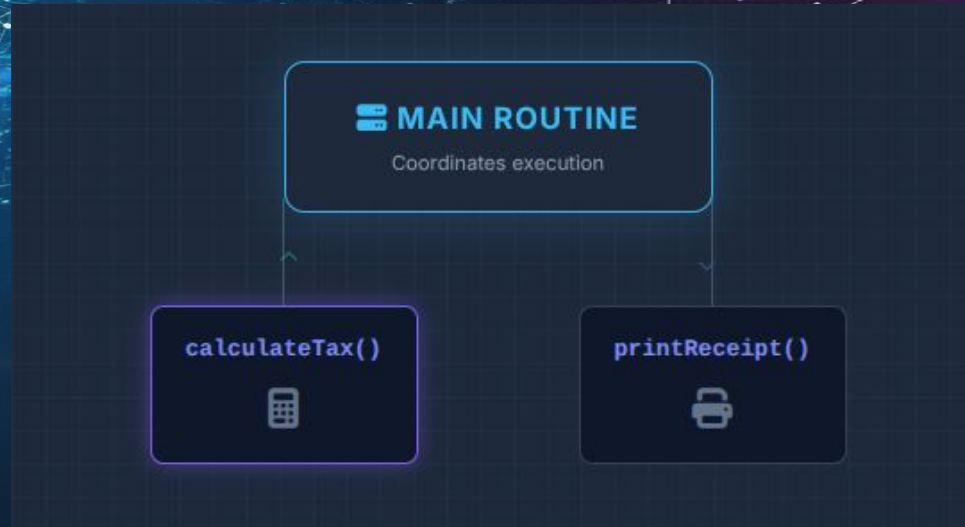
    // Call Procedure 2
    printReceipt(cartValue, myTax);

    return 0;
}
```

Uso di sottoprogrammi e funzioni, per la riusabilità del codice e le librerie di funzioni. Invece di scrivere ogni passaggio nel blocco di esecuzione principale, definiamo funzioni isolate che ricevono input, svolgono un compito specifico e restituiscono un output. Il blocco main() agisce quindi da coordinatore.

PROGRAMMAZIONE PROCEDURALE

Il concetto: il programma principale non svolge tutto il lavoro. Invia argomenti, cioè dati, a procedure specifiche. La procedura elabora i dati e restituisce il risultato al programma principale, che decide poi cosa fare. Questo rende il codice molto modulare e riutilizzabile.



Visualizzare il salto e il ritorno

Programma principale

```
print("Starting ... ")
```

```
res = calculate(10)
```

```
print("Result:", res)
```

Sottoprogramma (memoria)

```
def calculate(x):
```

```
    val = x * 2
```

```
    return val
```



Come ricorda dove deve tornare?

Quando un programma salta a un sottoprogramma, deve sospendere il lavoro corrente e ricordare esattamente dove si era fermato. Lo fa usando lo stack delle chiamate.

- 📦 **Push:** prima del salto, il computer inserisce nello stack nascosto nella memoria di sistema un "indirizzo di ritorno", cioè un segnalibro.
- 🖨 **Esecuzione:** la CPU sposta completamente l'attenzione sulla nuova funzione, allocando nuove variabili locali.
- ⬅ **Pop, o ritorno:** quando la funzione incontra la parola chiave return, la CPU rimuove quell'indirizzo dallo stack, recupera i dati e riprende l'esecuzione correttamente.



PROGRAMMAZIONE STRUTTURATA

- Può essere considerata un sottoparadigma della programmazione imperativa.
- Nella pratica, il programmatore è limitato all'uso delle sole strutture di controllo canoniche che non includono istruzioni di salto incondizionato, come GOTO. La sintassi del linguaggio vieta quindi strutture che non rispettano determinati vincoli.
- L'uso di GOTO porta inevitabilmente a una scarsa leggibilità del codice, il cosiddetto spaghetti code.

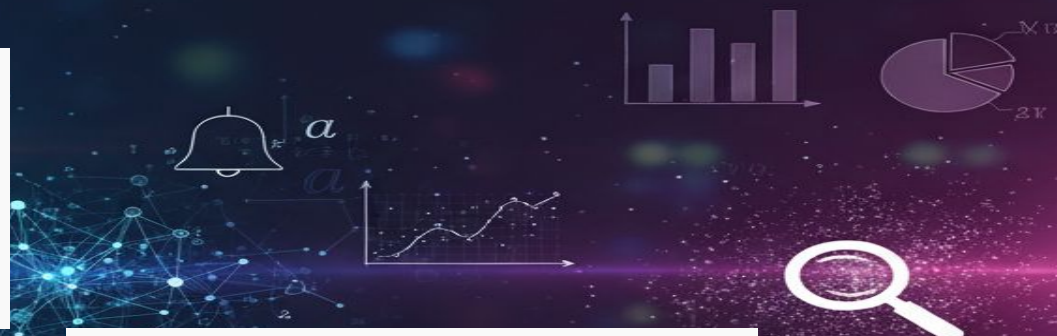


PROGRAMMAZIONE STRUTTURATA

• 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

- Può essere considerata un sottoparadigma della programmazione imperativa.
- Questo paradigma consente di definire oggetti software che possono interagire tra loro.
- Organizzare il software come oggetti facilita il riuso e una migliore organizzazione dei grandi progetti.
- I linguaggi OOP raggruppano parti del codice sorgente in classi. Ogni classe contiene dati e metodi, cioè funzioni, che operano su tali dati. Le classi sono modelli astratti invocati durante l'esecuzione per creare, o istanziare, oggetti software.
- Un linguaggio orientato agli oggetti consente di implementare tre meccanismi di base tramite la propria sintassi: incapsulamento, polimorfismo ed ereditarietà.

PROGRAMMAZIONE ORIENTATA AGLI OGGETTI



Attributes (Variables)

The variables that hold the geometric shape's properties.

```
double    sideLength ;
```

```
String    color ;
```

```
int       x_position ;
```

```
int       y_position ;
```



Methods (Functions)

The mathematical functions or actions the shape can perform.

```
double getArea() {  
    return this.sideLength *  
           this.sideLength;  
}
```

```
void resize(double newSide) {  
    this.sideLength = newSide;  
}
```

PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

Geometry.java

```
public static void main(String[] args) {  
  
    // 1. Instantiate Object 1 (Small Red Square)  
    Square boxA = new Square(5.0, "Red");  
    double areaA = boxA.getArea(); // Returns 25.0  
  
    // 2. Instantiate Object 2 (Large Blue Square)  
    Square boxB = new Square(10.0, "Blue");  
    boxB.resize(12.0); // Change its side length  
    double areaB = boxB.getArea(); // Returns 144.0  
  
}
```

OBJECTS IN MEMORY

boxA (Instance)



sideLength: 5.0

color: "Red"

x_position: 0

y_position: 0

boxB (Instance)



sideLength: 12.0

color: "Blue"

x_position: 0

y_position: 0

PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

Incapsulamento: una separazione precisa tra l'implementazione e l'interfaccia della classe. Gli utenti della classe, o dell'oggetto, non devono conoscere i dettagli dell'implementazione. Usano la classe tramite metodi e dati pubblici, interagendo con l'oggetto senza conoscere tali dettagli.



PROGRAMMAZIONE ORIENTATA AGLI OGGETTI



Square.java (Bad)

WITHOUT ENCAPSULATION

```
public class Square {  
    // Danger: Direct access allowed!  
    public double sideLength;  
}  
  
// Somewhere in Main:  
Square box = new Square();  
  
box.sideLength = -10.0; // Math is broken!  
Squares can't be negative.
```



Square.java (Good)

WITH ENCAPSULATION

```
public class Square {  
    // Safe: Hidden inside the object  
    private double sideLength;  
  
