

Università degli Studi di Padova

Corso di laurea triennale in Ingegneria informatica

Appunti delle lezioni

Fondamenti di Informatica

Codice insegnamento

IN0000225

Canale

B

Docente

Prof. Sergio Canazza Targon

Anno accademico

2026–2027

Anno di corso

1

Semestre

1

Autore

Simone Siega

Data

29 settembre 2026

Versione

0.1.0

Materiale didattico non ufficiale, non approvato né pubblicato dall'Università degli Studi di Padova.

Opera distribuita con licenza CC BY-SA 4.0.

Indice dei contenuti

1	Introduzione all'informatica	1
1.1	Informazione, calcolo e programmazione	1
1.2	Algoritmi e risoluzione dei problemi	2
1.3	La macchina di Turing e i limiti del calcolo	4
1.4	Architettura del calcolatore ed esecuzione delle istruzioni	6
1.5	Memoria, archiviazione e dispositivi di ingresso e uscita	9
1.6	Linguaggi di programmazione e forme di esecuzione	10
2	Rappresentazione delle informazioni	13
2.1	Sistemi di numerazione posizionali	13
2.2	Notazione binaria e conversioni	14
2.3	Numeri naturali e rappresentazione con modulo e segno	15
2.4	Complemento a due	16
2.5	Aritmetica intera e overflow	18
2.6	Virgola fissa e virgola mobile	19
2.7	Precisione e arrotondamento	22
2.8	Notazione esadecimale e ottale	23
2.9	Rappresentazione dei caratteri	24
3	Introduzione a Python	26
3.1	Python e l'esecuzione dei programmi	26
3.2	Variabili, tipi di dato e input/output	27
3.3	Logica booleana e istruzioni condizionali	28
3.4	Cicli e iterazioni	29
3.5	Liste	30
3.6	Funzioni	31
3.7	Gestione dei file di testo	32
4	Ricorsione	34
4.1	Funzioni ricorsive e terminazione	34
4.2	Il fattoriale: iterazione e ricorsione	34
4.3	Invocazioni sospese e pila delle chiamate	35
4.4	Ricorsione in coda e limiti di Python	36
4.5	Ricorsione multipla e numeri di Fibonacci	37
4.6	Esercizi di riepilogo	39

1 Introduzione all'informatica

L'informatica studia il trattamento automatico dell'informazione e i processi di calcolo che lo rendono possibile. Il suo oggetto non è soltanto il calcolatore come dispositivo fisico, ma anche la formulazione dei problemi, la progettazione dei procedimenti risolutivi e i limiti della loro esecuzione. Per comprendere come un programma trasformi i dati in risultati occorre distinguere il problema da risolvere, il metodo scelto e la macchina che lo esegue.¹

1.1 Informazione, calcolo e programmazione

Un *dato* è una rappresentazione utilizzabile per conservare, trasmettere o elaborare informazione. Il suo significato dipende dalle convenzioni adottate: una sequenza di simboli può descrivere, per esempio, un importo oppure un testo. L'elaborazione opera sulle rappresentazioni, mentre l'interpretazione dei risultati deve rimanere coerente con il problema originario. Le convenzioni per rappresentare numeri e caratteri sono sviluppate nel [capitolo 2](#).

Il significato di un dato non coincide, tuttavia, con la quantità di informazione che esso può trasmettere. Nel modello introdotto da Claude Shannon, la quantità di informazione associata a un evento dipende dalla sua probabilità, indipendentemente dal significato del messaggio. Un evento poco probabile fornisce, quando si verifica, una quantità di informazione maggiore rispetto a uno più probabile. Per un esito di probabilità $0 < p \leq 1$, il contenuto informativo è

$$I(p) = -\log_2 p.$$

Il logaritmo in base due esprime questa quantità in *bit*: un esito fra due alternative equiprobabili fornisce un bit, mentre un esito certo fornisce zero bit. Per esempio, conoscere l'esito di un esperimento con quattro risultati equiprobabili fornisce $-\log_2(1/4) = 2$ bit. Per esiti con probabilità $p_1, \dots, p_r$, la cui somma è uno, l'*entropia di Shannon* misura il contenuto informativo medio:

$$H = -\sum_{i=1}^r p_i \log_2 p_i,$$

con contributo nullo per gli esiti di probabilità zero. Il bit come unità di informazione non coincide concettualmente con una *cifra binaria*: quest'ultima è un simbolo 0 o 1. Una cifra binaria aleatoria ha entropia di un bit solo se i due valori sono equiprobabili; se il valore è già noto, la sua entropia è nulla.²

L'elaborazione automatica dei dati è affidata a un *computer*, o *calcolatore*, ossia un dispositivo fisico programmabile che memorizza dati, esegue istruzioni e scambia informazioni con l'esterno. Una *computazione* è invece un processo di trasformazione dei dati secondo regole specificate. Il dispositivo e il processo sono concetti distinti: lo stesso calcolatore può eseguire computazioni diverse, e lo stesso procedimento può essere realizzato su macchine differenti. Le componenti fisiche costituiscono l'*hardware*, mentre i programmi costituiscono il *software*.

Per stabilire quale elaborazione debba essere eseguita, occorre anzitutto definire il *problema di calcolo*, specificando i dati ammissibili in ingresso e il risultato richiesto per ciascuno di essi. Una particolare scelta dei dati costituisce un'*istanza* del problema. La specifica chiarisce che cosa ottenere, ma non come ottenerlo: il

¹Riferimento didattico: S. Canazza, *Presentazione del corso*, file 2026_FI_00_canazza.pdf.

²C. E. Shannon, *A Mathematical Theory of Communication*, *Bell System Technical Journal*, 1948, parte I: misura dell'informazione ed entropia per sorgenti discrete.

procedimento risolutivo è un *algoritmo*, mentre un *programma* ne esprime una realizzazione in un linguaggio di programmazione. Uno stesso algoritmo può avere più implementazioni, differenti per linguaggio o organizzazione interna, senza modificare il problema risolto.

La distinzione fra problema, algoritmo e programma permette di precisare il ruolo della *programmazione*, che comprende la progettazione, l'implementazione e la verifica dei programmi rispetto alle specifiche. Essa richiede di individuare rappresentazioni dei dati e procedimenti risolutivi appropriati, traducendoli in istruzioni conformi al linguaggio utilizzato. Il calcolatore esegue tali istruzioni secondo le regole del linguaggio e dell'ambiente di esecuzione, senza correggerle autonomamente sulla base delle intenzioni del programmatore.

A livello della macchina, anche i procedimenti più articolati si riconducono a operazioni elementari, come trasferimenti di dati, calcoli, confronti e scelte della prossima istruzione. La loro combinazione mediante sequenze, selezioni fra percorsi alternativi e ripetizioni permette di realizzare elaborazioni complesse a partire da un insieme limitato di operazioni fondamentali.

1.2 Algoritmi e risoluzione dei problemi

La descrizione di un metodo deve precisare tanto le operazioni quanto il loro ordine e le condizioni che determinano il proseguimento del calcolo. Un'indicazione come «ripetere finché il risultato è adeguato» non è sufficiente se non viene stabilito un criterio verificabile di adeguatezza. Analogamente, chiedere di individuare il paesaggio «più rilassante» richiede prima di definire un criterio preciso di confronto.

Definizione 1.1 (Algoritmo per un problema di calcolo). Un algoritmo è una descrizione finita e non ambigua di un procedimento, composto da passi effettivamente eseguibili, che termina dopo un numero finito di passi per ogni ingresso del dominio previsto. Per risolvere correttamente un problema, il procedimento deve inoltre produrre il risultato prescritto per ciascun ingresso ammissibile.

L'*eseguibilità* richiede che ogni passo possa essere svolto mediante operazioni definite, senza richiedere informazioni non disponibili o procedure di cui non sia specificato il funzionamento. L'*assenza di ambiguità* riguarda sia il significato delle operazioni sia la scelta del passo successivo. La *terminazione* riguarda ogni ingresso ammissibile: una descrizione di poche istruzioni può imporre una ripetizione senza fine, quindi la finitezza del testo non assicura la finitezza dell'esecuzione.

La terminazione non impone lo stesso numero di passi per tutti gli ingressi, né garantisce che il risultato sia corretto. Correttezza e terminazione devono essere giustificate rispetto alle ipotesi del problema. Un programma che implementa un algoritmo deve inoltre rispettarne tali ipotesi; errori nella traduzione o limiti della rappresentazione dei dati possono alterare il risultato.

Un semplice problema numerico permette di esaminare insieme terminazione e correttezza di un procedimento iterativo.

Esempio 1.1 (Raddoppio di un capitale). Si vuole determinare il primo anno in cui un capitale di 20 000 euro raggiunge almeno 40 000 euro, con interesse annuo costante del 5% capitalizzato a fine anno. Si escludono versamenti, prelievi, imposte e arrotondamenti intermedi. Il procedimento è il seguente:

1. inizializzare il numero di anni $n = 0$ e il saldo $S = 20\,000$;
2. finché $S < 40\,000$, sostituire S con $1,05S$ e aumentare n di uno;
3. restituire n .

Dopo n ripetizioni il saldo è $S_n = 20\,000(1,05)^n$: ogni passo aggiunge al saldo il 5% del suo valore corrente.

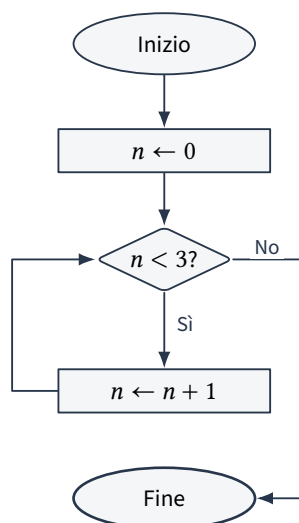
Finché la soglia non viene raggiunta, il saldo è almeno 20 000 euro e aumenta di almeno 1 000 euro all'anno. La soglia viene quindi raggiunta in non più di venti ripetizioni, garantendo la terminazione. Poiché la condizione viene controllata prima di ogni aggiornamento, il valore restituito corrisponde al primo anno in cui il capitale raggiunge la soglia. In particolare,

$$S_{14} \approx 39\,598,63 < 40\,000, \quad S_{15} \approx 41\,578,56 > 40\,000,$$

e il risultato è 15 anni. Le approssimazioni mostrate non sostituiscono l'ipotesi di calcolo esatto del modello.

L'**esempio 1.1** mostra come sia possibile descrivere un procedimento e giustificarne terminazione e correttezza senza ricorrere a un particolare linguaggio di programmazione. La sua implementazione richiede invece di tradurre inizializzazione, confronto, aggiornamento e ripetizione nelle istruzioni previste dal linguaggio scelto.

Il controllo di un algoritmo può anche essere rappresentato mediante un diagramma di flusso. Quello seguente illustra un semplice conteggio.



Un *diagramma di flusso* rappresenta graficamente il controllo di un algoritmo: ovali indicano inizio e fine, rettangoli le operazioni, rombi le decisioni e frecce l'ordine dei passi. Le uscite da una decisione corrispondono ai possibili esiti della condizione; una freccia che ritorna a un passo precedente indica una ripetizione. Il diagramma descrive quindi l'ordine delle operazioni, ma non costituisce di per sé una prova della loro correttezza o terminazione.

Una volta precisato come descrivere un algoritmo e verificarne le proprietà, occorre distinguere due questioni più generali: l'esistenza di una soluzione algoritmica e le risorse necessarie per eseguirla.

Un problema è *calcolabile* se esiste un procedimento algoritmico che lo risolve per tutti gli ingressi ammissibili. La calcolabilità riguarda quindi l'esistenza del procedimento, indipendentemente dal fatto che esso sia già noto. L'*efficienza* riguarda invece le risorse richieste dalla soluzione, soprattutto tempo e memoria, al variare dei dati. La *complessità computazionale* studia queste risorse in funzione della dimensione dell'ingresso, secondo un modello di calcolo specificato; non coincide necessariamente con il tempo misurato su un particolare computer.

Anche un algoritmo che termina correttamente può richiedere risorse troppo grandi per un impiego pratico. La possibilità teorica di risolvere un problema e quella di farlo con le risorse disponibili sono dunque questioni distinte.

1.3 La macchina di Turing e i limiti del calcolo

La questione della calcolabilità, introdotta nella [sezione 1.2](#), richiede un modello preciso del significato di «procedimento eseguibile». Nel contesto delle ricerche sui fondamenti della matematica promosse da Hilbert, il problema della decisione chiedeva un metodo generale per stabilire la validità delle formule della logica del primo ordine. Nel 1936 Church e Turing ottennero, con approcci diversi, risultati negativi su questa possibilità. Turing introdusse un modello di calcolo astratto, oggi chiamato *macchina di Turing*, che permette di studiare il calcolo indipendentemente dalla tecnologia di una macchina fisica.³

Nella versione deterministica qui considerata, la macchina dispone di un nastro diviso in celle, ciascuna contenente un simbolo di un alfabeto finito che comprende un simbolo di bianco. Il nastro è teoricamente illimitato: non viene imposto un limite alla memoria utilizzabile. L'ingresso occupa inizialmente un tratto finito, mentre le altre celle sono bianche. Una testina legge e scrive una cella alla volta e può spostarsi alla cella adiacente a destra o a sinistra.

Il controllo della macchina ha un insieme finito di *stati*, fra cui uno iniziale e uno di arresto. Le *regole di transizione* stabiliscono, in base allo stato corrente e al simbolo letto, quale simbolo scrivere, dove spostare la testina e quale stato assumere. Per ogni coppia di stato e simbolo per la quale è definita una regola, l'azione è determinata senza ambiguità. Scrivere può anche significare lasciare invariato il simbolo della cella.

