2. CHE LINGUA PARLA UN COMPUTER?

Il computer non comprende il linguaggio umano, ma lavora con un linguaggio macchina, noto come **codice binario**.
Il codice binario si basa su due soli simboli, **1** e **0**, che rappresentano lo stato di **accensione (1)** o **spegnimento (0)** dei circuiti elettrici all'interno del computer. Questo sistema di rappresentazione binaria è la lingua fondamentale dei computer. Gli algoritmi e i programmi vengono tradotti in istruzioni binarie che il computer può interpretare e eseguire.
Le complesse sequenze di 1 e 0, note come "linguaggio macchina", costituiscono il **codice sorgente** dei programmi.
I programmatori scrivono il codice in **linguaggi di programmazione** più comprensibili e poi un "interprete" traduce il codice sorgente in istruzioni binarie comprensibili per il computer.
La traduzione tra il linguaggio di programmazione umano e il linguaggio binario è ciò che consente agli esseri umani di comunicare le loro intenzioni e istruzioni ai computer e a questi di eseguirle per svolgere varie attività, dal calcolo matematico alla gestione di applicazioni.
Ad esempio per scrivere una semplice lettera del nostro alfabeto (bisogna distinguere inoltre tra minuscole e maiuscole) viene utilizzata una sequenza di ben 8 cifre di 0 e 1 (ad esempio la a = 01100001 oppure A = 01000001).
È chiaro, quindi, che dialogare nella lingua del computer è assolutamente impossibile per un essere umano perché anche parole semplici come "ciao" corrisponderebbero ad una sequenza impossibile da memorizzare (ciao = 01100011 01101001 01100001 01101111).

Il traduttore uomo-computer: il linguaggio di programmazione

Come detto, un linguaggio di programmazione è come un traduttore utilizzato per convertire il pensiero umano in codice macchina.
Esistono molti linguaggi che operano questa conversione e tra quelli più diffusi troviamo Python, C, C++, Java e molti altri.
Si tratta di sistemi diversi, specializzati per piattaforma (Windows, macOS, Android, UNIX) o per hardware, ma ognuno di essi presenta una serie di elementi comuni come il set di istruzioni, i dati da memorizzare, gli operatori che mettono in combinazione i dati per ottenere un determinato risultato.

Questi linguaggi vengono definiti di alto livello se simili al linguaggio umano e vengono utilizzati dagli sviluppatori o programmatori per comunicare con le macchine.
Utilizzano poche parole del linguaggio umano in inglese, facilmente memorizzabili, ma affinché il computer le possa comprendere è necessario un **compilatore**, una sorta di traduttore che li converte in linguaggio macchina.
I linguaggi di basso livello sono quelli più vicini al linguaggio macchina (codice binario), sono complessi e difficilmente interpretabili dall'uomo ma hanno il vantaggio di essere più veloci perché il calcolatore non ha bisogno del compilatore per eseguirli.
L'uso del linguaggio di programmazione più adatto dipende dal tipo di applicazione e dallo scopo del progetto. Ogni linguaggio ha le sue peculiarità, punti di forza e debolezze. La scelta del linguaggio appropriato è un aspetto importante nello sviluppo del software.

1. **Python, C, C++, Java e altri linguaggi**: questi sono alcuni dei linguaggi di programmazione più diffusi, ognuno con le proprie caratteristiche e utilizzi specifici. Python, ad esempio, è noto per la sua sintassi leggibile e amichevole per gli sviluppatori, mentre C e C++ sono linguaggi più "vicini all'hardware" che offrono un controllo più dettagliato sulla memoria e le operazioni a basso livello. Java è noto per la sua portabilità e viene spesso utilizzato per sviluppare applicazioni multipiattaforma.
2. **Linguaggi di alto livello vs. linguaggi di basso livello**: i linguaggi di alto livello sono progettati per essere simili al linguaggio umano, rendendo più facile per i programmatori scrivere codice comprensibile. Tuttavia, richiedono un "traduttore", noto come compilatore o interprete, per convertire il codice sorgente in istruzioni comprensibili per il computer. I linguaggi di basso livello, come il linguaggio Assembly, sono più vicini al linguaggio macchina e sono meno leggibili per gli esseri umani, ma possono essere più efficienti in termini di risorse e velocità di esecuzione.
3. **Compilatori e interpreti**: i compilatori traducono l'intero codice sorgente in linguaggio macchina in un'unica fase, creando un file eseguibile. Gli interpreti traducono il codice sorgente riga per riga e lo eseguono "al volo". Entrambi sono utilizzati per rendere il codice scritto in un linguaggio di alto livello eseguibile dal computer.