    // The "Gatekeeper" method (Setter)  
    public void setSideLength(double s) {  
        if (s > 0) {  
            this.sideLength = s;  
        } else {  
            print("Error: Invalid size!");  
        }  
    }  
}
```

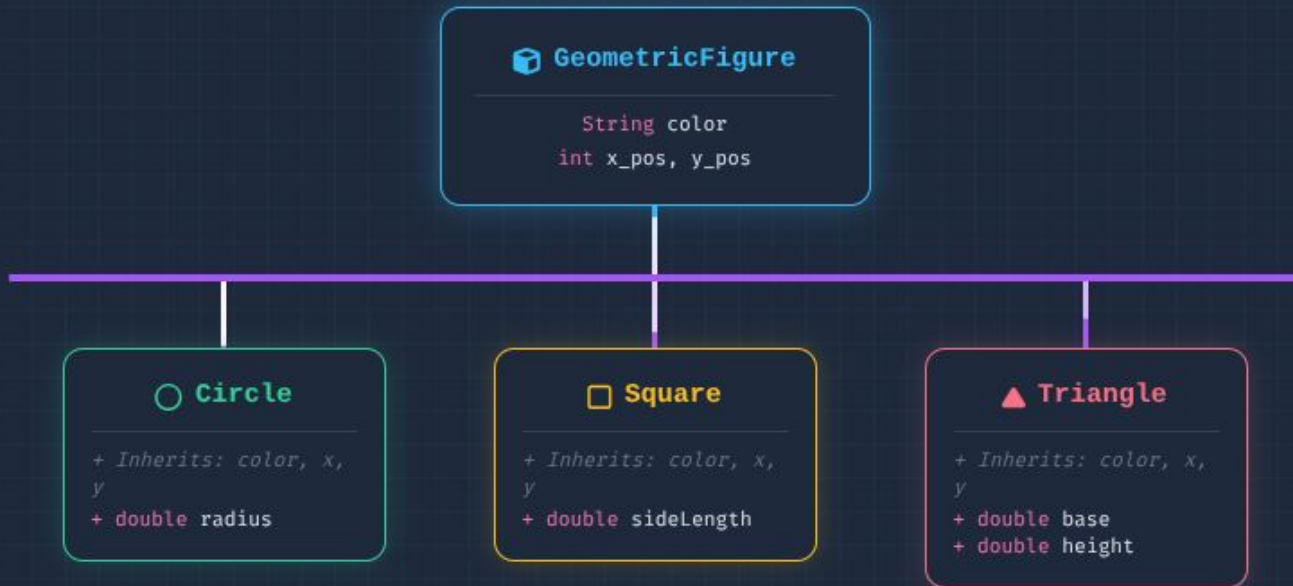
PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

- **Ereditarietà: una classe può ereditare da una classe base ed evolvere o specializzare le proprie funzionalità.**
 - Per esempio, possiamo immaginare una classe base `figura_geometrica` da cui derivano classi come `triangolo`, `cerchio` e `quadrato`.
- **Le classi derivate da una classe base ereditano tutti i metodi e le proprietà della classe base, ma possono specializzarsi definendo metodi e dati propri.**
 - Se B è una sottoclasse, o più in generale un sottotipo, di A, qualunque programma o funzione che può usare A può usare anche B.



PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

2. The Class Hierarchy



PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

GeometricFigure.java

```
public abstract class GeometricFigure {  
    // Shared attributes for ALL shapes  
    protected String color;  
    protected int x_pos;  
    protected int y_pos;  
  
    // A contract: Every shape MUST have an  
    // area  
    public abstract double getArea();  
}
```

Shapes.java

```
// Circle inherits color, x, y  
public class Circle extends GeometricFigure  
{  
    private double radius; // Specialization  
  
    public double getArea() {  
        return Math.PI * (radius * radius);  
    }  
}  
  
// Square inherits color, x, y  
public class Square extends GeometricFigure  
{  
    private double sideLength; //  
    // Specialization  
  
    public double getArea() {  
        return sideLength * sideLength;  
    }  
}
```



PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

- **Polimorfismo: possiamo esemplificare il concetto come più definizioni della stessa funzione, cioè overloading, e classi e funzioni parametriche rispetto al tipo di dato. Un esempio semplice è il sovraccarico di funzione.**
- **Possiamo distinguere formalmente almeno quattro tipi di polimorfismo: per inclusione, parametrico, per overloading e per coercizione.**
- **Presenteremo solo alcuni esempi semplici per chiarire il concetto generale.**



OBJECT-ORIENTED PROGRAMMING

Let's imagine the usual `geometric_figure` class from which we will have derived two classes: `circle` and `triangle`



When the user calls the `calculate area` method it will perform a certain action (calculating the area in both cases even though they have the same name)

PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

Shapes.java

THE SETUP

```
// Each shape implements its own math
class Circle extends GeometricFigure {
    public double getArea() {
        return Math.PI * (radius * radius);
    }
}

class Triangle extends GeometricFigure {
    public double getArea() {
        return (base * height) / 2.0;
    }
}
```

Main.java

POLYMORPHISM IN ACTION

```
// 1. Store different shapes in ONE generic
array
GeometricFigure[] myShapes = {
    new Circle(5.0),
    new Triangle(4.0, 6.0)
};

// 2. Loop through them blindly
for (GeometricFigure shape : myShapes) {
    // The EXACT SAME line of code...
    // ... does two completely different
    things!
    double area = shape.getArea();
    System.out.println(area);
}
```



DICHIARATIVO



PROGRAMMAZIONE FUNZIONALE

- Come suggerisce il nome, il flusso di esecuzione assume la forma della valutazione di una serie di funzioni matematiche. Il programma è quindi un insieme di funzioni.
- Nei linguaggi funzionali puri non esiste il concetto di assegnazione né di allocazione esplicita della memoria.
- I valori non si ottengono modificando lo stato del programma, poiché non vi è assegnazione di valori, ma costruendo il nuovo stato tramite funzioni a partire dallo stato precedente.
- È usata nell'IA, con uso limitato o nullo nei contesti industriali.
- Le funzioni possono essere passate come parametri e restituite come risultato da altre funzioni.

PROGRAMMAZIONE FUNZIONALE

Invece di scrivere cicli for che modificano manualmente una variabile total, come nella programmazione imperativa, concatenamo funzioni pure che ricevono un array e restituiscono un array completamente nuovo, senza modificare i dati originali.

functional_sum.js

PURE PIPELINE

```
// 1. Immutable Data (Never changes)
const numbers = [1, 2, 3, 4, 5];

// 2. The Data Transformation Pipeline
const result = numbers
  // WHAT to do (Keep evens)
  .filter(n => n % 2 !== 0)
  // WHAT to do (Double them)
  .map(n => n * 2)
  // WHAT to do (Sum them up)
  .reduce((sum, n) => sum + n, 0);

console.log(result); // Outputs 12
console.log(numbers); // Still [1, 2, 3, 4, 5]!
```

FUNCTIONAL PROGRAMMING



Imperative Approach (Manual State)

Instead of writing `for` loops that manually change a `total` variable, we maintain a mutable state.

▼ imperative_sum.js

```
// 1. Manually initialize the state
let result = 0;

// 2. Explicit control flow (HOW to loop)
for (let i = 0; i < numbers.length; i++) {
  // 3. The "filter" step
  if (numbers[i] % 2 === 0) {
    // 4. The "map" step
    let doubled = numbers[i] * 2;
    // 5. The "reduce" step (mutating state)
    result = result + doubled;
  }
}

console.log(result); // Outputs: 12
```



Functional Approach (Pure Pipeline)

We chain together pure functions that take an array and return a brand new array, never modifying the original data.

▼ functional_sum.js

```
// 1. Immutable Data (Never changes)
const numbers = [1, 2, 3, 4, 5];

// 2. The Data Transformation Pipeline
const result = numbers
  .filter(n => n % 2 === 0) // Keep evens
  .map(n => n * 2) // Double them
  .reduce((sum, n) => sum + n, 0); // Sum them up

console.log(result); // Outputs: 12
```

PROGRAMMAZIONE FUNZIONALE

Il concetto: pensa a un impianto di purificazione dell'acqua. L'acqua, cioè i dati, scorre attraverso filtri e processori chimici diversi, cioè le funzioni. Le tubature non modificano la sorgente originale dell'acqua, ma restituiscono una versione trasformata di ciò che le attraversa.



LINGUAGGI DICHIARATIVI, O LOGICI

- Rispetto al paradigma imperativo, il programma è costituito da una serie di affermazioni e non di comandi.
- Il programma specifica CHE COSA si desidera ottenere, non COME. Il come è lasciato all'esecutore.
- In pratica, il programma, o la sua esecuzione, può essere considerato una dimostrazione della verità di un'affermazione.



LINGUAGGI DICHIARATIVI, O LOGICI

Imperative (The HOW)

You provide explicit, step-by-step instructions on how the computer must achieve the goal, carefully managing loops and mutating state.

```
let evens = [];  
for (let i = 0; i < arr.length; i++) {  
  if (arr[i] % 2 === 0) {  
    evens.push(arr[i]);  
  }  
}
```

Declarative (The WHAT)

You declare the result you want. The underlying engine determines the most efficient way to achieve that outcome.