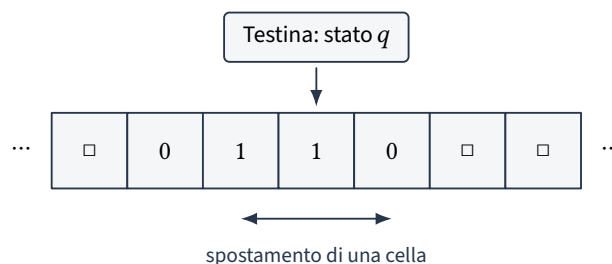


Figura 1.1. Un tratto del nastro e la posizione della testina. Il simbolo □ indica il bianco; q è lo stato corrente.

La [figura 1.1](#) mostra soltanto una parte del nastro, non un limite alla sua estensione. Una *configurazione* completa comprende il contenuto del nastro, la posizione della testina e lo stato del controllo. L'applicazione di una regola trasforma una configurazione nella successiva; una computazione è la successione di questi passi. Quando viene raggiunto lo stato di arresto, il risultato si ricava dal nastro secondo la convenzione scelta. La macchina può anche continuare indefinitamente. La finitezza dell'insieme degli stati di controllo non impone un limite alla lunghezza del nastro; ogni esecuzione che termina, tuttavia, visita soltanto un numero finito di celle.

Una *macchina universale di Turing* riceve una descrizione codificata di un'altra macchina e il suo ingresso, e ne simula l'esecuzione. Le regole della macchina universale rimangono fisse: cambiando la descrizione sul nastro cambia il procedimento simulato. Il programma può quindi essere trattato come un dato. Universalità significa capacità di simulare ogni macchina del modello, non capacità di risolvere qualsiasi problema concepibile, né garanzia che la simulazione termini.

³Per il modello, il contesto storico, l'universalità e il problema dell'arresto: L. De Mol, *Turing Machines*, *Stanford Encyclopedia of Philosophy*, §§ 1–2 e 5.

La *tesi di Church–Turing* identifica la nozione intuitiva di procedimento effettivo con la calcolabilità mediante macchina di Turing. È una tesi sul significato del calcolo algoritmico, non un teorema secondo cui ogni problema sarebbe risolvibile. Il modello fornisce così un riferimento comune per distinguere problemi calcolabili e problemi che non ammettono una soluzione algoritmica generale.

Fra questi rivestono particolare importanza i *problemi di decisione*, che richiedono una risposta fra due alternative, convenzionalmente «sì» e «no». Un problema di decisione è *decidibile* se esiste un algoritmo che termina per ogni ingresso ammissibile e fornisce sempre la risposta corretta; in caso contrario è *indecidibile*. La decidibilità riguarda quindi l'esistenza di un procedimento generale, non la possibilità di risolvere una singola istanza.

Un esempio fondamentale è il *problema dell'arresto*: stabilire se un programma, eseguito su un determinato ingresso, terminerà oppure continuerà indefinitamente. Non basta osservare che una particolare esecuzione termina; per decidere il problema occorrerebbe un procedimento capace di fornire una risposta corretta in entrambi i casi, per qualunque programma e ingresso.

Proposizione 1.1 (Indecidibilità del problema dell'arresto). *Non esiste un algoritmo che, ricevuti la descrizione di una macchina di Turing arbitraria e un suo ingresso, termini sempre e stabilisca correttamente se quella macchina si arresterà su quell'ingresso.*

Dimostrazione (per assurdo). Si supponga che esista un algoritmo, chiamato *osservatore*, capace di stabilire se un qualsiasi programma terminerà su un dato ingresso. Indichiamo con $O(p, x)$ il risultato dell'osservatore, dove p è la descrizione codificata di un programma e x è il suo ingresso. Per ipotesi, l'osservatore termina sempre e restituisce

$$O(p, x) = \begin{cases} 1 & \text{se il programma descritto da } p \text{ termina su } x, \\ 0 & \text{se il programma descritto da } p \text{ non termina su } x. \end{cases}$$

L'ipotesi è più forte della semplice simulazione: l'osservatore deve rispondere in tempo finito anche quando il programma esaminato non si arresta.

Utilizzando l'osservatore, si costruisce un nuovo programma D che riceve come ingresso la descrizione p di un programma. Il programma D chiede all'osservatore se il programma descritto da p , eseguito sulla propria descrizione p , terminerà. Successivamente si comporta in modo opposto alla previsione:

$$D(p) : \begin{cases} \text{esegue un ciclo senza fine} & \text{se } O(p, p) = 1, \\ \text{termina immediatamente} & \text{se } O(p, p) = 0. \end{cases}$$

La costruzione è possibile sotto l'ipotesi iniziale: D utilizza l'osservatore come parte del proprio procedimento e sceglie quale comportamento adottare in base alla risposta ricevuta.

Poiché anche D è un programma, possiede una descrizione codificata. Indichiamola con d e consideriamo l'esecuzione $D(d)$, cioè il programma D applicato alla propria descrizione. Si presentano due possibilità:

1. Se $O(d, d) = 1$, l'osservatore prevede che $D(d)$ termini. Per costruzione, però, D esegue un ciclo senza fine, contraddicendo la previsione.
2. Se $O(d, d) = 0$, l'osservatore prevede che $D(d)$ non termini. Per costruzione, invece, D termina immediatamente, contraddicendo nuovamente la previsione.

Entrambe le risposte possibili conducono a una contraddizione. L'ipotesi dell'esistenza di un osservatore algoritmico sempre terminante e corretto è quindi falsa: il problema dell'arresto non è decidibile in generale.

La contraddizione nasce dalla possibilità di trattare la descrizione di un programma come un dato, consentendo di applicare D alla propria descrizione. Non è necessario che il programma modifichi il proprio codice durante l'esecuzione.

Questo risultato non impedisce di stabilire la terminazione di programmi particolari. Se un programma termina, per esempio, una simulazione completa permette di verificarlo. Se invece continua indefinitamente, la sola simulazione non permette di concludere in un tempo finito che non terminerà mai. La [proposizione 1.1](#) esclude un algoritmo capace di risolvere correttamente tutti i casi possibili, non la possibilità di analizzarne singoli casi mediante altri ragionamenti.

Il problema dell'arresto mostra dunque un limite intrinseco del calcolo algoritmico, indipendente dalla potenza della macchina utilizzata. Il nastro illimitato della macchina di Turing è infatti un'idealizzazione: un computer reale dispone di memoria finita, tempi di esecuzione limitati e risorse fisiche. Può simulare passi del modello entro le risorse disponibili, ma non coincide letteralmente con una macchina a memoria illimitata. Un limite di calcolabilità non va pertanto confuso con un'insufficienza di memoria o con un tempo di esecuzione eccessivo.

1.4 Architettura del calcolatore ed esecuzione delle istruzioni

Il modello della macchina di Turing chiarisce i limiti teorici del calcolo; per comprendere come i programmi vengano eseguiti dai calcolatori reali occorre invece esaminarne l'architettura. Questa descrive l'organizzazione delle componenti hardware e le modalità con cui cooperano. Tra i modelli classici si distinguono l'*architettura di von Neumann* e l'*architettura Harvard*.

Nel modello di von Neumann, la CPU, la memoria centrale e i dispositivi di ingresso e uscita comunicano attraverso collegamenti condivisi. Istruzioni e dati risiedono nella stessa memoria e utilizzano il medesimo percorso di accesso. Il *principio del programma memorizzato* prevede che anche le istruzioni siano conservate in memoria, consentendo di cambiare il programma eseguito senza modificare i circuiti della macchina. Tale principio non è esclusivo dell'architettura di von Neumann.

L'architettura Harvard distingue invece le memorie e i percorsi di accesso destinati alle istruzioni da quelli riservati ai dati, permettendo di effettuare i due tipi di accesso contemporaneamente. Le architetture moderne possono combinare caratteristiche di entrambi i modelli: per esempio, molti processori dispongono di uno spazio di indirizzamento unificato e di cache separate per istruzioni e dati.

La CPU (*Central Processing Unit*) preleva ed esegue le istruzioni. Al suo interno, l'*unità di controllo* interpreta le istruzioni e coordina le operazioni richieste, mentre l'*unità logico-aritmetica*, o ALU (*Arithmetic Logic Unit*), esegue operazioni aritmetiche e logiche, come somme e confronti. I *registri* sono piccole memorie interne alla CPU, rapidamente accessibili, che conservano temporaneamente dati, indirizzi e altre informazioni necessarie all'esecuzione.

La figura 1.2 rappresenta i principali blocchi funzionali del modello di von Neumann.

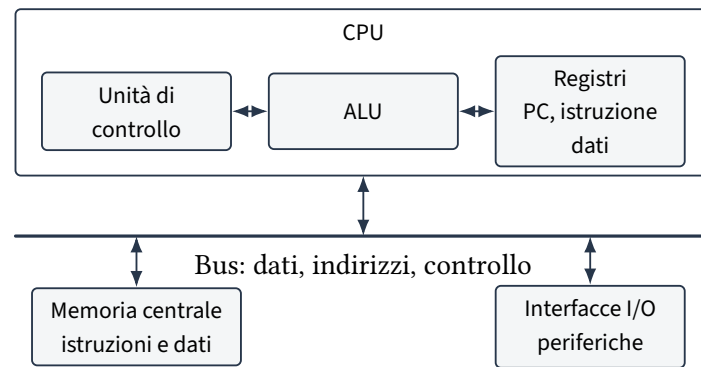


Figura 1.2. Organizzazione funzionale di von Neumann. Il collegamento condiviso comprende segnali di dati, indirizzi e controllo; le frecce indicano lo scambio complessivo, non la direzione di ogni segnale.

Le comunicazioni fra le componenti sono rappresentate mediante *bus*, ossia canali che consentono il trasferimento di informazioni secondo regole condivise. Si distinguono funzionalmente:

- il *bus dei dati*, che trasferisce dati e istruzioni;
- il *bus degli indirizzi*, che identifica la locazione di memoria o il dispositivo interessato;
- il *bus di controllo*, che trasporta i segnali necessari a coordinare operazioni come lettura e scrittura.

La distinzione riguarda la funzione dei segnali e non implica necessariamente la presenza di tre collegamenti fisici separati. Per leggere un valore dalla memoria, per esempio, si specifica l'indirizzo, si richiede l'operazione mediante i segnali di controllo e si trasferisce il risultato attraverso il canale dei dati.

La condivisione dei collegamenti semplifica l'organizzazione del sistema, ma introduce un limite alle prestazioni: più richieste possono competere per la stessa risorsa. La *banda*, cioè la quantità di dati trasferibile per unità di tempo, è limitata, e la CPU può dover attendere il completamento degli accessi alla memoria. Questo fenomeno è alla base del cosiddetto *collo di bottiglia di von Neumann*. Il modello rappresentato è una descrizione funzionale; i calcolatori moderni possono utilizzare interconnessioni più articolate.

Il registro *program counter* (PC) contiene l'indirizzo della prossima istruzione da prelevare. Nel modello didattico, il *registro istruzione* conserva l'istruzione corrente durante la sua interpretazione. L'esecuzione segue un ciclo fondamentale, articolato nelle seguenti fasi:

1. *fetch*: l'istruzione individuata dal PC viene prelevata dalla memoria;
2. *decode*: l'unità di controllo identifica l'operazione e gli operandi richiesti;
3. *execute*: vengono svolte le operazioni previste dall'istruzione, utilizzando gli operandi necessari;
4. *write-back*: quando previsto, il risultato viene scritto nella destinazione indicata, per esempio un registro o la memoria.

Durante l'esecuzione sequenziale, il PC viene aggiornato all'indirizzo dell'istruzione successiva, tenendo conto della sua lunghezza. Un'istruzione di salto può invece modificarlo, eventualmente in base all'esito di una condizione, permettendo di realizzare selezioni e cicli. Non tutte le istruzioni richiedono una scrittura del risultato.

Il *clock* è un segnale periodico che scandisce il funzionamento delle componenti sincrone. La sua frequenza, misurata in hertz, indica il numero di cicli al secondo. Un ciclo di clock non coincide necessariamente con l'esecuzione completa di un'istruzione: la frequenza da sola non determina quindi le prestazioni di un processore.

Per migliorare le prestazioni si può sfruttare il *parallelismo a livello di istruzioni* (ILP, *Instruction-Level Parallelism*), sovrapponendo l'elaborazione di istruzioni diverse.

Nella *pipeline*, l'esecuzione viene suddivisa in stadi. Mentre un'istruzione attraversa uno stadio, altre istruzioni possono occupare gli stadi precedenti o successivi. Una suddivisione didattica a cinque stadi comprende prelievo, decodifica, recupero degli operandi, esecuzione e scrittura del risultato, come illustrato nella [figura 1.3](#).

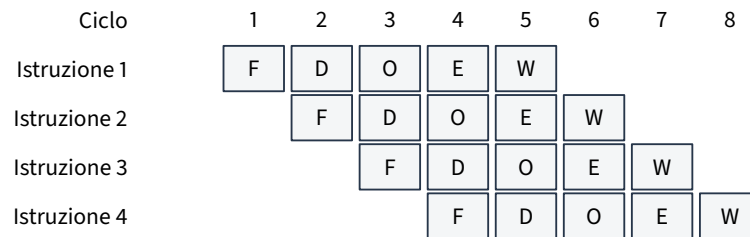


Figura 1.3. Pipeline ideale a cinque stadi, ciascuno della durata di un ciclo, senza attese: F = prelievo, D = decodifica, O = recupero operandi, E = esecuzione, W = scrittura. Dal quinto ciclo viene completata un'istruzione per ciclo finché la pipeline rimane alimentata.

La pipeline può aumentare il numero di istruzioni completate per unità di tempo, senza necessariamente ridurre la durata di una singola istruzione. Dipendenze fra istruzioni e attese per le risorse possono inoltre impedire il funzionamento al ritmo ideale.