Un po' di storia

Prima degli Anni '40, l'unico modo per poter programmare, quindi per poter dialogare con un computer, era il linguaggio macchina. Operazione lunghissima e noiosa che solo programmatori specializzati riuscivano a fare.
I primi linguaggi di programmazione nascono dal 1950 tra cui i più importanti furono il FORtrAN (FORmula trANslator), che svolgeva solo calcoli matematici e scientifici, e l'ALGOL (ALGOrithmic Language).
Una loro evoluzione nel 1960 fu il COBOL (COmmon Business Oriented Language), ideato per l'organizzazione dei dati a cui seguì, nel 1964, il famosissimo Basic il cui nome deriva dalla sua semplicità perché portò i linguaggi di programmazione alla portata dei principianti.
Nel 1970 fa la sua comparsa il Pascal a cui fece seguito pochi anni dopo linguaggio C molto versatile nella rappresentazione dei dati e la cui forza risiedeva in una limitatezza di strumenti a disposizione. Pochi strumenti ma in grado di fare qualunque cosa e questo lo rese uno dei linguaggi più apprezzati dai programmatori.
La vera svolta si ebbe nel 1983 quando Bjarne Stroustrup inventò il C++, il primo linguaggio di programmazione Orientato ad Oggetti (Object Oriented); è il linguaggio attraverso il quale siamo arrivati oggi ad usare le finestre colorate dei nostri sistemi operativi.
La storia dei linguaggi di programmazione è un esempio di come l'informatica abbia continuamente evoluto e migliorato i mezzi attraverso cui gli esseri umani possono comunicare con i computer, rendendo la programmazione più accessibile e potente nel corso degli anni.

George Boole e la logica matematica

George Boole fu un matematico e logico britannico vissuto nel 1800 ed è considerato il padre della logica matematica. Ma perché parlando di programmazione è così importante questo personaggio? La motivazione è molto semplice: l'**algebra di Boole o booleana**, non si basa sui classici operatori di *addizione* o *sottrazione* ma sugli operatori logici **AND**, **OR** e **NOT**.
Questi sono perfetti per il modo in cui "ragiona" un computer. Le condizioni di "vero" o "falso" (1 o 0) trovano nell'algebra booleana una rappresentazione perfetta.

Ad esempio l'operatore **AND,** detto anche di **CONGIUNZIONE,** restituisce un valore vero solo se gli altri due valori sono veri, quindi:

1°VALORE	2°VALORE	RISULTATO
Falso	Falso	Falso
Falso	Vero	Falso
Vero	Falso	Falso
Vero	Vero	Vero

L'operatore **OR**, detto anche di **DISGIUNZIONE**, restituisce un valore vero solo nel caso in cui almeno uno dei due valori è vero, quindi:

1°VALORE	2°VALORE	RISULTATO
Falso	Falso	Falso
Falso	Vero	Vero
Vero	Falso	Vero
Vero	Vero	Vero

Infine, l'operatore **NOT**, detto anche di **NEGAZIONE**, è una semplice operazione di negazione per cui se il primo valore è FALSO essa restituisce il contrario, ossia VERO, mentre se il primo valore è VERO esso restituisce FALSO.

1°VALORE	RISULTATO
Falso	Vero
Vero	Falso

Grazie a questi tre semplici operatori, è possibile formalizzare il nostro pensiero realizzando complesse strutture e sui calcolatori è possibile formalizzare qualsiasi tipo di input.

Il pensiero umano e i diagrammi di flusso

Per rappresentare un algoritmo si può utilizzare un **diagramma di flusso**.
Un diagramma di flusso è uno schema composto **da blocchi** o forme geometriche che rappresentano le varie istruzioni e azioni, collegati da **frecce** che indicano l'ordine di esecuzione. Questi diagrammi semplificano la comprensione dell'algoritmo, rendendolo visivamente chiaro per i programmatori e gli sviluppatori di software.

Molti linguaggi di programmazione moderni incorporano l'uso di diagrammi di flusso che sono strumenti intuitivi che traducono il pensiero in algoritmi attraverso rappresentazioni grafiche.
Questa struttura offre due vantaggi principali: la possibilità di descrivere in modo logico il pensiero e la capacità di controllare attentamente il funzionamento del programma.
Ogni diagramma di flusso è composto da una serie di simboli che rappresentano azioni specifiche. Per poter utilizzare i diagrammi di flusso è necessario familiarizzare con essi.
La tabella di seguito descrive forma e significato.