```
const evens = arr.filter(n => n % 2 === 0);  
  
// No loops, no manual mutation.  
// The intent is clear immediately.
```

SQL: il pioniere

Structured Query Language (SQL) is arguably the most famous and widely used declarative language in history.

When you write a query, you never specify the search algorithms, memory allocation, or indexing strategies.

```
SELECT name, email  
FROM users  
WHERE status = 'active'  
ORDER BY created_at DESC;
```

The database engine acts as the "chef", figuring out the most optimal execution plan to retrieve your data.



Tipizzazione nella programmazione



Comprendere le variabili

Gli elementi fondamentali di tutti i programmi per computer. Come memorizziamo, etichettiamo e manipoliamo i dati in memoria.

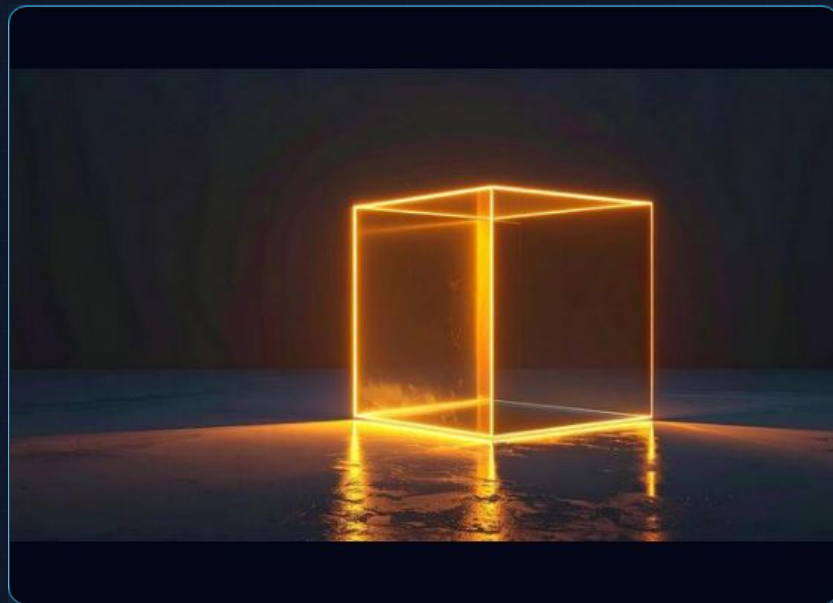
La metafora della "scatola di archiviazione"

Il modo più semplice per comprendere una variabile è pensarla come una scatola di archiviazione etichettata nella memoria del computer.

 **Identificatore, o nome:** l'etichetta che scrivi all'esterno della scatola per ritrovarla in seguito, ad esempio `playerName`.

 **Valore, o dato:** l'informazione effettiva che inserisci nella scatola, per esempio il testo `"Alex"`.

 **Mutabilità:** proprio come in una scatola fisica, puoi togliere i vecchi dati e sostituirli con nuovi dati in qualsiasi momento durante l'esecuzione del programma.



Anatomia di una variabile

1. Dichiarazione e assegnazione

Per usare una variabile, devi prima dichiararla, cioè creare la scatola, e poi assegnarle un valore, cioè inserire i dati.

Nei linguaggi moderni, questa operazione viene di solito svolta in una sola riga di codice.

 Il segno di uguale = è l'operatore di assegnazione. Prende i dati a destra e li memorizza nella variabile a sinistra.

2. La sintassi del codice

```
// Esempio JavaScript  
let userAge = 25;  
let status = "Active";
```

```
// Esempio Python  
score = 100
```

Parola chiave	Nome	Operatore	Valore
let	userAge	=	25

Tipi di dati comuni, ovvero la forma della scatola



Stringhe

Dati testuali, racchiusi tra virgolette. Usati per nomi, messaggi e indirizzi.

```
"Hello World"
```



Numeri

Valori matematici usati per i calcoli. Possono essere interi o numeri decimali, cioè float.

```
42 | 3.14
```



Booleani

Un semplice interruttore logico che può assumere solo uno di due stati possibili. È usato per la logica condizionale.

```
true | false
```

ARRAY



Array

How can I easily handle information that is structured by nature? For example a complex number, number, or a vector or a matrix?

Typical structured variables are arrays

C Example: Vector

```
float v[10]; // Declaration
```

We can then access the i -th element of the vector:

```
v[i-1] = 3.5;
```

C Example: Matrix (2D Array)

Likewise I can define an array (two-dimensional array) as:

```
float m[10][10];
```

Array

Se una normale variabile è una singola scatola, un array è una fila continua di scatole, come una libreria, nella memoria del computer.

- Posizione di memoria: l'indirizzo fisico nella RAM, ad esempio 200 o 201. Gli array occupano spazio contiguo.
- Indice, o pedice: la posizione logica che usiamo per accedere ai dati.
- Indicizzazione da zero: in quasi tutti i linguaggi il primo elemento è sempre all'indice 0, non 1.



Array

Un array bidimensionale è semplicemente un "array di array". Lo visualizziamo concettualmente come una griglia o una matrice con righe e colonne.

- "Qui l'array è `abc[5][4]`, che può essere visto concettualmente come una matrice di 5 righe e 4 colonne. È importante notare che l'indice parte da zero, quindi `abc[0][0]` è il primo elemento dell'array."

2D ARRAY CONCEPTUAL REPRESENTATION

Second Subscript (Col) →

abc[0][0]	abc[0][1]	abc[0][2]	abc[0][3]
abc[1][0]	abc[1][1]	abc[1][2]	abc[1][3]
abc[2][0]	abc[2][1]	abc[2][2]	abc[2][3]
abc[3][0]	abc[3][1]	abc[3][2]	abc[3][3]
abc[4][0]	abc[4][1]	abc[4][2]	abc[4][3]

↑
First Subscript (Row)



Example

```
1 float a[N], b[N];
2 for (i=0; i<N; ++i)
3 {
4     a[i] = (float)rand()/(float)(RAND_MAX/N);
5     b[i] = (float)rand()/(float)(RAND_MAX/N);
6 }
7 s = 0.0;
8 for (i=0; i<N; ++i)
9 {
10     s = s + a[i]*b[i];
11 }
```



LINGUAGGI

- I linguaggi possono anche essere classificati in base al loro sistema di tipi: tipizzazione statica e dinamica.
- **Tipizzazione statica:** il programmatore è obbligato a specificare esplicitamente il tipo di ogni elemento sintattico. Per esempio, deve specificare il tipo di una variabile e il linguaggio garantisce poi che la variabile venga usata in modo coerente con la dichiarazione.
- **Tipizzazione dinamica:** in questo caso il dato assume un tipo che può variare durante l'esecuzione in base alle assegnazioni effettuate, come in Python.
- Possiamo inoltre distinguere tra tipizzazione debole e forte.



LINGUAGGI

1. WHEN are types checked?

Static Typing: Checked at Compile-Time.

Dynamic Typing: Checked at Run-Time.

2. HOW STRICTLY are they checked?

Strong Typing: Strict enforcement, no implicit conversions.

Weak Typing: Loose enforcement, language attempts to guess/coerce types.



LINGUAGGI

Static Typing

The programmer explicitly declares the type of a variable. The compiler guarantees that variable will only ever hold that type of data. Errors are caught **before** the program runs.

Java / C++

```
int age = 25;  
// The compiler locks 'age' as an integer  
  
age = "Twenty Five";  
^ ERROR: Incompatible types
```

Dynamic Typing

Variables do not have types; only the *values* do. A variable can hold a number right now, and a string on the next line. Types are evaluated **while** the program is running.

Python / JavaScript