Un processore *superscalare* dispone di più unità funzionali e può avviare più istruzioni nello stesso ciclo, purché le dipendenze e le risorse disponibili lo consentano. La pipeline sovrappone stadi di istruzioni diverse, mentre l'organizzazione superscalare consente di sfruttare più unità funzionali nello stesso intervallo di tempo. Le due tecniche possono coesistere.

Un'ulteriore forma di parallelismo riguarda l'impiego di più *core* o processori. Un processore *multicore* integra più core capaci di eseguire flussi di istruzioni; un sistema multiprocessore comprende più processori, eventualmente con memoria condivisa. Nei *multicalcolatori*, invece, nodi dotati di memoria propria comunicano mediante messaggi. In tutti questi casi, l'accelerazione dipende dalla possibilità di distribuire il lavoro e dai costi di coordinamento: aggiungere unità di calcolo non rende automaticamente più veloce un procedimento sequenziale.

La *GPU* (*Graphics Processing Unit*), originariamente sviluppata per la grafica, è adatta anche all'elaborazione parallela di grandi quantità di dati. Rispetto a una CPU, progettata per gestire in modo efficiente flussi di controllo complessi e operazioni con bassa latenza, una GPU privilegia l'esecuzione di numerose operazioni simili su dati differenti.

Elaborazione di immagini, operazioni su matrici e simulazioni sono esempi di applicazioni che possono beneficiare di questa organizzazione. Il vantaggio dipende tuttavia dalla quantità di lavoro parallelizzabile, dalle dipendenze fra operazioni e dai trasferimenti dei dati. Le unità di calcolo di una GPU non sono direttamente equivalenti ai core di una CPU.

CUDA (*Compute Unified Device Architecture*) è una piattaforma NVIDIA per la programmazione del calcolo su GPU. In questo modello la CPU, detta *host*, coordina il programma e avvia sulla GPU, detta *device*, funzioni di calcolo chiamate *kernel*.

L'esecuzione di un kernel coinvolge numerosi *thread*, organizzati in *blocchi*; l'insieme dei blocchi costituisce una *griglia*. Il modello SIMT (*Single Instruction, Multiple Threads*) permette a gruppi di thread di eseguire istruzioni su dati differenti. Questo non implica che tutti i thread vengano eseguiti fisicamente nello stesso istante. CPU e GPU cooperano nell'elaborazione: CUDA è una piattaforma di programmazione, non un algoritmo né l'unica tecnologia disponibile per il calcolo su GPU.⁴

⁴NVIDIA, *CUDA Programming Guide*, § 1.2, *Programming Model*, in particolare sistemi eterogenei, blocchi, griglie e SIMT.

1.5 Memoria, archiviazione e dispositivi di ingresso e uscita

L'esecuzione delle istruzioni dipende anche dal modo in cui dati e programmi vengono memorizzati e trasferiti. Nel contesto della memorizzazione, un *bit* indica una cifra binaria, con valori convenzionali 0 e 1; un *byte* è un gruppo di otto bit. Il bit come unità di informazione, introdotto nella [sezione 1.1](#), è un concetto distinto. Queste unità descrivono la quantità di dati memorizzati, non il loro significato, che dipende dalle convenzioni illustrate nel [capitolo 2](#).

La memoria è organizzata in *locazioni*, ciascuna identificata da un *indirizzo*. Un'operazione di lettura ne restituisce il contenuto, mentre una scrittura lo sostituisce con il dato fornito. Nelle architetture indirizzabili per byte, ogni indirizzo identifica un byte, anche se una singola operazione può trasferirne più di uno. L'unità minima indirizzabile non coincide quindi necessariamente con la quantità di dati trasferiti.

Lo *spazio di indirizzamento* è l'insieme degli indirizzi disponibili. Se gli indirizzi sono rappresentati mediante a bit, si possono distinguere al più 2^a locazioni. Nel caso di memoria indirizzabile per byte, ciò corrisponde a 2^a byte: con sedici bit si ottengono 2^{16} byte, ossia 64 KiB, mentre con trentadue bit si raggiungono 2^{32} byte, equivalenti a 4 GiB.

La capacità teoricamente indirizzabile non coincide necessariamente con la RAM installata: alcune regioni dello spazio di indirizzamento possono essere riservate ad altri dispositivi o non essere utilizzabili. Inoltre, la dimensione dei registri di un processore non determina necessariamente il numero di bit effettivamente impiegati per gli indirizzi di memoria.

Per esprimere le capacità occorre distinguere i prefissi decimali da quelli binari. Il simbolo B indica il byte: 1 kB corrisponde a 10^3 B, 1 MB a 10^6 B e 1 GB a 10^9 B. I prefissi binari utilizzano invece potenze di due: 1 KiB vale 2^{10} B, 1 MiB vale 2^{20} B e 1 GiB vale 2^{30} B. Le due convenzioni non sono intercambiabili e devono essere distinte quando si indicano capacità di memoria.⁵

La *memoria centrale* contiene dati e istruzioni necessari all'esecuzione dei programmi. La memoria RAM (*Random Access Memory*), utilizzata come memoria di lavoro, consente operazioni di lettura e scrittura ed è normalmente *volatile*: il suo contenuto viene perso quando manca l'alimentazione. L'espressione «accesso casuale» indica che una locazione può essere raggiunta direttamente, senza dover attraversare quelle precedenti, e non che gli indirizzi vengano scelti casualmente. Ciò non implica che tutti gli accessi abbiano necessariamente la stessa latenza.

La memoria ROM (*Read-Only Memory*) è invece non volatile ed è destinata principalmente alla lettura durante il normale funzionamento. Tradizionalmente viene utilizzata per conservare il *firmware*, ossia il software strettamente legato al funzionamento di un dispositivo, comprese le procedure di avvio. Nei sistemi moderni il firmware può risiedere anche in memorie non volatili riprogrammabili, come quelle flash. La possibilità di modificare il contenuto e la capacità di conservarlo senza alimentazione sono quindi proprietà distinte.

La *memoria secondaria*, o *memoria di massa*, permette di conservare dati e programmi anche quando non sono in esecuzione. Rispetto alla memoria centrale offre generalmente capacità maggiori, ma tempi di accesso più elevati. Tra i dispositivi più comuni si distinguono i dischi rigidi e le unità a stato solido.

Un *disco rigido* (HDD, *Hard Disk Drive*) memorizza i dati magneticamente su piatti rotanti; l'accesso richiede il posizionamento delle testine e può comportare attese dovute alla rotazione. Un'unità a stato solido (SSD, *Solid State Drive*) utilizza normalmente memoria flash, priva di parti meccaniche mobili, e offre in genere tempi di accesso inferiori. Entrambi i dispositivi sono non volatili, ma rimangono soggetti a guasti e perdita dei dati.

Durante l'esecuzione, i dati necessari vengono trasferiti dalla memoria secondaria alla memoria centrale. Le modifiche presenti nella memoria di lavoro devono invece essere salvate su un supporto persistente per poter essere conservate dopo lo spegnimento.

⁵NIST, *Prefixed for binary multiples*, prefissi IEC e confronto con i prefissi SI.

Le differenze di velocità, capacità e costo motivano l'organizzazione in una *gerarchia di memoria*, che comprende registri, cache, memoria centrale e memoria secondaria. I livelli più vicini alla CPU offrono generalmente accessi più rapidi, ma capacità inferiori rispetto a quelli destinati all'archiviazione.

La *cache* è una memoria piccola e veloce che conserva copie di dati e istruzioni per ridurre la necessità di accedere a memorie più lente. La sua efficacia dipende dal comportamento dei programmi, che spesso presenta due forme di *località*:

- *località temporale*: un dato utilizzato recentemente ha una maggiore probabilità di essere utilizzato nuovamente;
- *località spaziale*: l'accesso a una locazione è spesso seguito dall'accesso a locazioni vicine.

Queste proprietà permettono alla cache di migliorare le prestazioni quando i dati richiesti sono già disponibili al suo interno. Non costituiscono tuttavia garanzie valide per ogni programma o sequenza di accessi.

I dispositivi di *ingresso e uscita* (I/O, *Input/Output*) consentono lo scambio di dati fra il calcolatore e l'ambiente esterno. Una tastiera acquisisce informazioni, mentre uno schermo presenta i risultati dell'elaborazione. Altri dispositivi, come schede di rete e unità di archiviazione, svolgono entrambe le funzioni. Le periferiche possono comunicare anche con altri sistemi, senza richiedere necessariamente l'interazione di un utente.

Il trasferimento dei dati viene coordinato attraverso interfacce e meccanismi di controllo differenti.

Nell'*I/O programmato* la CPU esegue direttamente le istruzioni necessarie al trasferimento. Può inoltre verificare ripetutamente lo stato della periferica mediante *polling*, occupando tempo di elaborazione durante l'attesa.

Gli *interrupt* permettono invece a un dispositivo di segnalare un evento che richiede attenzione. La CPU può interrompere il flusso corrente, conservare le informazioni necessarie a riprenderlo ed eseguire una procedura di servizio. Terminata la gestione dell'evento, può riprendere il lavoro precedentemente sospeso.

Il *DMA* (*Direct Memory Access*) consente a un controller di trasferire blocchi di dati fra memoria e periferiche senza richiedere che la CPU esegua ogni singola operazione di trasferimento. La CPU configura l'operazione e può ricevere un interrupt quando questa è completata.

Polling, interrupt e DMA svolgono dunque funzioni differenti e possono essere utilizzati in combinazione. In particolare, il DMA riduce il coinvolgimento diretto della CPU nei trasferimenti, ma richiede comunque il coordinamento nell'accesso alla memoria.

1.6 Linguaggi di programmazione e forme di esecuzione

L'organizzazione dell'hardware chiarisce come vengono eseguite le operazioni, ma non come queste siano specificate in un programma. I linguaggi naturali permettono di esprimere concetti il cui significato può dipendere dal contesto e risultare ambiguo. Per descrivere istruzioni che un calcolatore possa interpretare secondo regole precise, si utilizzano invece linguaggi formali.

Un *linguaggio formale* è un insieme di stringhe costruite a partire da un *alfabeto*, cioè un insieme finito di simboli. Una *stringa* è una sequenza finita di simboli; la stringa vuota si indica con ε . Se Σ è un alfabeto, Σ^* rappresenta l'insieme di tutte le stringhe finite costruibili sui suoi simboli, inclusa quella vuota. Un linguaggio formale è pertanto un sottoinsieme $L \subseteq \Sigma^*$.

Una *grammatica formale* descrive come generare le stringhe di un linguaggio mediante simboli terminali, simboli non terminali, un simbolo iniziale e regole di produzione. I terminali costituiscono i simboli delle stringhe prodotte, mentre i non terminali vengono utilizzati durante la derivazione.

Per esempio, considerando i terminali a, b , il simbolo iniziale non terminale S e le produzioni

$$S \rightarrow aSb, \quad S \rightarrow ab,$$

si ottengono le stringhe $ab, aabb, aaabbb, \dots$. La prima regola aggiunge un simbolo a a sinistra e un simbolo b a destra; la seconda conclude la derivazione. Il linguaggio generato contiene quindi stringhe con lo stesso numero positivo di a e b , disposti in quest'ordine.

Noam Chomsky ha classificato le grammatiche in una gerarchia basata sulle restrizioni imposte alle produzioni. In ordine di potere espressivo crescente si distinguono:

- i *linguaggi regolari*, riconosciuti da automi finiti;
- i *linguaggi liberi dal contesto*, riconosciuti da automi a pila;
- i *linguaggi sensibili al contesto*, riconosciuti da macchine di Turing con memoria limitata linearmente rispetto alla dimensione dell'ingresso;
- i *linguaggi di tipo zero*, riconosciuti da macchine di Turing generali.

Ogni classe è contenuta nella successiva. La gerarchia mette in relazione la struttura delle grammatiche con le capacità dei modelli di calcolo. Occorre tuttavia distinguere il *riconoscimento* dalla *decisione*: una macchina di Turing che riconosce un linguaggio di tipo zero può non terminare quando riceve una stringa che non vi appartiene, mentre un procedimento decisore deve terminare per ogni ingresso.

Le grammatiche permettono di descrivere la struttura sintattica dei linguaggi di programmazione, ma non ne determinano da sole il significato e il comportamento.⁶

Un processore esegue direttamente operazioni appartenenti al proprio *insieme di istruzioni* (*instruction set*), che ne definisce le operazioni disponibili e le relative codifiche. Il *linguaggio macchina* esprime i programmi attraverso tali codifiche binarie. Le istruzioni possono specificare, per esempio, trasferimenti di dati, operazioni aritmetiche, confronti e salti. Poiché gli insiemi di istruzioni dipendono dall'architettura, un programma in linguaggio macchina non è necessariamente eseguibile su processori differenti.

Il *linguaggio assembly* rappresenta le istruzioni macchina mediante nomi mnemonici e permette di utilizzare riferimenti simbolici per dati e indirizzi. Un programma detto *assemblatore* traduce il codice assembly in codice macchina. Il linguaggio rimane strettamente legato all'architettura del processore, anche se non ogni riga del sorgente deve necessariamente corrispondere a una singola istruzione macchina: possono essere presenti direttive e altre costruzioni elaborate dall'assemblatore.

I *linguaggi ad alto livello*, come Python e Java, permettono invece di descrivere elaborazioni e strutture di controllo senza specificare direttamente le operazioni sui registri del processore. Una singola istruzione del linguaggio può richiedere numerose istruzioni macchina, rendendo necessaria una forma di traduzione o un ambiente capace di eseguirla.

Il testo di un programma nel linguaggio scelto costituisce il *codice sorgente*. La *sintassi* stabilisce quali costruzioni sono ammesse, mentre la *semantica* ne definisce il significato e il comportamento. Un programma può rispettare tutte le regole sintattiche senza essere corretto rispetto al problema che deve risolvere.