NOME	SIMBOLO	AZIONE
Partenza		Azione che avvia il processo.
Fine		Azione che conclude il processo.
Documenti		Documenti di lavoro.
Ingresso/Uscita		Interazione con il mondo esterno. Azione che mostra il risultato ottenuto.
Collaborazione		Indica altre unità che collaborano allo svolgimento del processo.
Test/ Confronti		Divide il flusso di lavoro. In genere attende un input dal mondo esterno per eseguire una specifica azione.
Assegnazioni		Serve per assegnare dei valori o definire costanti.
Salti		Indica la direzione secondo cui si svolge una determinata azione.

Utilizzando questi simboli, rappresentare sotto forma di algoritmo una qualunque azione diventa possibile. Vediamo come descrivere la decisione di uscire di casa in base alle condizioni climatiche.

Le istruzioni condizionali nei diagrammi di flusso: la selezione

I diagrammi di flusso sono uno strumento visuale che aiuta a rappresentare in modo chiaro il percorso logico di un processo. Tra gli elementi chiave di un diagramma di flusso, le **istruzioni condizionali** svolgono un ruolo cruciale, consentendo al programma di prendere decisioni in base alle condizioni specificate.
Un'istruzione condizionale è un **blocco di codice** che permette al programma di eseguire azioni diverse a seconda delle condizioni definite. Nel contesto di un diagramma di flusso, ciò si traduce in una "decisione" o "selezione" che influenza il percorso del flusso. Nel linguaggio comune è rappresentata da un **SE** e da un **ALTRIMENTI**.
Il simbolo principale per rappresentare un'istruzione condizionale è il **rombo**. All'interno di questo rombo, vengono indicate le condizioni che il programma deve valutare.
Due percorsi escono dal rombo: uno per il caso in cui la condizione è vera e un altro per il caso in cui è falsa.

Vediamo un esempio.

Immagina di dover creare un programma che controlla se una persona è maggiorenne prima di consentirle l'accesso a una determinata area.
Il diagramma di flusso potrebbe apparire così.

1. **Inizio**: il flusso del programma inizia qui.
2. **Input Età**: il programma richiede all'utente di inserire la propria età.
3. **Condizione**: viene utilizzato un rombo per rappresentare l'istruzione condizionale. La condizione potrebbe essere: "Se l'età è maggiore o uguale a 18".
4. **Vero**: se la condizione è vera, il flusso si sposta lungo la freccia che porta al "Sì". Qui, potremmo avere l'azione "Accesso Consentito".
5. **Falso**: se la condizione è falsa, il flusso si sposta lungo la freccia che porta al "No". Qui, potremmo avere l'azione "Accesso Negato".
6. **Fine**: il programma termina qui.

Vediamo un altro esempio.
Immagina di dover creare un programma che verifica se un numero inserito dall'utente è positivo o negativo. Ecco come potrebbe apparire il diagramma di flusso.

1. **Inizio**: il flusso del programma inizia qui.
2. **Input Numero**: l'utente inserisce un numero.

3. **Condizione di Positività**: viene utilizzato un rombo per rappresentare l'istruzione condizionale. La condizione potrebbe essere: "Se il numero è maggiore di zero".
4. **Vero**: se la condizione è vera, il flusso si sposta lungo la freccia che porta al "Sì". Qui, potremmo avere l'azione "Il numero è positivo".
5. **Falso**: se la condizione è falsa, il flusso si sposta lungo la freccia che porta al "No". Qui, potremmo avere l'azione "Il numero è negativo o zero".
6. **Fine**: il programma termina qui.

Le istruzioni condizionali nei diagrammi di flusso sono uno strumento potente per gestire le decisioni nei programmi in modo chiaro e organizzato. Il rombo, con i suoi rami "Sì" e "No", fornisce una rappresentazione visiva intuitiva delle alternative possibili.

Le istruzioni cicliche nei diagrammi di flusso: l'iterazione

L'**iterazione** rappresenta il concetto di ripetere una serie di azioni o istruzioni fino a quando una condizione specifica è verificata.
In un diagramma di flusso, questo è spesso rappresentato da un "**ciclo**" che si ripete finché la condizione desiderata è valida.
Il simbolo principale per rappresentare l'iterazione è il ciclo o loop. Esistono diversi tipi di loop, ma uno dei più comuni è il "loop condizionale", dove le azioni vengono ripetute finché una condizione è vera.

Vediamo un esempio.