```
age = 25  
# 'age' points to an integer value  
  
age = "Twenty Five"  
✓ PERFECTLY VALID
```



Oltre i linguaggi più diffusi



LINGUAGGI

Linguaggi o paradigmi per la programmazione parallela, destinati alle architetture moderne. Per approfondire:
<http://www.storchi.org/lecturenotes/acr/index.html>

- **Linguaggi esoterici: talvolta complessi e poco chiari. Diffusi soprattutto tra utenti esperti, vengono usati esclusivamente per mettere alla prova le capacità di programmazione, in sostanza per divertimento.**
- **Scripting: sviluppati inizialmente per le shell Unix, questi linguaggi automatizzano compiti ripetitivi e dispendiosi in termini di tempo.**

LINGUAGGI

Parallel Programming

Designed for **modern architectures** (Multi-core CPUs, GPUs, Supercomputers).

Instead of executing instructions one by one sequentially, parallel languages divide a large problem into smaller sub-problems that are solved **simultaneously**.

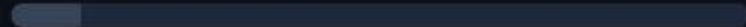
Use Cases:

Scientific simulations, Machine Learning training, Video rendering, Big Data processing.

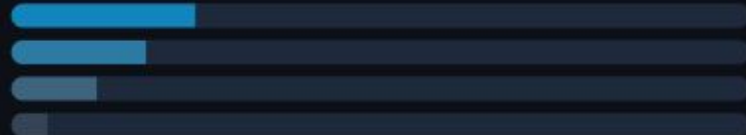
Examples:

CUDA (Nvidia GPUs), OpenMP, MPI, Erlang.

SEQUENTIAL (TRADITIONAL)



PARALLEL EXECUTION



LINGUAGGI

>_ Scripting Languages

backup_script.sh

```
#!/bin/bash
# Automating repetitive tasks
echo "Starting daily backup ..."
tar -czf backup_$(date +%F).tar.gz /var/www/html
scp backup_*.tar.gz admin@server:/backups/
echo "Done! Files secured."
-
```

Initially developed for Unix shells to **automate repetitive and time-consuming tasks**.

Today, they power the web and system administration. They are usually *interpreted* (run line-by-line) rather than compiled, making them fast to write and test.

System Admin

Bash, PowerShell

Web & General

Python, JavaScript, Ruby

LINGUAGGI

Esoteric Languages (Esolangs)

Designed to be deliberately **complex and unclear**.

They are popular only among experienced programmers. They are never used for production software, but rather to test programming skills, push the boundaries of language design, or purely for **entertainment purposes**.

Famous Examples:

- **Brainfuck:** Uses only 8 simple characters (+ - < > . , []).
- **Malbolge:** Designed to be the hardest language to write.
- **Shakespeare:** Code reads like a theatrical play.

BRAINFUCK: "HELLO WORLD" PROGRAM

```
+++++++[>++++[>++>+++>+++>+<<<←]>+>+>-  
>>+[<]←]>>.>——.+++++++..+++.>>.<.<.+  
+.<——.———.>>+.>+.
```

Yes, that actually prints "Hello World".

Altri concetti



Dal sorgente all'eseguibile

- Il codice sorgente è scritto in file di testo ASCII. Esprime l'algoritmo implementato nel linguaggio scelto. Per scriverlo si possono usare semplici editor di testo, come VI o Emacs, oppure IDE, ambienti di sviluppo integrati che includono strumenti quali compilatori, linker e debugger.
- **Compilazione:** il codice sorgente viene tradotto dal compilatore da un linguaggio di alto livello a codice eseguibile. Il vantaggio è un'esecuzione rapida e un codice ottimizzato per la piattaforma specifica. Lo svantaggio è che deve essere ricompilato per ogni diverso sistema operativo o hardware.
- **Linking:** ogni programma usa in genere una o più librerie e il linker collega tali librerie al programma sorgente. Il linking può essere statico o dinamico, per esempio con librerie .so in Linux e sistemi simili a Unix o .dll in Windows.



Dal sorgente all'eseguibile

1. The Source Code

The source code is written in simple **ASCII text files**. It expresses the algorithm implemented in the chosen language.

The Tools

- **Text Editors:** Lightweight tools like VI, Nano, Emacs, or VS Code.
- **IDEs (Integrated Development Environments):** Heavy-duty platforms (like Visual Studio or Eclipse) that bundle editors with built-in *compilers, linkers, and debuggers*.

app.c - Emacs

```
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

Dal sorgente all'eseguibile

⚙️ 2. Compilation

The compiler acts as a translator. It reads the high-level human-readable code and translates it entirely into low-level **machine code (binary)** before it can be run.

👍 The Advantage

Execution is extremely fast because the code is optimized directly for the specific hardware/CPU.

👎 The Disadvantage

Lack of portability. You must recompile the code separately for Windows, Linux, Mac, ARM, x86, etc.



Dal sorgente all'eseguibile

3. Linking

Your program rarely stands alone. It needs pre-written code (like math functions or UI drawing tools) from **Libraries**. The Linker stitches your compiled Object Code together with these libraries to create the final executable.

Static Linking

The library code is physically copied and baked right into your final executable file. (Bigger file size, easier to distribute).

Dynamic Linking

The executable just contains a "pointer" to the library. The OS loads the library into memory when the program runs.

Windows: `.dll` | Linux/Unix: `.so`



L'alternativa

- **Interpretazione:** per evitare il problema della portabilità dei programmi si usano gli interpreti. In questo caso il codice sorgente non viene compilato o tradotto, ma eseguito da un interprete. Ciò introduce altri problemi, come le prestazioni.
- **Bytecode e P-code:** è un approccio intermedio in cui il programma sorgente viene tradotto in un codice intermedio interpretato da una macchina virtuale. Combina una buona velocità di esecuzione con un'elevata portabilità. JIT, Just in Time, compila il codice intermedio in codice macchina al momento dell'esecuzione.



Interpretazione, l'alternativa

To solve the portability problem of compiled executables, the **Interpreter** model reads and executes the source code line-by-line at runtime, without creating a standalone executable file.

The Advantage

Extreme portability. The same source code file can run on Windows, Mac, or Linux, as long as the system has the Interpreter installed.

The Disadvantage

Lower performance. Translating high-level code to machine instructions on-the-fly during execution adds significant overhead.



 Python, Ruby, PHP

L'approccio ibrido, bytecode

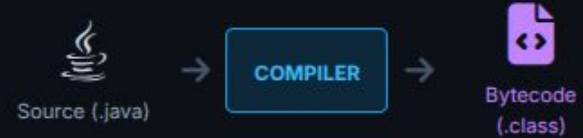
An intermediate approach combining the speed of compilation with the portability of interpretation.

1. Bytecode (P-Code)

The source code is first translated into a highly optimized, portable intermediate language called Bytecode. It is not machine-specific.

2. Virtual Machine & JIT

A Virtual Machine (VM) executes the bytecode. Modern VMs use **JIT (Just-In-Time) Compilation** to translate bytecode into native machine code exactly when it's needed during execution.