Per trasformare il codice sorgente in operazioni eseguibili si utilizzano differenti tecniche. Un *compilatore* traduce un programma in un'altra rappresentazione, che può essere direttamente codice macchina oppure *codice intermedio* destinato a ulteriori elaborazioni.

Quando la compilazione produce codice macchina destinato all'esecuzione nativa, può essere necessaria una fase di *collegamento* (*linking*) con altri moduli o librerie. Nei sistemi dotati di sistema operativo, l'eseguibile

⁶Per la gerarchia di Chomsky: J. E. Hopcroft, R. Motwani e J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, terza edizione, Pearson.

viene quindi caricato e avviato. Deve essere compatibile sia con l'architettura del processore sia con l'ambiente di esecuzione: la portabilità del codice sorgente non garantisce automaticamente quella del programma compilato.

Un *interprete* è invece un programma che realizza le operazioni previste da un linguaggio, elaborando direttamente il sorgente oppure una sua rappresentazione intermedia. L'interprete viene a sua volta eseguito dal processore e non ne sostituisce il funzionamento.

Compilazione e interpretazione non sono necessariamente alternative: descrivono modalità di elaborazione ed esecuzione che possono essere combinate all'interno dello stesso ambiente.

Per esempio, nell'implementazione di Python denominata *CPython*, il codice sorgente viene tradotto in una rappresentazione intermedia chiamata *bytecode*, successivamente eseguita dall'interprete. Nel percorso usuale di Java, il compilatore produce bytecode destinato alla *Java Virtual Machine* (JVM).

La JVM è un ambiente di esecuzione software, non un processore fisico: può interpretare il bytecode e tradurne alcune parti in codice macchina durante l'esecuzione. In entrambi i casi, fra il sorgente e le istruzioni effettivamente eseguite dalla CPU possono quindi esistere più livelli di rappresentazione e traduzione.⁷

⁷Per la distinzione fra rappresentazioni ed esecuzione: documentazione Python, voce *bytecode*; Oracle, *The Java Virtual Machine Specification*, § 1.2, *The Java Virtual Machine*.

2 Rappresentazione delle informazioni

Come anticipato nella [sezione 1.1](#), un elaboratore conserva e manipola informazioni mediante sequenze di bit. Il significato di una sequenza non è determinato dai soli simboli che la compongono, ma dalla convenzione adottata per interpretarli: gli stessi bit possono descrivere un numero naturale, un intero negativo, un valore in virgola mobile oppure parte di un testo. Una rappresentazione stabilisce la corrispondenza fra tali configurazioni e gli oggetti rappresentati, insieme ai limiti entro i quali le operazioni conservano il significato matematico atteso.¹

2.1 Sistemi di numerazione posizionali

Si adotta la convenzione $\mathbb{N} = \{0, 1, 2, \dots\}$. Il numero naturale e la scrittura utilizzata per esprimerlo sono oggetti distinti: il valore cento rimane lo stesso qualunque sia il sistema di numerazione, mentre cambiano le cifre e le loro posizioni. Nel sistema decimale sono disponibili dieci cifre, da 0 a 9. Il contributo di ciascuna cifra dipende dalla posizione occupata, il cui *peso* è una potenza di dieci.

Definizione 2.1 (Valore posizionale). Siano $b \geq 2$ una base intera, $n \geq 1$ il numero di cifre della parte intera e $m \geq 0$ il numero di cifre della parte frazionaria. Per cifre $a_k \in \{0, 1, \dots, b-1\}$, la scrittura $(a_{n-1} \dots a_0.a_{-1} \dots a_{-m})_b$ rappresenta il valore

$$x = \sum_{k=-m}^{n-1} a_k b^k = \sum_{k=0}^{n-1} a_k b^k + \sum_{j=1}^m a_{-j} b^{-j}.$$

Se $m = 0$, la seconda somma è nulla e si omette il separatore.

La cifra immediatamente a sinistra del separatore ha peso $b^0 = 1$. Spostandosi a sinistra, i pesi vengono moltiplicati per b ; spostandosi a destra, vengono divisi per b . La parte frazionaria ha pertanto pesi $b^{-1}, b^{-2}, \dots$, non potenze positive. Nel capitolo si usa il punto come separatore della parte frazionaria, come avviene comunemente nei linguaggi di programmazione. È una convenzione di scrittura, non una proprietà della notazione posizionale. Il pedice indica la base; un numero privo di pedice in un calcolo è inteso in base dieci.

Esempio 2.1 (Pesi delle cifre decimali).

$$\begin{aligned}(753)_{10} &= 7 \cdot 10^2 + 5 \cdot 10^1 + 3 \cdot 10^0 = 700 + 50 + 3, \\ (2048)_{10} &= 2 \cdot 10^3 + 0 \cdot 10^2 + 4 \cdot 10^1 + 8 \cdot 10^0, \\ (4.34)_{10} &= 4 \cdot 10^0 + 3 \cdot 10^{-1} + 4 \cdot 10^{-2} = 4 + \frac{3}{10} + \frac{4}{100}.\end{aligned}$$

La cifra 3 contribuisce con 300 in $(345)_{10}$, ma con 30 in $(34.5)_{10}$: conta la sua posizione rispetto al separatore, non soltanto il simbolo utilizzato. Uno zero interno, come in 2048, mantiene le posizioni delle altre cifre pur avendo contribuito nullo.

¹Riferimento didattico: S. Canazza, *Rappresentazione delle informazioni*, file 2026_FI_01_canazza_RappresentazioneInformazioni.pdf.

2.2 Notazione binaria e conversioni

In base due sono ammesse soltanto le cifre 0 e 1, dette *bit* (dall'inglese *binary digit*). La scelta è adatta ai circuiti digitali: due intervalli distinguibili di una grandezza fisica, per esempio una tensione, possono essere associati ai due valori logici. Non è necessario che lo stato 0 corrisponda letteralmente a un circuito spento; è necessario poter distinguere in modo affidabile i due stati.

I pesi binari della parte intera sono 1, 2, 4, 8, 16, 32, 64, 128, ...; quelli della parte frazionaria sono $1/2, 1/4, 1/8, \dots$. La conversione in decimale applica direttamente la [definizione 2.1](#).

Esempio 2.2 (Da binario a decimale).

$$\begin{aligned}(1101)_2 &= 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 1 = 13, \\(1.101)_2 &= 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} \\&= 1 + \frac{1}{2} + \frac{1}{8} = \frac{13}{8} = (1.625)_{10}.\end{aligned}$$

Entrambe le uguaglianze sono esatte; la presenza di una parte frazionaria non implica di per sé un'approssimazione.

Per convertire in binario un valore non negativo, si trattano separatamente la parte intera e quella frazionaria, ricongiungendo poi le due scritture con il separatore. Le due parti richiedono procedimenti differenti.

2.2.1 Divisioni successive per la parte intera

Sia $N > 0$ un numero naturale e si ponga $q_0 = N$. La divisione euclidea per due determina, a ogni passo, un quoziente e un resto:

$$q_i = 2q_{i+1} + r_i, \quad r_i \in \{0, 1\}.$$

Si prosegue con il quoziente fino a ottenere $q_t = 0$. Per sostituzione successiva si ricava

$$N = r_0 + 2r_1 + 2^2r_2 + \dots + 2^{t-1}r_{t-1}.$$

Il primo resto è dunque la cifra delle unità, mentre l'ultimo è la cifra più a sinistra. La scrittura binaria si ottiene leggendo i resti nell'ordine inverso a quello di produzione. Per $N = 0$ il risultato è semplicemente la cifra 0.

Esempio 2.3 (Conversione di cento). Le divisioni successive sono

$$\begin{aligned}100 &= 2 \cdot 50 + 0, & 50 &= 2 \cdot 25 + 0, & 25 &= 2 \cdot 12 + 1, \\12 &= 2 \cdot 6 + 0, & 6 &= 2 \cdot 3 + 0, & 3 &= 2 \cdot 1 + 1, \\1 &= 2 \cdot 0 + 1.\end{aligned}$$

I resti prodotti sono 0, 0, 1, 0, 0, 1, 1; letti in senso inverso danno $(100)_{10} = (1100100)_2$. La verifica è $2^6 + 2^5 + 2^2 = 64 + 32 + 4 = 100$.

2.2.2 Moltiplicazioni successive per la parte frazionaria

Sia $f_0 \in [0, 1)$ la parte frazionaria. A ogni passo si moltiplica per due, si registra la parte intera d_j e si conserva come nuovo valore la parte frazionaria f_j :

$$2f_{j-1} = d_j + f_j, \quad d_j \in \{0, 1\}, \quad 0 \leq f_j < 1.$$

Se il prodotto è almeno uno, il bit è 1 e si sottrae uno; altrimenti il bit è 0. In particolare, un prodotto esattamente uguale a uno produce il bit 1 e una nuova frazione nulla. Iterando l'identità,

$$f_0 = \frac{d_1}{2} + \frac{d_2}{2^2} + \dots + \frac{d_t}{2^t} + \frac{f_t}{2^t}.$$

Poiché $0 \leq f_t < 1$, il termine residuo $f_t/2^t$ tende a zero per $t \rightarrow \infty$. I bit si leggono nell'ordine di produzione, diversamente dai resti delle divisioni. Se $f_t = 0$, la conversione termina.

Esempio 2.4 (Frazione binaria finita). Per $f_0 = 0.625$ si ottiene

$$2 \cdot 0.625 = 1 + 0.25,$$

$$2 \cdot 0.25 = 0 + 0.5,$$

$$2 \cdot 0.5 = 1 + 0.$$

Pertanto $(0.625)_{10} = (0.101)_2 = 1/2 + 1/8$.

Esempio 2.5 (Frazione binaria periodica). Per $0.35 = 7/20$ i prodotti, calcolati esattamente, sono

$$2 \cdot 0.35 = 0 + 0.70,$$

$$2 \cdot 0.70 = 1 + 0.40,$$

$$2 \cdot 0.40 = 0 + 0.80,$$

$$2 \cdot 0.80 = 1 + 0.60,$$

$$2 \cdot 0.60 = 1 + 0.20,$$

$$2 \cdot 0.20 = 0 + 0.40.$$

La frazione 0.40 ricompare dopo il sesto passo: i passi dal terzo al sesto si ripetono e producono il periodo 0110. Quindi

$$(0.35)_{10} = (0.01\overline{0110})_2, \quad (100.35)_{10} = (1100100.01\overline{0110})_2.$$

La barra indica le sole cifre ripetute indefinitamente. Una verifica indipendente usa la somma geometrica:

$$(0.01\overline{0110})_2 = \frac{1}{4} + \frac{2^{-4} + 2^{-5}}{1 - 2^{-4}} = \frac{1}{4} + \frac{1}{10} = \frac{7}{20}.$$

Per una frazione $p/q \in [0, 1)$ ridotta ai minimi termini, con $p \in \mathbb{N}$ e $q \in \mathbb{N} \setminus \{0\}$, una rappresentazione finita in base due esiste se e solo se q è una potenza di due, includendo $q = 1$. Infatti, una scrittura con m cifre frazionarie ha valore $K/2^m$ per un intero K ; il denominatore ridotto deve quindi dividere 2^m . Viceversa, se $q = 2^h$, sono sufficienti al più h cifre frazionarie.

Negli altri casi la rappresentazione binaria è infinita e periodica a partire da una certa posizione. Nell'algoritmo le frazioni residue sono multipli di $1/q$ compresi fra zero e uno: le possibilità sono finite. Se non si raggiunge zero, una frazione deve ricomparire e, da quel momento, si ripetono anche i bit successivi. Questo ragionamento presuppone calcoli esatti sui resti; un calcolo già arrotondato non è una verifica affidabile della periodicità.

2.3 Numeri naturali e rappresentazione con modulo e segno

Le conversioni precedenti non impongono un numero prefissato di cifre. Nei formati numerici a lunghezza fissa, invece, il numero di bit disponibili è stabilito in anticipo. Per $n \geq 1$, una sequenza $a_{n-1} \dots a_0$ contiene n bit e ammette 2^n configurazioni. Nell'interpretazione *senza segno*, il suo valore è

$$U = \sum_{k=0}^{n-1} a_k 2^k.$$

Il minimo è zero, ottenuto con tutti i bit nulli; il massimo è $\sum_{k=0}^{n-1} 2^k = 2^n - 1$, ottenuto con tutti i bit uguali a uno. Si rappresentano esattamente gli interi dell'intervallo

$$[0, 2^n - 1] \cap \mathbb{Z},$$

dove $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$ è l'insieme degli interi. Con otto bit, ossia un *byte*, sono disponibili 256 configurazioni, da 0000 0000 a 1111 1111, corrispondenti ai valori da 0 a 255. Gli spazi nelle sequenze servono soltanto a facilitarne la lettura.

Il bit a_{n-1} , a sinistra, è detto *più significativo* (*most significant bit*, MSB); il bit a_0 , a destra, è detto *meno significativo* (*least significant bit*, LSB). Nella rappresentazione senza segno essi hanno rispettivamente peso 2^{n-1} e 1. L'estensione ai numeri negativi richiede una diversa regola di interpretazione delle configurazioni disponibili.

Definizione 2.2 (Modulo e segno). Per $n \geq 2$, nella rappresentazione con *modulo e segno*, il primo bit s indica il segno e i rimanenti $n - 1$ bit esprimono il modulo:

$$a = (-1)^s M, \quad M = \sum_{k=0}^{n-2} a_k 2^k.$$

Il bit $s = 0$ indica il segno positivo, mentre $s = 1$ indica il segno negativo.

Per esempio, su sei bit 001100 rappresenta +12 e 101100 rappresenta -12, poiché in entrambi i casi il modulo è $(01100)_2 = 8 + 4$. Il massimo modulo è $2^{n-1} - 1$, per cui l'intervallo rappresentabile è

$$[-(2^{n-1} - 1), 2^{n-1} - 1] \cap \mathbb{Z}.$$

Esso contiene $2^n - 1$ interi distinti, non 2^{n-1} : fra le 2^n configurazioni, $00 \dots 0$ e $10 \dots 0$ rappresentano entrambe zero, convenzionalmente indicato come +0 e -0. Su otto bit l'intervallo è dunque $[-127, 127] \cap \mathbb{Z}$.