Immagina di dover creare un programma che chiede all'utente di inserire una serie di numeri e ne calcola la somma. Il diagramma di flusso potrebbe apparire così:

1. **Inizio**: il flusso del programma inizia qui.
2. **Inizializzazione Somma**: una variabile "somma" viene inizializzata a zero.
3. **Input Numero**: il programma chiede all'utente di inserire un numero.
4. **Aggiunta alla Somma**: il numero inserito viene aggiunto alla variabile "somma".
5. **Condizione di Continuazione**: viene utilizzato un ciclo per rappresentare l'iterazione. La condizione potrebbe essere: "Se l'utente vuole inserire un altro numero".
6. **Sì**: se la condizione è vera, il flusso si ripete dal punto 3.

7. **No**: se la condizione è falsa, il flusso prosegue al punto successivo.
8. **Stampa Risultato**: il programma stampa il risultato finale, che è la somma dei numeri inseriti.
9. **Fine**: il programma termina qui.

Vediamo un altro esempio.
Immaginiamo di dover creare un programma che chiede all'utente di indovinare un numero segreto. Il programma continuerà a chiedere all'utente di inserire un numero finché non indovina correttamente il numero segreto.
Ecco come potrebbe apparire il diagramma di flusso:

1. **Inizio**: il flusso del programma inizia qui.
2. **Generazione Numero Segreto**: il programma genera casualmente un numero segreto.
3. **Input Tentativo**: l'utente inserisce un numero nel tentativo di indovinare il numero segreto.
4. **Condizione di Indovinato**: viene utilizzato un ciclo per rappresentare l'iterazione. La condizione potrebbe essere: "Se il numero inserito non è uguale al numero segreto".
5. **Sì**: se la condizione è vera, il flusso ritorna al punto 3, chiedendo all'utente di inserire un nuovo tentativo.
6. **No**: se la condizione è falsa, il flusso prosegue al punto successivo.
7. **Messaggio di Congratulazioni**: il programma mostra un messaggio di congratulazioni, indicando che l'utente ha indovinato il numero segreto.
8. **Fine**: il programma termina qui.

Questo esempio illustra come l'iterazione, attraverso il ciclo condizionale, consenta al programma di continuare a chiedere all'utente di inserire un numero finché non viene indovinato correttamente il numero segreto.
L'iterazione nei diagrammi di flusso è un modo efficace di gestire situazioni in cui è necessario ripetere un insieme di azioni. Il ciclo fornisce una chiara rappresentazione visiva del processo iterativo.

Trovare e correggere gli errori di un programma: il *debugging*

Quando scriviamo un programma è importante fare una specie di controllo finale. Dobbiamo assicurarci che il nostro codice funzioni bene e che riesca a gestire tutte le situazioni possibili.
La sfida è che il programma possa essere eseguito più volte con input diversi.
Quindi, quando scriviamo le istruzioni per il computer, è difficile prevedere tutti i casi possibili fin dall'inizio. Spesso dobbiamo migliorare il programma dopo averlo scritto, guardando attentamente come funzionano le istruzioni, passo dopo passo.
Questa fase cruciale si chiama ***debugging***, che significa trovare e correggere gli errori nel codice.
Quando cerchiamo errori è interessante notare che spesso non si trovano nelle parti "semplici" del codice, dove le cose si svolgono una dietro l'altra; piuttosto, si nascondono nelle condizioni, come le scelte che il computer deve fare.
Immagina di scrivere un programma per ordinare una lista di libri in base al loro titolo. Ogni libro ha un autore, un genere e un anno di pubblicazione. Ora, quando scrivi il tuo programma, potrebbe sembrare che funzioni perfettamente, ma c'è un piccolo problema: non hai considerato i libri con titoli simili.
Dopo aver scritto il codice, inizi a eseguirlo con diverse liste di libri; noti che quando hai libri con titoli molto simili, il programma non li ordina correttamente: ecco dove entra in gioco il *debugging*.
Esaminando il flusso del tuo programma, ti rendi conto che il problema è legato alle condizioni che hai impostato per confrontare i titoli; forse hai usato una condizione troppo stretta, trascurando di considerare casi in cui i titoli sono simili ma non identici. Quindi, fai alcune modifiche al codice, aggiungendo una condizione più flessibile per gestire titoli simili.
Dopo questa correzione, il programma riesce a ordinare correttamente i libri anche quando i titoli sono leggermente diversi.
In sostanza, fare il *debugging* è come cercare di rendere il programma più intelligente e capace di gestire tutte le sfumature delle situazioni che potrebbe incontrare.