Calcolabilità

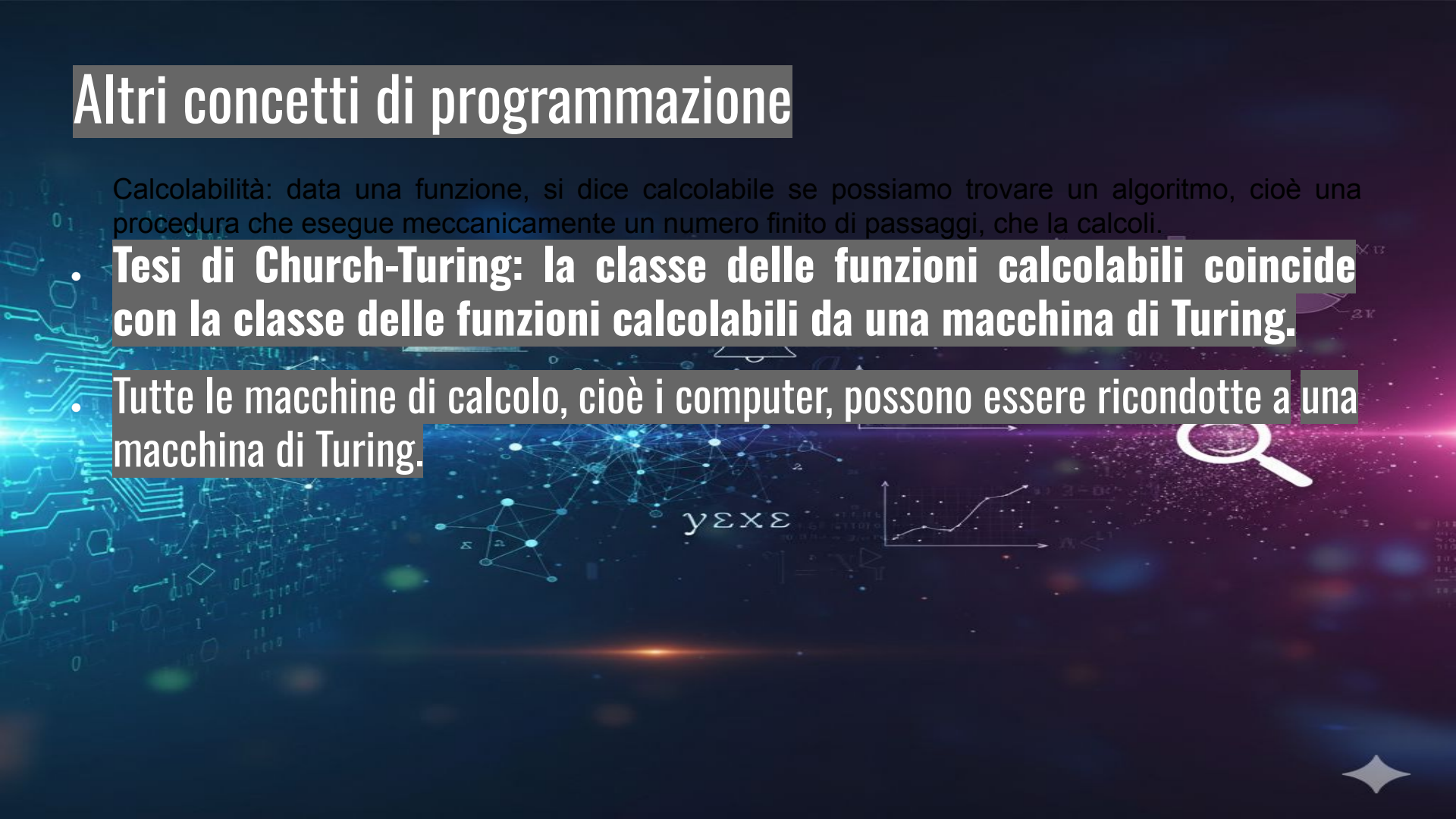


Altri concetti di programmazione

Calcolabilità: data una funzione, si dice calcolabile se possiamo trovare un algoritmo, cioè una procedura che esegue meccanicamente un numero finito di passaggi, che la calcoli.

- **Tesi di Church-Turing: la classe delle funzioni calcolabili coincide con la classe delle funzioni calcolabili da una macchina di Turing.**

- **Tutte le macchine di calcolo, cioè i computer, possono essere ricondotte a una macchina di Turing.**



Che cos'è la calcolabilità?

Given a mathematical function or a problem, it is said to be **computable** if we can find an algorithm that calculates it.

📌 What defines an Algorithm?

- ✓ **Finite Steps:** It must eventually terminate and produce a result.
- ✓ **Mechanical:** It must be a strict procedure that requires no human intuition or creativity to execute.
- ✓ **Unambiguous:** Every step must be exactly defined.

Input Data x

MECHANICAL PROCEDURE

1. Read x
2. Apply Rule A
3. If true, jump to 5
4. Apply Rule B
- ...

Result $f(x)$

La tesi di Church-Turing

"The class of computable functions exactly coincides with the class of functions computable by a Turing machine."

This is a **thesis**, not a mathematical theorem, because "intuitive computability" is a human concept. However, it is universally accepted.

It states that no matter how advanced, complex, or futuristic a computing system is, if a problem can be solved algorithmically, a simple Turing Machine can also solve it.

All computers trace back here

Turing's Infinite
Tape

↔ **Modern RAM /
Storage**

R/W Head + State
Table

↔ **CPU (ALU & Control
Unit)**

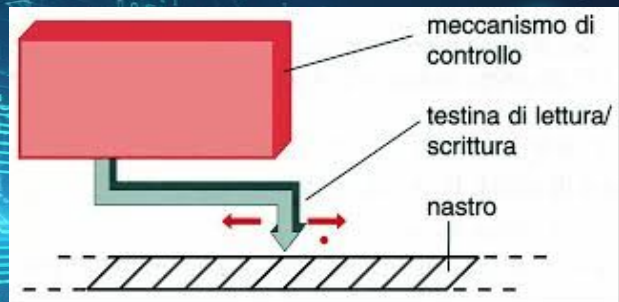
Universal Turing
Machine

↔ **Programmable
Computer**

Every laptop, smartphone, and supercomputer is essentially a physical implementation of a Universal Turing Machine.

La macchina di Turing

• MdT: macchina di Turing



Modello deterministico con nastro e istruzioni a 5 campi:

1. Un nastro di lunghezza arbitraria che può contenere caratteri o spazi vuoti.
2. Una testina di lettura e scrittura che legge e scrive sul nastro e può spostarsi a destra o a sinistra.
3. La macchina ha uno stato interno.

A ogni passo, la MdT legge un simbolo e, in base al proprio stato interno, può cambiare stato, scrivere un simbolo sul nastro e spostarsi a destra o a sinistra.

Il comportamento della MdT viene programmato definendo regole, o quadruple, del tipo:

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

Altri concetti di programmazione

MdT e problema dell'arresto:
dati un certo programma e
un certo input, è impossibile
stabilire se il programma
terminerà oppure no. Il
problema è strettamente
collegato al teorema di
incompletezza di Gödel.

∞ Connection to Gödel

The Halting Problem is the computational equivalent of Gödel's First Incompleteness Theorem in mathematics.



Kurt Gödel (1931): In any sufficiently strong formal logical system, there are true mathematical statements that simply *cannot be proven* within that system.



Alan Turing (1936): In any universal computing system, there are well-defined problems that simply *cannot be computed* by any algorithm.

THE FUNDAMENTAL LIMITS OF LOGIC



LA TEORIA FONDAMENTALE

Il teorema di Böhm-Jacopini

Le basi della programmazione strutturata e il modo in cui tre semplici regole logiche governano tutta l'informatica moderna.

L'intuizione del 1966

Dimostrato nel 1966 dagli informatici italiani Corrado Böhm e Giuseppe Jacopini, questo teorema ha cambiato per sempre il modo in cui scriviamo software, passando dalla programmazione non strutturata con GOTO a quella strutturata.

L'enunciato:

"Ogni algoritmo calcolabile può essere scritto usando esclusivamente una combinazione di tre strutture fondamentali di controllo."

Che si tratti di calcolare la fisica della luce in un videogioco moderno o di far elaborare a una rete neurale milioni di dati, alla base di tutto ci sono queste tre strutture.



Le tre strutture fondamentali



1. Sequenza

L'esecuzione delle istruzioni in ordine cronologico, rigorosamente una dopo l'altra.

```
x = 10;  
y = x + 5;  
print(y);
```



2. Selezione

La diramazione del flusso logico in base a una condizione, i famosi if ed else.

```
if (x > 10) {  
  print("Maggiore");  
} else {  
  print("Minore");  
}
```



3. Iterazione

La ripetizione di un blocco di istruzioni finché una condizione specifica risulta vera.

```
while (n > 0) {  
  print(n);  
  n = n - 1;  
}
```

La realtà della programmazione moderna

Anche se a livello hardware esistono sempre queste tre strutture, i linguaggi moderni offrono strumenti per nascondere il caos.

🚫 Codice spaghetti

Se provassimo a scrivere l'intero codice di un videogioco, composto da milioni di righe, usando solo lunghe sequenze di if e cicli, creeremmo codice illeggibile e impossibile da mantenere.

```
if (player.x > enemy.x) {  
  if (player.y == enemy.y) {  
    while (enemy.hp > 0) { ... }  
  } else { ... }  
} else if (...) { ... }  
// ... x 1,000,000 lines
```

📦 L'illusione della complessità

Per gestire l'enorme complessità del software moderno, gli informatici hanno inventato concetti astratti.

Questi concetti poggiano sulle tre strutture di base e le raggruppano in blocchi logici sensati che le persone possono leggere e gestire efficacemente.

Astrazioni per gestire la complessità



Programmazione orientata agli oggetti, OOP

Raggruppa variabili e funzioni in entità logiche chiamate oggetti, per esempio il giocatore in un videogioco, così da mantenere il codice rigorosamente organizzato.



Funzioni e moduli

Consentono di impacchettare una complessa sequenza di if e cicli, assegnarle un nome descrittivo e riutilizzarla infinite volte senza riscriverla.



Parallelismo e concorrenza

Invece di avere una sola sequenza alla volta, migliaia di sequenze, scelte e cicli vengono eseguiti contemporaneamente su processori diversi.

Ritorno alle origini

La verità sotto la superficie.

Alla fine, quando il nostro codice elegante e astratto passa attraverso il compilatore per essere tradotto nel linguaggio macchina, Assembly o binario, che la CPU deve effettivamente eseguire...