La duplicazione dello zero non è l'unico inconveniente. Per sommare due valori di segno uguale occorre sommare i moduli e mantenere il segno, controllando che il nuovo modulo sia rappresentabile. Per segni opposti occorre invece confrontare i moduli, sottrarre il minore dal maggiore e scegliere il segno del valore con modulo maggiore. Se i moduli sono uguali, il risultato è zero e va scelta una sua rappresentazione. La semplice somma binaria delle sequenze non realizza tutte queste operazioni. Per questo motivo, le comuni architetture rappresentano gli interi con segno in complemento a due. Un campo di segno separato rimane tuttavia utile in altri formati.

2.4 Complemento a due

Il complemento a due assegna un intero distinto a ogni configurazione e permette di usare la stessa addizione binaria per operandi positivi e negativi. La larghezza n fa parte della definizione: senza conoscerla non è possibile interpretare correttamente il segno e il valore.

Definizione 2.3 (Codifica in complemento a due). Fissato $n \geq 2$, sia $a \in [-2^{n-1}, 2^{n-1} - 1] \cap \mathbb{Z}$. La sua codifica è la scrittura senza segno su n bit dell'intero

$$C_n(a) = \begin{cases} a, & a \geq 0, \\ a + 2^n, & a < 0. \end{cases}$$

I non negativi occupano le configurazioni con MSB nullo; i negativi occupano quelle con MSB uguale a uno. Per decodificare una sequenza, si calcola il valore senza segno U e si usa la regola

$$a = \begin{cases} U, & U < 2^{n-1}, \\ U - 2^n, & U \geq 2^{n-1}. \end{cases}$$

Equivalentemente, direttamente dai bit,

$$a = -a_{n-1}2^{n-1} + \sum_{k=0}^{n-2} a_k 2^k. \quad (2.1)$$

Infatti, se il primo bit vale uno, la sottrazione di 2^n trasforma il suo peso $+2^{n-1}$ in -2^{n-1} . I bit rimanenti non rappresentano il modulo del numero negativo.

Il minimo si ottiene con $100 \dots 0$ e vale -2^{n-1} ; il massimo si ottiene con $011 \dots 1$ e vale $2^{n-1} - 1$. L'intervallo è asimmetrico perché zero occupa una delle configurazioni con primo bit nullo. Lo zero ha una sola rappresentazione, $00 \dots 0$, mentre $11 \dots 1$ rappresenta -1 . Con otto bit l'intervallo è $[-128, 127] \cap \mathbb{Z}$; con trentadue bit è $[-2147483648, 2147483647] \cap \mathbb{Z}$, come per il tipo `int` di Java.²

2.4.1 Codifica, inversione e decodifica

Per codificare $-m$, con $1 \leq m \leq 2^{n-1}$, è sufficiente calcolare $2^n - m$ e scriverlo su n bit. Un procedimento equivalente parte dalla scrittura senza segno di m , inverte tutti i bit e aggiunge uno, conservando soltanto n bit. L'inversione sostituisce ogni 0 con 1 e ogni 1 con 0; il valore senza segno diventa $(2^n - 1) - m$. L'aggiunta di uno produce proprio $2^n - m$.

Esempio 2.6 (Un intero negativo su otto bit). Per -13 , il metodo della [definizione 2.3](#) fornisce

$$2^8 - 13 = 256 - 13 = 243 = (1111\ 0011)_2.$$

Il metodo dell'inversione dà lo stesso risultato:

$$0000\ 1101 \longrightarrow 1111\ 0010 \xrightarrow{+1} 1111\ 0011.$$

La verifica con l'equazione (2.1) è

$$-128 + 64 + 32 + 16 + 2 + 1 = -13.$$

Analogamente, -14 è codificato da $1111\ 0010$, il cui valore è $-128 + 64 + 32 + 16 + 2 = -14$.

Per decodificare una sequenza negativa si può sottrarre 2^n dal valore senza segno, oppure invertire i bit e aggiungere uno per ottenere il modulo come numero senza segno. Il caso estremo merita attenzione: $100 \dots 0$ ha modulo 2^{n-1} , che è rappresentabile senza segno ma non come intero positivo in complemento a due su n bit. La sua negazione matematica richiede quindi più bit.

Attenzione 2.1 (Formato e larghezza non sono intercambiabili). La sequenza $1111\ 1111$ vale 255 senza segno, -127 con modulo e segno e -1 in complemento a due su otto bit. La sequenza su sedici bit $0000\ 0000\ 1111\ 1111$ vale invece +255 in complemento a due. Gli zeri iniziali non possono essere omissi se ciò rende ambigua la larghezza del formato.

²Oracle, *The Java Language Specification*, § 4.2.1, *Integral Types and Values*.

Per aumentare la larghezza di un intero in complemento a due senza modificarne il valore, si replica il bit di segno a sinistra (*estensione del segno*). Per esempio, -13 passa da $1111\ 0011$ a $1111\ 1111\ 1111\ 0011$. Per un negativo, aumentando la larghezza da n a $n+h$, il nuovo valore senza segno è $U + (2^{n+h} - 2^n)$; sottraendo 2^{n+h} si ritrova $U - 2^n$. Per un non negativo si aggiungono invece zeri.

2.5 Aritmetica intera e overflow

La somma binaria procede da destra a sinistra. Senza riporto entrante, $0 + 0 = 0$, $0 + 1 = 1$, $1 + 0 = 1$ e $1 + 1 = (10)_2$: nell'ultimo caso si scrive zero e si riporta uno alla colonna successiva. Indicando con c_k il riporto entrante nella posizione k e con r_k il bit del risultato, la regola generale per due operandi con bit a_k, b_k è

$$a_k + b_k + c_k = r_k + 2c_{k+1}, \quad c_0 = 0.$$

La regola comprende anche $1 + 1 + 1 = (11)_2$.

Su n bit il risultato conservato è il resto della somma senza segno divisa per 2^n ; il riporto finale c_n viene scartato. Si parla di calcolo *modulo* 2^n : due interi sono congruenti quando differiscono per un multiplo di 2^n . Poiché $C_n(a)$ è congruente ad a , sommando le codifiche si ottiene una configurazione congruente ad $a + b$. Se $a + b$ appartiene all'intervallo della [definizione 2.3](#), la decodifica dà esattamente la somma matematica; altrimenti si verifica *overflow*, o trabocco.

Esempio 2.7 (Riporto scartato senza overflow). Su otto bit la somma $5 + (-2) = 3$ è

$$\begin{array}{r} 0000\ 0101 \\ + 1111\ 1110 \\ \hline 1\ 0000\ 0011 \end{array}$$

I valori senza segno sommano a $5 + 254 = 259 = 256 + 3$. Scartato il riporto, rimane la codifica di 3, che è anche la somma corretta degli interi con segno. Il riporto finale non prova dunque l'esistenza di overflow con segno.

Proposizione 2.1 (Criterio di overflow nell'addizione). *Nella somma su n bit di due interi in complemento a due si verifica overflow se e solo se gli operandi hanno lo stesso segno e il risultato conservato ha segno opposto. Equivalentemente, l'overflow si verifica se e solo se il riporto entrante nel bit di segno c_{n-1} è diverso dal riporto uscente c_n .*

Dimostrazione. Due operandi di segno opposto hanno una somma compresa fra gli operandi stessi, quindi entro l'intervallo disponibile. Due non negativi possono superare soltanto il massimo: in tal caso la sequenza conservata ha primo bit uno. Due negativi possono scendere soltanto sotto il minimo: in tal caso la sequenza conservata ha primo bit zero. Per verificare il criterio dei riporti basta applicare la regola di somma alla colonna di segno: per segni 00 si ha overflow quando $c_{n-1} = 1, c_n = 0$; per segni 11 quando $c_{n-1} = 0, c_n = 1$. Per segni diversi i due riporti coincidono. $\square$

Esempio 2.8 (Overflow positivo e negativo su quattro bit). L'intervallo disponibile è $[-8, 7] \cap \mathbb{Z}$. Per $6 + 2$,

$$0110 + 0010 = 1000.$$

La somma matematica è 8, non rappresentabile; la sequenza prodotta vale -8 , non $+8$. Non vi è riporto finale, ma vi è riporto entrante nel bit di segno: $c_3 = 1$, $c_4 = 0$.

Per $(-3) + (-7)$, le codifiche sono 1101 e 1001, poiché $16 - 3 = 13$ e $16 - 7 = 9$. Si ottiene

$$1101 + 1001 = 1\ 0110.$$

Scartato il riporto, 0110 vale $+6$, mentre la somma matematica è -10 , inferiore a -8 . Qui $c_3 = 0$, $c_4 = 1$. I risultati conservati sono corretti modulo 16, ma non coincidono con le somme matematiche richieste.

Con otto bit gli analoghi casi di frontiera sono $127 + 1$, che produce 1000 0000, e $-128 + (-1)$, che produce, dopo lo scarto del riporto, 0111 1111. La distinzione è essenziale: nell'addizione senza segno il riporto finale indica il superamento di $2^n - 1$; nell'addizione con segno si usa invece il criterio della [proposizione 2.1](#), non una semplice osservazione delle due cifre più a sinistra del risultato.

Anche la sottrazione si realizza con l'addizione: si invertono i bit del sottraendo, si aggiunge uno e si somma al minuendo, conservando n bit. Per esempio, su otto bit $5 - 7$ dà

$$0000\ 0101 + 1111\ 1001 = 1111\ 1110,$$

che rappresenta -2 . In una sottrazione $a - b$ si verifica overflow quando a e b hanno segni opposti e il risultato ha segno diverso da quello di a . Se $b = -2^{n-1}$, la sua negazione non è rappresentabile nel formato: il procedimento sui bit rimane valido modulo 2^n , ma non va descritto come una negazione intermedia esatta su n bit. Il controllo del risultato resta indispensabile.

2.6 Virgola fissa e virgola mobile

Un formato con un numero finito di bit può distinguere soltanto un numero finito di valori. Non può quindi rappresentare tutti i razionali, e tanto meno tutti i reali, presenti in un intervallo non degenerare. Alcuni valori, tuttavia, sono esattamente rappresentabili, come gli esempi finiti della [sezione 2.2](#): l'approssimazione è una necessità generale, non una proprietà di ogni singolo valore.

Nella *virgola fissa* il separatore ha una posizione prestabilita. Per esempio, usando n bit senza segno di cui m frazionari, con $0 \leq m < n$, si rappresenta $U 2^{-m}$, con $0 \leq U \leq 2^n - 1$. Il passo fra valori consecutivi è sempre 2^{-m} , mentre il massimo è $(2^n - 1)2^{-m}$. Aumentare le cifre frazionarie, a parità di larghezza, migliora la risoluzione ma riduce il massimo. La virgola fissa rimane utile quando è appropriata una scala costante; non è un formato escluso dall'uso nei calcolatori.

La *virgola mobile* separa invece un fattore significativo da un fattore di scala. In base b , un valore non nullo si scrive

$$x = (-1)^s M b^E,$$

dove $s \in \{0, 1\}$ indica il segno, $M > 0$ è il *significando*, spesso chiamato *mantissa*, ed $E \in \mathbb{Z}$ è l'esponente. Per esempio, $(1024.3)_{10} = (1.0243)_{10} \cdot 10^3$. Senza vincoli su M , questa scrittura non è unica: moltiplicare il significando per la base e diminuire l'esponente di uno conserva il valore. La *normalizzazione* impone $1 \leq M < b$, scegliendo un'unica posizione del separatore per ogni valore non nullo.

In base due il significando normalizzato ha dunque la forma $(1.f_1 f_2 \dots)_2$. Per esempio,

$$(10.5)_{10} = (1010.1)_2 = (1.0101)_2 \cdot 2^3.$$

L'esponente controlla l'ordine di grandezza, mentre il numero di cifre del significando controlla la precisione. Un formato concreto limita entrambi: la normalizzazione matematica non garantisce che qualunque valore possa esservi memorizzato.

2.6.1 I formati IEEE 754 binary32 e binary64

Lo standard IEEE 754 specifica formati e regole per l'aritmetica in virgola mobile. Fra i suoi formati binari, *binary32* e *binary64* sono comunemente detti singola e doppia precisione; non sono gli unici formati previsti dallo standard. I loro campi, nell'ordine dal più significativo, sono segno, esponente codificato e frazione del significando.³

La [tabella 2.1](#) distingue i bit memorizzati per la frazione dalla precisione del significando normale, che comprende anche il bit iniziale implicito.

Tabella 2.1. Campi e parametri dei due formati binari IEEE 754.

Parametro	binary32	binary64
Bit complessivi	32	64
Bit di segno	1	1
Bit dell'esponente w	8	11
Bit della frazione t	23	52
Bias B	127	1023
Precisione normale $p = t + 1$	24	53
Esponenti normali E	-126 a 127	-1022 a 1023

Siano e il valore senza segno del campo esponente e F il valore senza segno del campo frazione, con $0 \leq F < 2^t$. La frazione binaria $(0.f_1 \dots f_t)_2$ vale $F/2^t$. Il *bias* è la costante $B = 2^{w-1} - 1$: per i numeri normali l'esponente effettivo è $E = e - B$. L'esponente codificato non è quindi un intero in complemento a due. I valori $e = 0$ ed $e = 2^w - 1$ sono riservati e non si interpretano con questa stessa regola.