...tutte le astrazioni svaniscono.

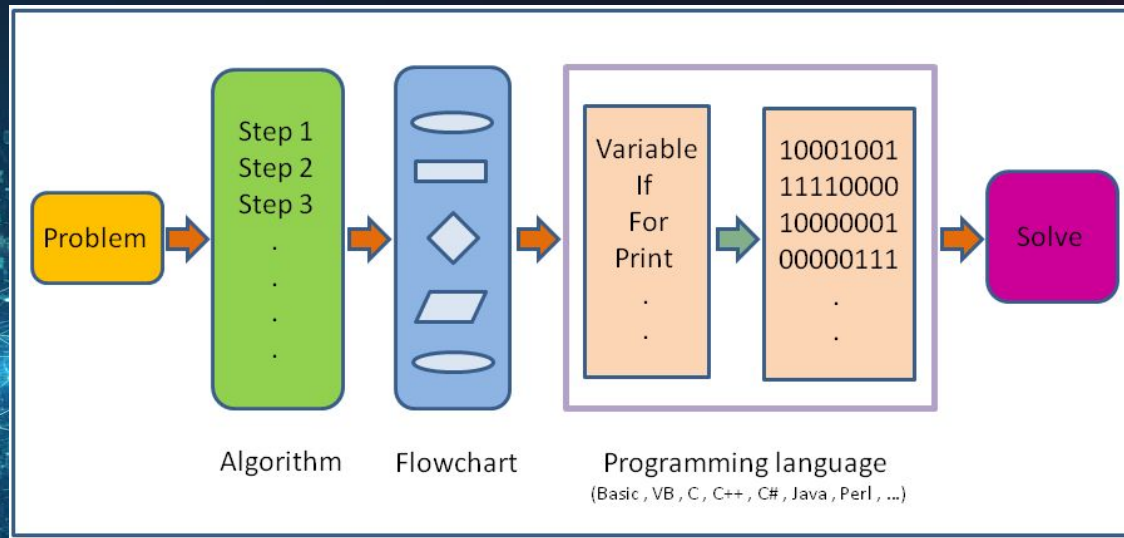
Il codice torna alla sua forma più pura, esattamente come descritto dal teorema di Böhm-Jacopini: un nastro molto lungo di istruzioni sequenziali, salti condizionali e cicli ripetuti.



DIAGRAMMI DI FLUSSO E PSEUDOCODICE



DIAGRAMMI DI FLUSSO E PSEUDOCODICE



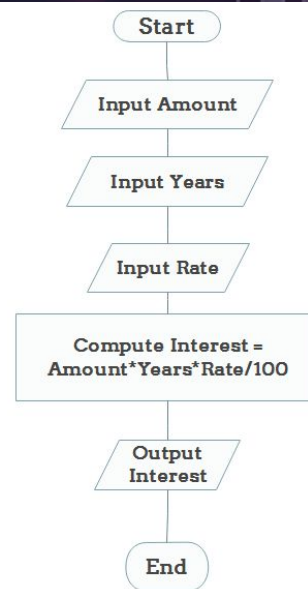
Esistono tre strutture fondamentali usate per la risoluzione algoritmica dei problemi: selezione, iterazione e sequenza, cioè una sequenza di istruzioni. L'istruzione GOTO, presente nei linguaggi macchina fin dagli anni Settanta, è stata progressivamente scoraggiata o eliminata.

Esempi: <https://github.com/lstorchi/teaching>

DIAGRAMMI DI FLUSSO E PSEUDOCODICE

- Vedremo un semplice esempio: calcolare l'interesse di un deposito bancario.

1.  Read amount,
2.  Read years,
3.  Read rate,
4.  Calculate the interest with formula "Interest=Amount*Years*Rate/100",
5.  Print interest.



DIAGRAMMI DI FLUSSO E PSEUDOCODICE

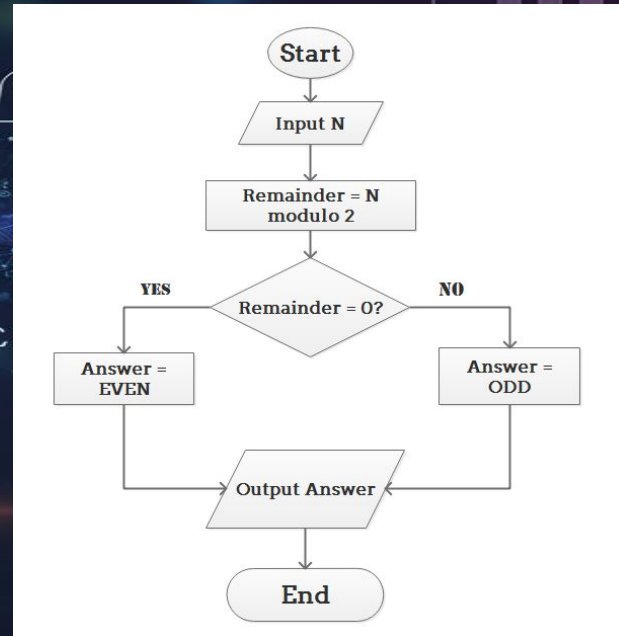
• Stabilire e mostrare se il numero N è pari o dispari

Passo 1: leggi il numero N.

Passo 2: imposta il resto a N modulo 2.

Passo 3: se il resto è uguale a 0, il numero N è pari, altrimenti è dispari.

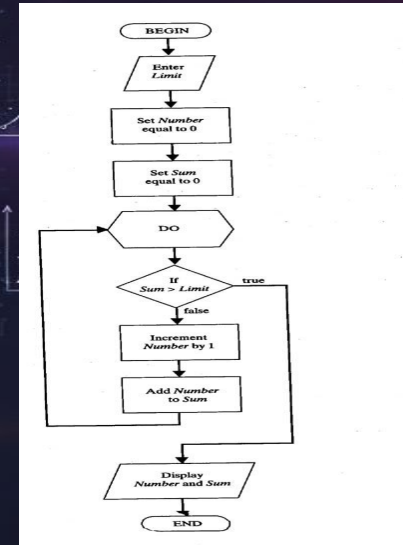
Passo 4: stampa il risultato.



DIAGRAMMI DI FLUSSO E PSEUDOCODICE

- Dato un valore **Limite**, qual è il più piccolo intero positivo **Numero** per cui la somma $\text{Somma} = 1 + 2 + \dots + \text{Numero}$ è maggiore di **Limite**? Qual è il valore di questa **Somma**?

1. Enter **Limit**
2. Set **Number** = 0.
3. Set **Sum** = 0.
4. Repeat the following:
 - a. If **Sum** > **Limit**, terminate the repetition, otherwise.
 - b. Increment **Number** by one.
 - c. Add **Number** to **Sum** and set equal to **Sum**.
5. Print **Number** and **Sum**.



STRUTTURE DI CONTROLLO



SEQUENZE

Sequenze

La struttura di controllo sequenziale esegue le istruzioni riga per riga, nello stesso ordine in cui compaiono nel programma. Può svolgere, per esempio, una serie di operazioni di lettura o scrittura, operazioni aritmetiche o assegnazioni a variabili.

Passo 1: leggi l'importo.

Passo 2: leggi gli anni.

Passo 3: leggi il tasso.

Passo 4: calcola l'interesse con la formula

" $\text{Interesse} = \text{Importo} * \text{Anni} * \text{Tasso} / 100$ "

Passo 5: stampa l'interesse.



SELEZIONE



Struttura di controllo della selezione

Logica generale per l'esecuzione condizionale:

```
if (condition 1)
    statements 1
else if (condition 2)
    statements 2
. . .
else
    statements N
endif
```

- 
- Flessibilità: else if può essere ripetuto indefinitamente.
 - Minimalismo: le strutture possono esistere come singolo blocco if.

Logica di selezione annidata

Le decisioni complesse usano istruzioni annidate:

```
if (condition 1)
  statement 1
  if (condition 2)
    statements 2
  end if
else
  statements 3
end if
```



- Controllo gerarchico: i blocchi interni vengono eseguiti solo se sono soddisfatte le condizioni del blocco padre.
- Profondità logica: consente un filtraggio granulare dei dati e del flusso del programma.