Definizione 2.4 (Valori normali e subnormali). Nei formati della [tabella 2.1](#), se $1 \leq e \leq 2^w - 2$, il valore normale è

$$x = (-1)^s \left(1 + \frac{F}{2^t}\right) 2^{e-B}.$$

Il bit iniziale 1 è implicito e non viene memorizzato. Se $e = 0$ e $F \neq 0$, il valore è invece *subnormale*:

$$x = (-1)^s \frac{F}{2^t} 2^{1-B}.$$

In questo caso il bit iniziale implicito è 0 e l'esponente di scala è $1 - B$, non $-B$.

³Per i campi, i valori limite e le regole di arrotondamento: Oracle, *Numerical Computation Guide*, capitolo 2, IEEE Arithmetic. I formati indicati come *single* e *double* corrispondono rispettivamente a binary32 e binary64.

Esempio 2.9 (Codifica di un valore normale). Per $10.5 = (1.0101)_2 \cdot 2^3$ in binary32, il segno è $s = 0$, l'esponente codificato è $e = 3 + 127 = 130 = (10000010)_2$ e il campo frazione è 0101 seguito da diciannove zeri:

$$\underbrace{0}_s \quad \underbrace{10000010}_e \quad \underbrace{010100000000000000000000}_{\text{frazione}}.$$

La frazione vale $2^{-2} + 2^{-4} = 5/16$, quindi la decodifica fornisce $(1 + 5/16)2^3 = (21/16)8 = 10.5$, esattamente.

2.6.2 Valori limite, zeri e configurazioni speciali

Il più piccolo normale positivo si ottiene con $e = 1, F = 0$ e vale 2^{1-B} . Il più piccolo subnormale positivo si ottiene invece con $e = 0, F = 1$ e vale 2^{1-B-t} . Il massimo finito positivo ha $e = 2^w - 2, F = 2^t - 1$, per cui vale $(2 - 2^{-t})2^{2^w-2-B}$. In particolare,

$$\begin{aligned} \text{binary32: } x_{\min, \text{norm}} &= 2^{-126} \approx 1.17549435 \cdot 10^{-38}, \\ x_{\min, \text{sub}} &= 2^{-149} \approx 1.40129846 \cdot 10^{-45}, \\ x_{\max} &= (2 - 2^{-23})2^{127} \approx 3.40282347 \cdot 10^{38}. \end{aligned}$$

Per binary64 il minimo normale positivo vale 2^{-1022} , il minimo subnormale positivo 2^{-1074} e il massimo finito positivo $(2 - 2^{-52})2^{1023}$. Le espressioni in potenze di due sono esatte; i valori decimali preceduti da $\approx$ sono soltanto approssimazioni.

I subnormali sono multipli di 2^{1-B-t} e riempiono con passo costante la regione fra zero e il minimo normale. Il massimo subnormale positivo è $(1 - 2^{-t})2^{1-B}$, distante dal minimo normale proprio un passo. Questa estensione realizza il *gradual underflow*, ossia l'avvicinamento graduale a zero, anziché un salto direttamente dal minimo normale a zero. La precisione relativa si riduce verso zero, poiché diminuisce il numero di cifre significative effettive. Su alcune architetture la gestione dei subnormali può anche avere un costo diverso da quella dei normali; non si tratta di una proprietà matematica del formato.

Quando $e = 0, F = 0$, il bit di segno distingue $+\infty$ e $-\infty$. Quando $e = 2^w - 1, F = 0$, si rappresentano $+\infty$ e $-\infty$. Se invece $e = 2^w - 1, F \neq 0$, si rappresenta un NaN (*Not a Number*). La [tabella 2.2](#) raccoglie questi casi senza applicare indebitamente la formula dei normali.

Tabella 2.2. Interpretazione dei campi riservati in binary32 e binary64.

Esponente codificato	Frazione	Interpretazione
$e = 0$	$F = 0$	Zero con segno
$e = 0$	$F \neq 0$	Subnormale
$1 \leq e \leq 2^w - 2$	Qualunque F	Normale
$e = 2^w - 1$	$F = 0$	Infinito con segno
$e = 2^w - 1$	$F \neq 0$	NaN

Gli zeri con segno risultano uguali nei confronti numerici ordinari, ma il segno può influenzare alcune operazioni: nella gestione IEEE 754 predefinita, $1/(+0)$ produce $+\infty$ e $1/(-0)$ produce $-\infty$, segnalando la divisione per zero. Un infinito può anche risultare da un overflow in virgola mobile, con esito dipendente dalla modalità di arrotondamento. Non è il massimo numero finito del formato.

Un NaN segnala invece un risultato non numerico, come per $0/0, +\infty - (+\infty)$ o la radice quadrata di un numero negativo nell'aritmetica reale. Le operazioni che coinvolgono un NaN normalmente lo propagano; un confronto

di uguaglianza con NaN è falso, anche se i due operandi sono lo stesso NaN. I dettagli di segnalazione e gestione degli errori possono dipendere dall'ambiente di esecuzione. In Python, per esempio, una divisione per zero con l'operatore `/` solleva `ZeroDivisionError` anche per operandi float, anziché restituire automaticamente un infinito o un NaN.

2.7 Precisione e arrotondamento

Per i normali positivi compresi nell'intervallo $[2^E, 2^{E+1})$, con esponente normale E , incrementare F di uno aumenta il valore di

$$\delta_E = 2^{E-t}.$$

Il passo dipende dunque dall'ordine di grandezza: raddoppia passando all'intervallo successivo. Vi sono 2^t valori nell'intervallo $[2^E, 2^{E+1})$, non fra due valori consecutivi. In `binary32`, a partire da $2^{23} = 8388608$, il successivo valore è 8388609; a partire da $2^{27} = 134217728$, è invece 134217744, con passo 16. Negli ultimi intervalli il formato non distingue più tutti gli interi, pur potendo rappresentare valori molto grandi. Nei subnormali il passo è invece costante.

Quando un valore esatto non appartiene al formato, occorre selezionare un valore rappresentabile mediante *arrotondamento*. La modalità predefinita per questi formati è al più vicino, con scelta del valore che ha l'ultima cifra significativa pari nei casi di equidistanza (*round to nearest, ties to even*); in binario ciò significa un ultimo bit conservato uguale a zero. Sono previste anche modalità verso zero, verso $+\infty$ e verso $-\infty$. Arrotondare al più vicino non coincide in generale con troncature le cifre eccedenti.

Per operandi finiti, le operazioni aritmetiche elementari IEEE 754, fra cui somma e prodotto, sono arrotondate come se si calcolasse prima il risultato esatto sugli operandi rappresentati e poi lo si convertisse nel formato di destinazione. Vanno quindi distinti l'errore iniziale di rappresentazione degli operandi e l'eventuale nuovo errore introdotto dall'operazione.

Esempio 2.10 (Il valore decimale 4.35). La frazione $4.35 = 87/20$ non ha una scrittura binaria finita, poiché il denominatore ridotto contiene il fattore 5. In `binary64`, con arrotondamento al più vicino, viene memorizzato il valore esatto

$$\hat{x} = 4.3499999999999996447286321199499070644378662109375.$$

La sua scrittura abbreviata può essere visualizzata come 4.35, ma questo non rende esatto il valore memorizzato. Moltiplicando per il valore esatto 100 e arrotondando nuovamente in `binary64` si ottiene

$$\text{fl}(\hat{x} \cdot 100) = 435 - 2^{-44} \approx 434.99999999999994,$$

dove `fl` indica l'arrotondamento nel formato specificato. Il valore matematico $(87/20) \cdot 100$ è invece esattamente 435. L'esito dipende dal formato: in `binary32` il valore memorizzato di 4.35 è anch'esso leggermente inferiore al valore matematico, ma il prodotto arrotondato con 100 è esattamente 435. Non è quindi corretto attribuire a ogni rappresentazione in virgola mobile lo stesso risultato decimale.

Esempio 2.11 (Allineamento degli esponenti e perdita di cifre). La somma dei valori esatti 10.5 e 0.125 richiede di usare lo stesso esponente:

$$\begin{aligned} 10.5 + 0.125 &= (1.0101)_2 \cdot 2^3 + (1.0)_2 \cdot 2^{-3} \\ &= ((1.0101)_2 + (0.000001)_2) 2^3 \\ &= (1.010101)_2 \cdot 2^3 = 10.625. \end{aligned}$$

Il risultato richiede sette bit significativi complessivi, ossia uno iniziale e sei frazionari. In binary32 e binary64 è rappresentato esattamente, senza errore di somma.

In un formato didattico con soli $p = 5$ bit significativi complessivi, entrambi gli operandi sono rappresentabili, ma nell'intervallo $[8, 16)$ il passo è $2^{3-(5-1)} = 0.5$. Il risultato esatto 10.625 è più vicino a 10.5 che a 11, e viene quindi arrotondato a 10.5. Con $p = 6$ il passo è 0.25: 10.625 è a metà fra 10.5 e 10.75, e la regola di parità sceglie ancora 10.5. Il risultato dipende dunque dal numero di bit e dalla modalità di arrotondamento, non soltanto dal fatto che manchino cifre.

L'addendo più piccolo può quindi non modificare il risultato memorizzato. Poiché gli arrotondamenti avvengono anche durante le operazioni, l'addizione in virgola mobile non è in generale associativa: cambiare l'ordine delle somme può modificare il risultato. Le proprietà dell'aritmetica reale non si trasferiscono dunque automaticamente al calcolo con precisione finita.

2.8 Notazione esadecimale e ottale

Le sequenze binarie dei formati descritti finora possono essere lunghe da leggere e trascrivere. Le basi sedici e otto permettono di abbreviarle conservando una corrispondenza immediata con gruppi di bit. In base sedici le cifre sono 0, 1, ..., 9 e A, B, C, D, E, F, con valori rispettivamente da 0 a 15; in particolare $A = 10$, $B = 11$, fino a $F = 15$. La [definizione 2.1](#) rimane valida: per esempio,

$$(7F)_{16} = 7 \cdot 16 + 15 = 127.$$

Poiché $16 = 2^4$, ogni cifra esadecimale corrisponde esattamente a un gruppo di quattro bit. Per la parte intera si formano i gruppi a partire dal separatore verso sinistra, aggiungendo, se necessario, zeri a sinistra dell'ultimo gruppo. Per la parte frazionaria si procede dal separatore verso destra e si completano i gruppi con zeri a destra. Questo deriva dai pesi: quattro spostamenti binari equivalgono a uno spostamento esadecimale. In senso inverso si sostituisce ogni cifra con la sua scrittura binaria di quattro bit, conservando gli zeri interni.

Esempio 2.12 (Raggruppamento in base sedici).

$$\begin{aligned} (0111\ 1111)_2 &= (7F)_{16}, \\ (10110110)_2 &= (1011\ 0110)_2 = (B6)_{16}, \\ (1101.101)_2 &= (1101.1010)_2 = (D.A)_{16}. \end{aligned}$$

Nell'ultimo caso la verifica è $13 + 10/16 = 13.625 = 13 + 1/2 + 1/8$. La scrittura $0x7F$, comune nei linguaggi di programmazione, è un modo alternativo di indicare il valore esadecimale $(7F)_{16}$.

La base otto utilizza le cifre da 0 a 7. Poiché $8 = 2^3$, le conversioni con la base due si effettuano nello stesso modo, ma con gruppi di tre bit:

$$\begin{aligned}(100010)_2 &= (100\ 010)_2 = (42)_8 = 4 \cdot 8 + 2 = 34, \\(11100010)_2 &= (011\ 100\ 010)_2 = (342)_8 \\&= 3 \cdot 8^2 + 4 \cdot 8 + 2 = 226, \\(31)_8 &= 3 \cdot 8 + 1 = 25.\end{aligned}$$

Per la conversione inversa, $(342)_8$ produce 011 100 010. Se si sta scrivendo un naturale senza larghezza fissata, lo zero iniziale è superfluo; se si descrive un formato con segno, rimane valida l'avvertenza 2.1.

La notazione esadecimale è particolarmente comoda per byte, indirizzi e configurazioni binarie: un byte corrisponde a due cifre esadecimali. La notazione ottale è utile quando la suddivisione naturale è in gruppi di tre bit, come nelle terne di permessi dei sistemi Unix. Nessuna delle due notazioni stabilisce da sola l'interpretazione con segno: $0xFF$ descrive i bit di un byte, che possono valere 255 senza segno oppure -1 in complemento a due.

2.9 Rappresentazione dei caratteri

Per rappresentare un testo, anziché un numero, occorre associare ai caratteri codici numerici e specificare come questi si traducono in unità memorizzabili. Un *repertorio* stabilisce quali caratteri sono previsti; un *insieme di caratteri codificato* assegna loro codici; un *formato di codifica* stabilisce le sequenze di unità, per esempio byte, che li rappresentano. Le convenzioni possono coincidere nei casi più semplici, ma vanno distinte per interpretare correttamente il testo scambiato fra sistemi diversi. Il codice identifica il carattere, non il suo disegno in un particolare font.

2.9.1 ASCII e codifiche a otto bit

ASCII, acronimo di *American Standard Code for Information Interchange*, usa sette bit e definisce 128 codici, da 0 a 127. I codici da 0 a 31 e il codice 127, denominato DEL, sono di controllo. I codici da 32 a 126 sono stampabili, includendo lo spazio al codice 32. Non è quindi corretto includere 127 fra i caratteri stampabili.⁴

Tabella 2.3. Alcuni codici ASCII e intervalli fondamentali, in decimale.

Codice	Carattere o intervallo	Significato
9	HT, \t	Tabulazione orizzontale
10	LF, \n	Avanzamento di riga
13	CR, \r	Ritorno a inizio riga
32	Spazio	Separazione stampabile
48–57	0–9	Cifre decimali
65–90	A–Z	Lettere maiuscole
97–122	a–z	Lettere minuscole
127	DEL	Controllo di cancellazione

⁴V. G. Cerf, RFC 20, *ASCII format for Network Interchange*, 1969, tabella dei codici e descrizione dei caratteri di controllo.