Operatori

- In tutti i linguaggi di programmazione posso usare operatori relazionali per confrontare numeri e variabili, per esempio:

Description	Java, C, C++	Fortran
Greater than	>	.GT.
Greater than or equal	>=	.GE.
Less than	<	.LT.
Less than or equal	<=	.LE.
Equal	==	.EQ.
Not Equal	!=	.NE.

```
Int a;
```

```
a = 4 // operatori di assegnazione
```

```
If (a == 5)
```

```
{
```

```
    cout << "Hello" << std::endl;
```

```
}
```

```
else if (a > 5)
```

```
{
```

```
    cout << a << " > 5 " << std::endl;
```

```
}
```



Operatori logici

- Diamo per noti gli operatori logici AND, OR e NOT.

Logical Operators		
Operator	Description	Example
&&	AND	<pre>x=6 y=3 x<10 && y>1 Return True</pre>
	OR	<pre>x=6 y=3 x==5 y==5 Return False</pre>
!	NOT	<pre>x=6 y=3 !(x==y) Return True</pre>

Consideriamo la seguente espressione:

$5 < a < 7$

Nei linguaggi di programmazione viene scomposta in due espressioni elementari collegate dall'operatore AND:

$(a > 5) \text{ AND } (a < 7)$

Operazioni bit a bit

- Non confondere gli operatori precedenti con le operazioni bit a bit.
- Sono operazioni usate per manipolare dati a livello di bit.
 - & (AND)
 - | (OR)
 - ^ (XOR, cioè 1 XOR 1 è zero)
 - ~ (complemento a uno, cioè da 0 a 1 e da 1 a 0)
 - >> (shift a destra: 11100101 >> 1 è 01110010)
 - << (shift a sinistra)



Operazioni bit a bit

- Un semplice test per chiarire.

```
$ python3
Python 3.6.9 (default, F
[GCC 8.4.0] on linux
Type "help", "copyright"
>>> a = 60
>>> b = 13
>>> print(a&b)
12
>>> exit()
```



```
a = 60
b = 13
```

```
print(a&b)
```

```
12
```

a = 0011 1100

b = 0000 1101

a&b = 0000 1100



Case structure

Basically a series of if-then-else with some constraint. In practice, the choice between the blocks of instruction is guided by the value of a certain variable:

```
switch variable: {  
    val1 0: "!= min";  
    case 1: "or inear";  
    case 2: "== text";  
    case 3: "== imast";  
    break;  
}
```

```
If (variable == val1) {  
    // Code for val1  
} else if (variable == 1) {  
    // Code for case 1  
} else if (variable == 2) {  
    // Code for case 2  
} else if (variable == 3) {  
    // Code for case 3  
}
```

CICLI

γΣΧΣ



α

α



Cicli

Esistono tre tipi diversi: while..do, do..while e for.

- Non tutti i linguaggi hanno necessariamente tutte e tre queste strutture.
- Queste strutture consentono di ripetere un blocco di istruzioni finché si verifica una condizione.
- Anche in questo caso è possibile annidare più cicli uno dentro l'altro.

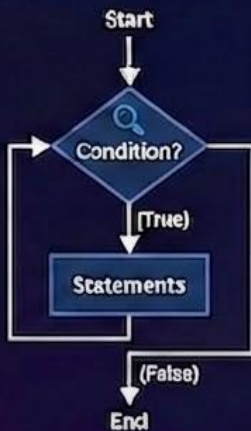


While..do e Do..while

WHILE..DO LOOP

The block of instructions can also never be executed, since the condition is checked at the beginning, as long as the condition condition is true, the block of instructions is executed

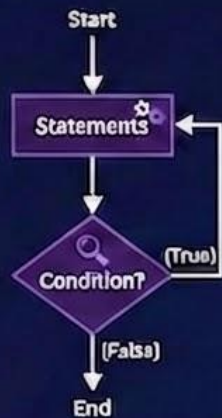
```
WHILE (condition)
    statements
END DO
```



DO..WHILE LOOP

do..while instead executes the block of instructions until the condition is false

```
DO
    statements
WHILE (condition)
```



Esempio

Un semplice esempio in C:

```
int i = 0, N = 10;
```

```
while (i != N)
```

```
{
```

```
    printf("%d \n", i);
```

```
    i++;
```

```
}
```

```
[redo@banquo csmall (master)]$ gcc -o whiledo whiledo.c
[redo@banquo csmall (master)]$ ./whiledo
0
1
2
3
4
5
6
7
8
9
[redo@banquo csmall (master)]$
```

Esempio

```
Type "help", "copyright"  
>>> while True:  
...     print("here")  
... 
```



```
while True:  
    print("here")
```



```
here  
here  
here  
here  
here  
here  
here  
here  
here  
here
```



Esempio

• `Int i = 0, N = 10;`

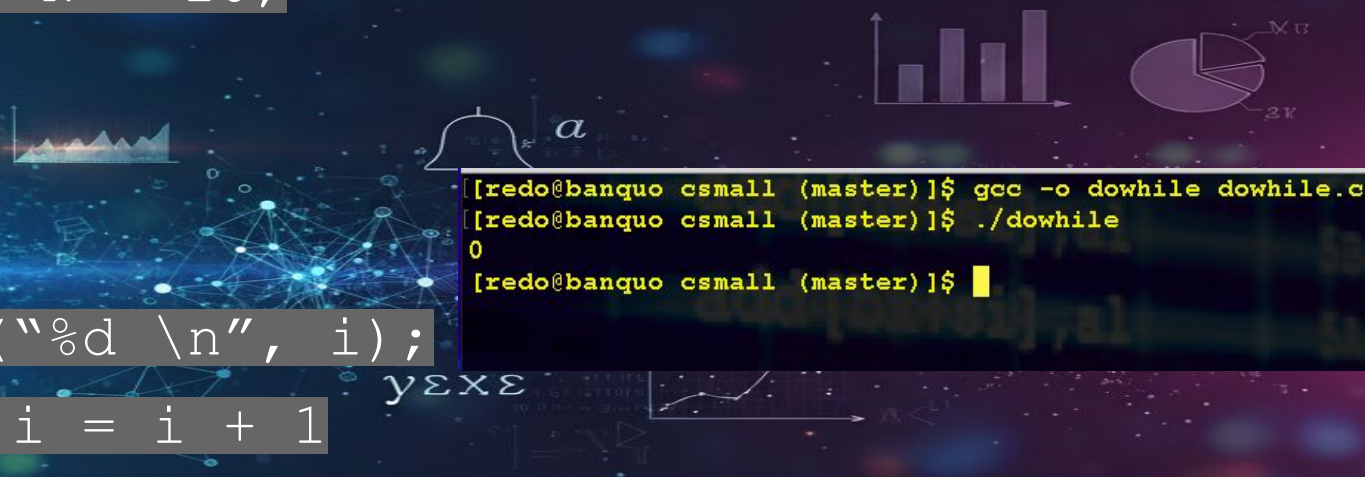
`do`

`{`

`printf ("%d \n", i);`

`i++; // i = i + 1`

`} while (i >= N);`



```
[redo@banquo csmall (master)]$ gcc -o dowhile dowhile.c
[redo@banquo csmall (master)]$ ./dowhile
0
[redo@banquo csmall (master)]$
```

For loop



- Executes a block of instructions a known number of times.
- Often uses an integer counter variable, changed step-by-step.
- End condition determined by comparing counter with a value.

```
for (counter = startingValue TO endValue STEP = stepValue) {  
    statements  
} // end for
```

Esempio

```
int i;  
for (i=0; i<10; ++i)  
{  
    printf ("%d \n", i);  
}
```

```
[redo@banquo csmall (master)]$ gcc -o forloop forloop.c  
[redo@banquo csmall (master)]$ ./forloop  
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
[redo@banquo csmall (master)]$
```