Le notazioni con barra inversa nella [tabella 2.3](#) sono sequenze di escape usate in molti linguaggi per esprimere un singolo carattere nel sorgente. Il ritorno a inizio riga CR non coincide con l'avanzamento di riga LF; i file di testo possono adottare convenzioni di fine riga diverse, fra cui LF e la coppia CR LF.

Il carattere 0 ha codice 48, non il valore numerico zero. Analogamente, il testo 13 è una sequenza di due caratteri con codici 49 e 51, non la codifica intera del numero tredici. La distinzione fra testo e valore numerico spiega perché sia necessaria una conversione quando un programma legge un numero da una stringa. I restanti intervalli stampabili, 33–47, 58–64, 91–96 e 123–126, contengono segni di punteggiatura e altri simboli.

ASCII non contiene le vocali accentate italiane. L'espressione *ASCII esteso* non identifica un unico standard: è una denominazione informale per diverse codifiche a otto bit, come ISO-8859-1 e Windows-1252, che conservano i primi 128 codici e assegnano quelli successivi secondo tabelle differenti. Un byte ammette 256 configurazioni, ma non tutte devono corrispondere a caratteri stampabili o assegnati. Nessuna singola estensione copre necessariamente tutti gli alfabeti occidentali. Per esempio, il byte esadecimale 80 rappresenta il simbolo dell'euro in Windows-1252, ma un codice di controllo in ISO-8859-1: per interpretarlo occorre conoscere la codifica.⁵

2.9.2 Unicode e formati UTF

Unicode definisce un repertorio molto più ampio e assegna ai caratteri valori detti *punti di codice* (*code point*). Si scrivono con il prefisso U+ seguito dal valore esadecimale, come U+0041 per A. Lo spazio dei codici va da U+0000 a U+10FFFF; non tutti i punti sono assegnati a caratteri. I primi 128 hanno le stesse assegnazioni di ASCII. Unicode non si limita quindi a 65536 caratteri e non impone due byte per ciascun carattere.⁶

I punti da U+D800 a U+DFFF sono riservati ai surrogati e non identificano caratteri autonomi. Gli altri punti dello spazio dei codici sono detti *valori scalari Unicode*. I formati UTF traducono questi valori in sequenze di unità di codifica: UTF-8 usa da uno a quattro byte; UTF-16 usa una o due unità di sedici bit; UTF-32 usa un'unità di trentadue bit. Sono formati dello stesso sistema Unicode, non repertori alternativi.

In UTF-8 i codici ASCII sono rappresentati da un unico byte con gli stessi sette bit, preceduti da zero: un testo esclusivamente ASCII ha quindi gli stessi byte anche in UTF-8. Per i codici successivi non vale più la corrispondenza fra un carattere e un byte. In UTF-16 i valori scalari fino a U+FFFF occupano un'unità; quelli superiori richiedono una *coppia di surrogati*. Quando unità di sedici o trentadue bit vengono scritte come byte, va inoltre specificato l'ordine dei byte: per esempio, UTF-16BE e UTF-16LE usano rispettivamente prima il byte più significativo o quello meno significativo.

Esempio 2.13 (Punto di codice e byte memorizzati). La lettera A, punto U+0041, è rappresentata in UTF-8 dal byte 41. La lettera è, punto U+00E8, è rappresentata in UTF-8 dai due byte C3 A8, mentre in ISO-8859-1 usa il singolo byte E8. In UTF-16 la medesima lettera usa l'unità 00E8, serializzata come 00 E8 in UTF-16BE oppure E8 00 in UTF-16LE.

Un carattere percepito può inoltre corrispondere a più punti di codice, come una lettera seguita da un segno combinante. Il numero di byte, di unità di codifica e di caratteri visualizzati non coincide quindi in generale: l'interpretazione dei bit richiede sempre un formato noto.

⁵La [tabella Windows-1252–Unicode](#) nell'archivio Unicode associa 80 a U+20AC (euro).

⁶Unicode Consortium, *The Unicode Standard*, versione 17.0, [capitolo 2](#), in particolare §§ 2.4–2.6 sui punti di codice e sui formati di codifica.

3 Introduzione a Python

Python permette di esprimere in forma eseguibile i procedimenti descritti nella [sezione 1.2](#). Variabili, condizioni, cicli e funzioni consentono di organizzare il calcolo, mentre le operazioni di ingresso e uscita permettono di acquisire i dati e comunicare i risultati. Nel seguito il nome Python indica sempre *Python 3*.¹

3.1 Python e l'esecuzione dei programmi

Python è un linguaggio *ad alto livello e general purpose*: permette di descrivere operazioni senza specificare le istruzioni del processore ed è utilizzabile in ambiti diversi. Nato dal lavoro di Guido van Rossum al CWI alla fine degli anni Ottanta, ebbe la prima versione pubblica nel 1991. Python 3 non è pienamente compatibile con Python 2, che non è più mantenuto: gli esempi destinati alle due versioni non sono quindi sempre intercambiabili.

L'*interprete Python* realizza l'esecuzione del programma. Nell'implementazione CPython, il codice sorgente viene tradotto in *bytecode*, una rappresentazione intermedia eseguita dall'interprete. L'interpretazione non consiste dunque necessariamente nell'eseguire una riga di sorgente alla volta, come chiarito nella [sezione 1.6](#).

Il sorgente può essere inserito in modalità interattiva oppure salvato in un file di testo con estensione `.py`, detto *script*. Avviando l'interprete dal terminale senza indicare uno script, si accede alla modalità interattiva, riconoscibile dal prompt `>>>`. Il ciclo *REPL* (*Read, Eval, Print, Loop*) legge il comando, lo valuta, mostra il risultato di un'espressione quando previsto e attende un nuovo comando. Un assegnamento, per esempio, non produce automaticamente una stampa. Questa modalità è utile per prove brevi; le definizioni rimangono disponibili durante la sessione, ma vengono perse quando questa termina, salvo che siano state salvate in un file.

Uno script conserva invece le istruzioni da eseguire a ogni avvio. Il ciclo di lavoro consiste nel modificare il sorgente, salvarlo, eseguirlo e verificare il risultato. Il flusso di esecuzione può essere modificato da condizioni e cicli, che determinano quali blocchi eseguire o ripetere. A differenza del REPL, una semplice espressione nello script non viene visualizzata automaticamente; per produrre un'uscita si può usare `print()`.

```
1 # Un messaggio sul terminale
2 print("Ciao, mondo!")
```

Una *stringa* è una sequenza di caratteri; negli esempi è delimitata nel sorgente da virgolette singole o doppie dello stesso tipo. Le virgolette non fanno parte del testo stampato. Le parentesi tonde indicano la chiamata della funzione `print`; il testo fra parentesi è il suo *argomento*. `print()` mostra i valori ricevuti, separandoli normalmente con spazi se sono più di uno, e aggiunge un fine riga. Fuori da una stringa, `#` introduce un commento fino alla fine della riga: serve a spiegare il codice e non viene eseguito.

Salvando l'esempio come `saluto.py`, dalla directory che lo contiene si ottiene la seguente esecuzione. Qui `python3` indica il comando che avvia Python 3; il suo nome può dipendere dall'ambiente.

```
1 $ python3 saluto.py
2 Ciao, mondo!
```

Il simbolo `$` è il prompt del terminale e non va digitato. Analogamente, `>>>` identifica il REPL e non fa parte del codice di uno script.

¹Riferimento didattico: S. Canazza, *Python*, file 2026_FI_02_canazza_Python.pdf. Per approfondire: Python Software Foundation, *The Python Tutorial*, in particolare le sezioni dedicate a espressioni, controllo del flusso, moduli e operazioni di ingresso e uscita.

3.2 Variabili, tipi di dato e input/output

Una *variabile* è un nome che fa riferimento a un valore, ossia a un oggetto gestito da Python. L'assegnamento `nome = espressione` valuta prima l'espressione a destra e poi associa il risultato al nome a sinistra. Non esprime un'uguaglianza matematica né una relazione che si aggiorni da sola. Dopo `a = 3` e `b = a`, l'istruzione `a = 4` lascia `b` associato al valore 3. Un nome deve essere definito prima di essere usato, altrimenti si ottiene un errore `NameError`.

L'aggiornamento `punteggio = punteggio + 1` usa il valore corrente di `punteggio`, aggiunge uno e associa al nome il nuovo risultato. Non richiede una dichiarazione separata della variabile. Il tipo è una proprietà dell'oggetto a cui il nome fa riferimento: uno stesso nome può essere riassegnato a oggetti di tipo diverso, ma mantenerne coerente il significato rende il programma più leggibile.

Gli *identificatori* possono usare lettere, cifre e il carattere `_`, ma non iniziare con una cifra né contenere spazi. Non possono coincidere con parole riservate, come `if` e `for`. Python distingue maiuscole e minuscole: `prezzo` e `Prezzo` sono nomi diversi. Si preferiscono nomi descrittivi in *snake_case*, come `numero_studenti`, evitando di riutilizzare nomi di funzioni predefinite, come `input` o `str`.

Fra i tipi fondamentali si distinguono:

- `int`: numeri interi, come 0, 42, -7;
- `float`: numeri in virgola mobile, come 3.14 e -0.5, non necessariamente rappresentazioni esatte dei valori decimali (sezione 2.7);
- `str`: stringhe, come "ciao" e "42";
- `bool`: valori logici `True` e `False`.

Gli interi `int` di Python non hanno una larghezza fissa: la loro precisione cresce secondo necessità, entro i limiti delle risorse disponibili. Non sono quindi soggetti all'overflow su un numero prefissato di bit descritto nella sezione 2.5.

Il separatore decimale è il punto. La funzione `type(valore)` restituisce il tipo: per esempio, `type(42)` è visualizzato come `<class 'int'>`. La stringa "42" è distinta dal numero 42, secondo le convenzioni introdotte nella sezione 2.9.

Gli operatori aritmetici `+`, `-` e `*` esprimono somma, sottrazione e prodotto; `**` esprime la potenza. Per operandi `int` o `float`, `/` produce una divisione con risultato `float`, anche quando il quoziente è intero: `10 / 2` vale 5.0. L'operatore `//` calcola invece il quoziente arrotondato verso il basso, cioè verso $-\infty$; con entrambi gli operandi interi il risultato è `int`, altrimenti è `float`. Non coincide con il troncamento verso zero.

Per interi a e $b \neq 0$, il resto `a % b` soddisfa

$$a = (a // b)b + (a \% b).$$

Il resto ha modulo minore di $|b|$ e, se non è nullo, il segno del divisore. Per esempio, `17 // 5` vale 3 e `17 % 5` vale 2, mentre `-17 // 5` vale -4 e `-17 % 5` vale 3. Il resto è utile, per esempio, per riconoscere gli interi pari mediante `numero % 2 == 0`. Un divisore nullo causa `ZeroDivisionError` per `/`, `//` e `%`.

Le parentesi modificano l'ordine di valutazione. Le potenze precedono prodotti, divisioni e resti, che precedono somme e sottrazioni; a pari livello questi ultimi operatori si raggruppano da sinistra. `**` si raggruppa invece da destra: `2 ** 3 ** 2` vale `2 ** (3 ** 2)`. La potenza precede il segno meno alla sua sinistra: `-3 ** 2` vale -9, mentre `(-3) ** 2` vale 9.

Gli operatori dipendono dal tipo degli operandi: su stringhe, `+` concatena testi e `*` con un intero li ripete. `"3" + "4"` vale quindi "34", non 7; `"ab" * 3` vale "ababab". La concatenazione diretta di una stringa con un numero causa `TypeError`.

Le funzioni `int()`, `float()`, `str()` e `bool()` consentono conversioni esplicite. `int("42")` produce 42; da un valore `float`, `int()` tronca verso zero, per cui `int(-3.9)` vale -3. `float("3.5")` produce 3.5 e `str(42)` produce "42". Un testo non ammissibile causa `ValueError`: `int("3.5")` e `float("ciao")` non sono conversioni valide.

`bool()` determina il valore di verità: zero e la stringa vuota `" "` sono falsi; numeri non nulli e stringhe non vuote sono veri. In particolare, `bool("False")` è `True`: la funzione non interpreta il significato della parola scritta.

`input(prompt)` mostra il messaggio *prompt*, attende una riga e la restituisce come stringa, senza il fine riga finale. Per elaborare numericamente il dato occorre convertirlo:

```
1 primo = int(input("Primo intero: "))
2 secondo = int(input("Secondo intero: "))
3 totale = primo
4 totale = totale + secondo
5 print("Somma:", totale)
6 print("Media:", totale / 2)
```

Con gli ingressi 3 e 4, il programma stampa `Somma: 7` e `Media: 3.5`. L'assegnamento aggiorna `totale` usando il valore precedente; la conversione permette la somma numerica. L'esempio assume che entrambi gli ingressi rappresentino interi validi.

3.3 Logica booleana e istruzioni condizionali

I confronti `==`, `!=`, `<`, `<=`, `>` e `>=` verificano rispettivamente uguaglianza, diversità e relazioni d'ordine, producendo un valore booleano. `=` assegna un valore; `==` confronta due valori. Per esempio, `voto = 18` esegue un assegnamento, mentre `voto == 18` verifica il valore corrente.

Gli operatori `and`, `or` e `not` combinano le condizioni. Per operandi booleani, la congiunzione è vera soltanto quando entrambi sono veri; la disgiunzione è vera quando almeno uno è vero, anche se lo sono entrambi; la negazione inverte il valore di verità. Le *tabelle di verità* elencano tutti i casi possibili:

<i>a</i>	<i>b</i>	<i>a and b</i>	<i>a or b</i>	<i>not a</i>
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

I confronti precedono gli operatori logici, fra i quali l'ordine è `not`, poi `and`, poi `or`. Quindi `True or False and False` vale `True`. Le parentesi rendono esplicito il raggruppamento anche quando non sono strettamente necessarie. Per condizioni booleane *a*, *b*, le *leggi di De Morgan* danno le equivalenze

$$\begin{aligned}\text{not } (a \text{ and } b) &\iff (\text{not } a) \text{ or } (\text{not } b), \\ \text{not } (a \text{ or } b) &\iff (\text{not } a) \text{ and } (\text{not } b).\end{aligned}$$

Negare una congiunzione significa che almeno una condizione non vale; negare una disgiunzione significa che nessuna vale.