Esempio

```
for i in range(10):
```

```
    print(i)
```

```
$ python loop.py  
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```



Complessità algoritmica



Complessità algoritmica

- La complessità algoritmica misura la difficoltà di un calcolo, cioè algoritmo più input.
- La qualità di un algoritmo viene valutata in base al tempo e allo spazio necessari per la sua esecuzione, quindi in generale in base alle risorse richieste.
- Il tempo di esecuzione dipende dal tipo di input e dall'hardware usato, perciò non ha senso classificare gli algoritmi in base al numero di secondi richiesti.
- Il tempo di calcolo viene quindi espresso come numero di operazioni elementari in funzione della dimensione N dei dati di input.

Complessità algoritmica

- Esempio di calcolo dell'efficienza di un algoritmo che cerca il minimo m in un insieme di N numeri $\{x_1, x_2, \dots, x_N\}$.
- Affrontiamo il problema così:
 - Scegliamo x_1 come minimo iniziale.
 - Lo confrontiamo con x_2 , poi con x_3 e così via.
 - Se troviamo un x_i più piccolo, continuiamo i confronti con quello come abbiamo fatto con x_1 .
 - Alla fine avremo trovato il minimo.
- Per fare tutto avremo eseguito N confronti, quindi l'efficienza dell'algoritmo è direttamente proporzionale alla dimensione N dell'input.



Complessità algoritmica

- Quanti confronti facciamo per trovare il minimo?
- Passo 1: inizializzazione.
 - Impostiamo il primo elemento come minimo, una operazione.
- Passo 2: iterazione.
 - Scorriamo gli elementi rimanenti. Poiché in totale ci sono N elementi, ne controlliamo $N - 1$.
- Quindi il numero di confronti è esattamente $N - 1$.



Complessità algoritmica

find_minimum.py

```
def find_min(numbers):  
    # 1. Assume the first is the smallest  
    current_min = numbers[0]  
  
    # 2. Iterate through the rest  
    for i in range(1, len(numbers)):  
        # 3. Compare and update  
        if numbers[i] < current_min:  
            current_min = numbers[i]  
  
    # 4. Return the result  
    return current_min
```

Complessità algoritmica

Searching for the minimum in: [12, 5, 18, 3, 9]



min = 12

5 < 12
min = 5

18 > 5
skip

3 < 5
min = 3

9 > 3
skip



Complessità algoritmica

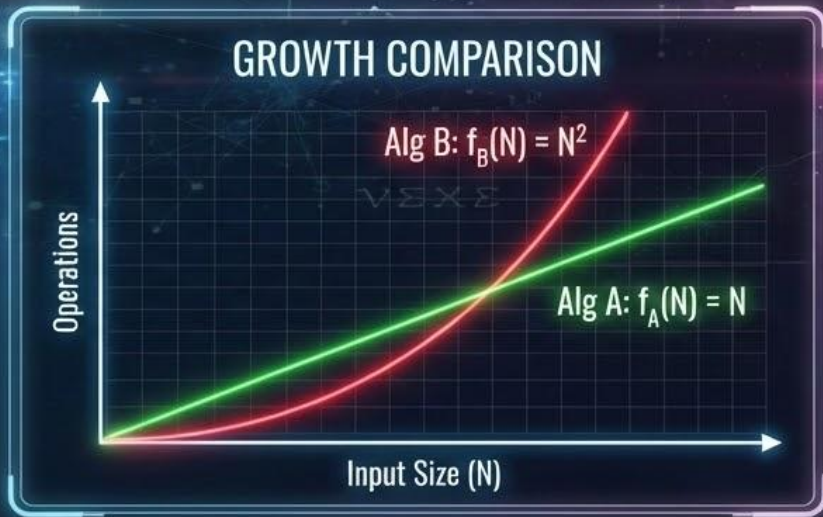
L'efficienza di un algoritmo può essere espressa come una funzione $f(N)$, quindi una funzione della variabile N che rappresenta la dimensione dei dati di input.

- **Questa funzione esprime il numero di operazioni elementari necessarie per risolvere il problema con l'algoritmo dato, in funzione della dimensione dell'input.**
- Rappresenta quindi la complessità computazionale.
- Dato N , un algoritmo A è più efficiente di un algoritmo B se, al crescere di N , $f_A(N)$ è minore o uguale a $f_B(N)$.
- **Il tempo di esecuzione di un programma dipende dalla complessità dell'algoritmo, dalla dimensione di N e dalla velocità della macchina su cui viene eseguito.**

Algorithmic complexity

Given input N, Algorithm A is more efficient than Algorithm B if, as N increases:

$$f_A(N) \leq f_B(N)$$



Complessità algoritmica

Per suddividere gli algoritmi in classi di complessità si usa il seguente criterio:

- Una funzione $f(N)$ è detta dell'ordine di $g(N)$ e si indica con $f(N) = O(g(N))$ se esiste una costante K tale che, salvo per un numero finito di valori di N , vale sempre la disuguaglianza $f(N) \leq K \cdot g(N)$. Si può anche scrivere $f(n) / g(N) \leq K$.
- **Per esempio, $2 \cdot N + 5 = O(N)$, infatti $2 \cdot N + 5 \leq 7 \cdot N$ per ogni N maggiore di zero.**



Algorithmic complexity

Prove that: $2N + 5 = O(N)$

We need to find a constant K where $2N + 5 \leq K * N$.

Let's assume $N \geq 1$.

If $N \geq 1$, then $5 \leq 5N$.

Therefore, $2N + 5 \leq 2N + 5N$.

$2N + 5 \leq 7N$.

The inequality is true for $K = 7$ for every $N > 0$.

P vs NP



Classe P, tempo polinomiale

Definizione

- P significa tempo polinomiale. Comprende tutti i problemi decisionali che possono essere risolti da una macchina di Turing deterministica usando una quantità polinomiale di tempo di calcolo.

Problemi “facili”

- Se un problema appartiene a P, è considerato trattabile o relativamente rapido da risolvere, anche per input grandi. Il tempo di esecuzione $f(n)$ cresce come $O(n^k)$.
 - Ordinare una lista di numeri.
 - Trovare il percorso più breve su una mappa.
 - Moltiplicare numeri grandi.

Classe NP, polinomiale non deterministica

Definizione

- NP significa tempo polinomiale non deterministico. Comprende problemi per i quali, data una possibile soluzione, è possibile verificarne la correttezza in tempo polinomiale. Equivalentemente, sono risolvibili da una macchina di Turing non deterministica in tempo polinomiale.
- “Difficili da risolvere, facili da verificare”
 - Pensa a un Sudoku. Risolverlo da zero è difficile. Ma se qualcuno ti consegna una griglia completa, verificare che tutte le righe e le colonne contengano i numeri da 1 a 9 è molto facile e rapido.
 - Sudoku e cruciverba.
 - Problema del commesso viaggiatore.

Decifrare password crittografiche.

Problemi NP-completi

I più difficili in NP

- I problemi NP-completi sono i più difficili della classe NP. Condividono una proprietà speciale: sono tutti collegati matematicamente.

È possibile ridurre qualsiasi problema NP a un problema NP-completo in tempo polinomiale. Quindi:

- Se uno è facile da risolvere, LO SONO TUTTI.
- Se uno è difficile da risolvere, LO SONO TUTTI.



Esempio: fattorizzazione in numeri primi

Il problema, un problema NP ma quasi certamente non NP-completo

- **Dato un intero N , trova i numeri primi a e b tali che $N = a \times b$.**

Perché appartiene a NP?

- **La verifica è immediata: se qualcuno afferma che $a = 13$ e $b = 17$ per $N = 221$, basta moltiplicarli.**
- **La soluzione è lenta: per un numero N di 500 cifre non si conosce alcun algoritmo rapido, cioè polinomiale, che trovi a e b partendo da zero.**

La domanda da un milione di dollari

$$P = NP ?$$

Il Clay Mathematics Institute offre 1.000.000 di dollari a chiunque riesca a dimostrare se $P = NP$ oppure $P \neq NP$.

Se è facile verificare che una soluzione sia corretta, è anche intrinsecamente facile trovarla partendo da zero?

La domanda da un milione di dollari

$$P \neq NP(\textit{Expected}) \quad P = NP(\textit{Miracle})$$



Se $P = NP$, la crittografia moderna verrebbe compromessa immediatamente. L'IA potrebbe risolvere con facilità la logistica perfetta, trovare cure per il cancro e ottimizzare tutto a livello globale. Semplicemente, non abbiamo ancora trovato l'algoritmo.



Complessità algoritmica

- Le classi di complessità P e NP: la questione se P sia uguale a NP oppure no è ancora aperta, ed è il problema da un milione di dollari del Clay Mathematics Institute.
- I problemi della classe P sono risolvibili da una macchina di Turing deterministica in tempo polinomiale.
- I problemi della classe NP non hanno un algoritmo di complessità polinomiale noto, ma possono essere verificati rapidamente.
- Problemi NP-completi: se uno di questi problemi è facile, lo sono tutti, poiché è possibile ridurre la soluzione di uno a un altro. Analogamente, se uno è difficile, lo sono tutti.
- Esempio: fattorizzare un intero in numeri primi è un problema NP, molto probabilmente non NP-completo. Dato c , trovare i fattori primi a e b tali che $a * b = n$.