Python valuta il secondo operando di `and` e `or` soltanto quando è necessario: è la cosiddetta valutazione *a cortocircuito*. Se gli operandi non sono booleani, questi operatori restituiscono uno degli operandi, non necessariamente `True` o `False`.

L'istruzione `if` esegue un blocco soltanto se la sua condizione è vera. Le clausole `elif` introducono ulteriori alternative; `else`, priva di condizione, copre il caso residuo.

```
1 voto = 27
2 if voto < 18:
3     print("Insufficiente")
4 elif voto < 24:
5     print("Sufficiente")
6 elif voto < 28:
7     print("Buono")
8 else:
9     print("Ottimo")
```

L'esempio assume un voto intero fra 0 e 30. Le condizioni vengono esaminate dall'alto verso il basso: viene eseguito soltanto il blocco della prima condizione vera. Con `voto = 27` l'uscita è `Buono`; i confronti successivi non vengono valutati. Se nessuna condizione è vera, viene eseguito `else`, quando presente. Più `if` indipendenti, invece, controllano separatamente le proprie condizioni e possono eseguire più blocchi: non equivalgono in generale a una catena `if/elif/else`.

I due punti introducono il blocco. L'*indentazione*, cioè il rientro iniziale, fa parte della sintassi e identifica le istruzioni che vi appartengono. Tutte le istruzioni allo stesso livello devono avere lo stesso rientro; il ritorno al livello precedente conclude il blocco. Si usano quattro spazi per livello, senza mescolarli con tabulazioni. Un rientro non valido causa un errore di sintassi; un rientro valido ma posto nel punto sbagliato può invece cambiare il comportamento senza produrre un errore automatico.

3.4 Cicli e iterazioni

Un *ciclo* esegue ripetutamente un blocco di istruzioni. Ogni ripetizione è detta *iterazione*. Il ciclo `while` è regolato da una condizione; `for` percorre invece i valori forniti da un oggetto *iterabile*, come una lista, un `range` o un file.

`while` controlla la condizione *prima* di ogni iterazione. Se è falsa già al primo controllo, il corpo non viene eseguito. Occorre inizializzare i valori usati nella condizione e, quando necessario, aggiornarli nel corpo affinché il ciclo possa terminare. Un uso semplice è la validazione di un ingresso:

```
1 eta = int(input("Eta non negativa: "))
2 while eta < 0:
3     eta = int(input("Valore non valido, riprovare: "))
4 print("Eta accettata:", eta)
```

Il programma accetta il primo intero non negativo. La nuova lettura aggiorna `eta` prima del controllo successivo; senza di essa, un primo valore negativo farebbe ripetere indefinitamente la stessa richiesta. La terminazione dipende dall'inserimento di un dato valido. La verifica riguarda il valore numerico: il programma assume comunque un testo convertibile con `int()`, non gestisce parole o frazioni.

In un ciclo *controllato da un contatore*, l'inizializzazione stabilisce il primo valore, la condizione il limite e l'aggiornamento il progresso. Per contare da uno a cinque si inizializza `n = 1`, si controlla `n <= 5` e, dopo ogni

elaborazione, si esegue $n = n + 1$, abbreviabile come $n += 1$. Un aggiornamento assente o nella direzione sbagliata può lasciare la condizione sempre vera: il ciclo è allora infinito, consuma risorse e impedisce di raggiungere le istruzioni successive. È un errore logico che l'interprete non riconosce in generale come errore di sintassi.

Un ciclo *con sentinella* termina invece quando viene letto un valore convenuto. Per sommare interi fino all'inserimento di zero, si inizializza una somma a zero e si legge il primo numero; finché $\text{numero} \neq 0$, si aggiunge il numero alla somma e si legge il successivo. La sentinella non viene elaborata come dato. Se è il primo ingresso, non viene svolta alcuna iterazione. La sentinella deve essere distinguibile dai dati da elaborare.

`for` associa alla variabile del ciclo, uno alla volta, i valori forniti dall'iterabile ed esegue il corpo per ciascuno. Non richiede un aggiornamento manuale di tale variabile. È adatto sia a un numero prestabilito di ripetizioni sia alla visita degli elementi di una lista.

`range()` descrive una sequenza di interi regolarmente spaziati, non una lista. Con un argomento, `range(stop)` parte da zero con passo uno; con due, `range(start, stop)` parte da `start`; con tre, `range(start, stop, step)` usa il passo indicato. L'estremo iniziale è incluso e quello finale è escluso. `range(5)` dà 0, 1, 2, 3, 4; `range(1, 6)` dà 1, 2, 3, 4, 5; `range(1, 10, 2)` dà 1, 3, 5, 7, 9. Con passo negativo, `range(5, 0, -1)` dà 5, 4, 3, 2, 1.

Il passo deve essere non nullo, altrimenti si ottiene `ValueError`. Con passo positivo vengono visitati valori minori di `stop`; con passo negativo, valori maggiori di `stop`. Se la direzione non permette di partire verso il limite, la sequenza è vuota, come in `range(5, 0)`. Anche `range(0)` è vuoto.

```
1 totale = 0
2 for numero in range(1, 6):
3     totale = totale + numero
4 print("Somma:", totale)
```

L'uscita è `Somma: 15`. La variabile `totale` è un *accumulatore*: conserva il risultato parziale, inizialmente zero per una somma, e viene aggiornata a ogni iterazione. Con una sequenza vuota manterrebbe il valore iniziale. L'uso di 6 come limite escluso permette di includere 5 senza un caso separato.

3.5 Liste

Una *lista* è un contenitore ordinato di elementi, scritto mediante parentesi quadre e valori separati da virgole. `[]` indica una lista vuota. Gli elementi possono avere tipi diversi, ma una lista omogenea rende normalmente più semplice l'elaborazione. «Ordinato» significa che ogni elemento ha una posizione, non che i valori siano già disposti in ordine crescente. Sono ammesse ripetizioni.

La funzione `len(lista)` restituisce il numero di elementi. In una lista di lunghezza n , gli indici non negativi validi sono $0, \dots, n - 1$: `lista[0]` indica il primo elemento. Gli indici negativi contano dalla fine: `lista[-1]` indica l'ultimo e `lista[-2]` il penultimo; l'indice minimo valido è $-n$. Un indice fuori da questi limiti causa `IndexError`. Una lista vuota non ammette accessi a singoli elementi.

Le liste sono *mutabili*: `lista[i] = valore` sostituisce l'elemento in posizione i .

Alcune operazioni si chiamano *metodi* e si scrivono dopo il nome della lista, separati da un punto. Il metodo `lista.append(valore)` aggiunge un elemento in fondo, modificando la lista senza restituire la lista aggiornata. Il metodo `lista.pop()` rimuove e restituisce l'ultimo elemento; se la lista è vuota, causa `IndexError`.

L'espressione `valore in lista` verifica la presenza e produce un booleano. Lo *slicing* `lista[i:j]` restituisce una nuova lista con gli elementi dall'indice i incluso a j escluso. Omettendo i si parte dall'inizio; omettendo

j si arriva alla fine. `sorted(lista)` restituisce una nuova lista ordinata in senso crescente senza modificare l'originale; presuppone elementi confrontabili fra loro.

```
1 voti = [20, 30, 27]
2 voti[0] = 21
3 voti.append(18)
4 ultimo = voti.pop()
5 ordinati = sorted(voti)
6 print("Primi due:", voti[:2])
7 print("Presente 30:", 30 in voti)
8
9 totale = 0
10 conteggio = 0
11 for voto in voti:
12     if voto >= 24:
13         totale = totale + voto
14         conteggio = conteggio + 1
15 if conteggio > 0:
16     print("Media dei voti almeno 24:", totale / conteggio)
```

Dopo la sostituzione, l'aggiunta e la rimozione, `voti` contiene `[21, 30, 27]`; `ultimo` vale 18. L'uscita mostra Primi due: `[21, 30]`, Presente 30: `True` e Media dei voti almeno 24: `28.5`. Il ciclo visita direttamente i valori, senza usare indici, e la condizione seleziona quelli da accumulare. Il contatore permette di calcolare la media dei soli valori selezionati; il controllo finale evita una divisione per zero se nessuno soddisfa la condizione. `ordinati` è una lista distinta con valori `[21, 27, 30]`, mentre l'ordine di `voti` rimane invariato.

3.6 Funzioni

Una *funzione* raccoglie sotto un nome istruzioni riutilizzabili. Consente di evitare ripetizioni di codice e di suddividere un problema in operazioni più semplici. La definizione usa `def`, il nome, le parentesi con gli eventuali *parametri* e i due punti; il corpo è indentato.

Eseguire la definizione rende disponibile la funzione, ma non esegue il suo corpo. La *chiamata*, scritta con il nome seguito dalle parentesi, avvia invece il corpo. I valori passati nella chiamata sono gli *argomenti*; i parametri sono i nomi con cui la funzione li usa. Le parentesi restano necessarie anche per una funzione senza argomenti.

```
1 def area Rettangolo(base, altezza):
2     return base * altezza
3
4
5 risultato = area Rettangolo(8, 5)
6 print("Area:", risultato)
```

`base` e `altezza` sono parametri; 8 e 5 sono gli argomenti della chiamata. `return` termina la chiamata e restituisce il risultato al punto che l'ha invocata, dove può essere assegnato oppure usato in un'altra espressione. L'esempio stampa Area: 40. I parametri e le variabili assegnate nel corpo sono normalmente locali alla chiamata: non diventano automaticamente nomi utilizzabili all'esterno.

`print()` mostra un valore, ma non lo restituisce come risultato del calcolo. Sostituendo il `return` dell'esempio con `print(base * altezza)`, la chiamata stamperebbe 40, ma `risultato` varrebbe `None`. Questo valore indica l'assenza di un risultato specificato: viene restituito quando si raggiunge la fine del corpo senza `return`,

oppure si esegue `return` senza espressione. Anche `print()` restituisce `None`. Una funzione può restituire un booleano, per esempio mediante `return numero % 2 == 0`, o scegliere risultati diversi con istruzioni condizionali.

Le definizioni possono essere raccolte in un file separato, detto *modulo*. Salvando la sola definizione precedente in `geometria.py`, un altro script nella stessa directory può utilizzarla così:

```
1 import geometria
2
3 print("Area:", geometria.area Rettangolo(8, 5))
```

`import geometria` rende disponibile il modulo. La funzione viene richiamata tramite il nome qualificato `geometria.area Rettangolo`, formato dal nome del modulo e da quello della funzione. L'estensione `.py` non si scrive in `import`. Nell'esempio il modulo contiene soltanto la definizione: le eventuali istruzioni poste al suo livello principale verrebbero eseguite anche durante l'importazione.

3.7 Gestione dei file di testo

Un file consente di conservare dati oltre la durata del programma, secondo la distinzione fra memoria di lavoro e archiviazione della [sezione 1.5](#). Un *file di testo* rappresenta caratteri mediante una codifica; un *file binario* contiene byte interpretati secondo il proprio formato, non necessariamente come testo. Anche il testo è memorizzato come byte: la differenza riguarda la loro interpretazione ([sezione 2.9](#)).

Il *percorso* identifica il file. Un percorso relativo, come `"voti.txt"`, viene risolto rispetto alla *directory di lavoro corrente* del processo, normalmente quella dalla quale viene avviato il programma, non necessariamente quella che contiene lo script. Un percorso assoluto non dipende invece dalla directory corrente.

La chiamata `open(percorso, modalita, encoding="utf-8")` apre un file di testo e restituisce un oggetto per leggerlo o scriverlo. Il parametro `encoding` specifica la codifica utilizzata nella scrittura e nell'interpretazione del testo in lettura. Per leggere correttamente un file esistente deve corrispondere alla codifica con cui è stato salvato. UTF-8 permette, fra l'altro, di conservare il testo italiano con i caratteri accentati.

Le modalità essenziali sono:

- `"r"`: lettura di un file esistente;
- `"w"`: scrittura; crea il file se manca e cancella il contenuto precedente già all'apertura;
- `"a"`: aggiunta in fondo; conserva il contenuto esistente e crea il file se manca.

Creare un file presuppone comunque che la directory di destinazione esista e che siano disponibili i permessi necessari.

La forma `with open(...) as file:` associa l'oggetto al nome `file` e ne delimita l'uso a un blocco indentato. All'uscita dal blocco il file viene chiuso automaticamente, anche se si verifica un errore nel corpo. La chiusura libera le risorse e completa le scritture in attesa; non occorre aggiungere una chiamata manuale a `close()`.

```
1 with open("voti.txt", "w", encoding="utf-8") as file:
2     file.write("24\n27\n")
3
4 with open("voti.txt", "a", encoding="utf-8") as file:
5     file.write("30\n")
```

`write()` scrive una stringa e non aggiunge automaticamente un fine riga. La sequenza `\n` indica un carattere di fine riga, secondo le convenzioni della [sezione 2.9](#). Il primo blocco sostituisce il contenuto con due righe, il

secondo ne aggiunge una terza. Il file risultante contiene 24, 27 e 30, un voto per riga; si tratta di testo, non della rappresentazione binaria degli interi. Rieseguendo tutto l'esempio si ottengono ancora soltanto queste tre righe, perché l'apertura iniziale in "w" cancella il contenuto.

Per leggere un file breve per intero si può usare `file.read()` all'interno del blocco `with`: il risultato è una sola stringa. In alternativa, `for riga in file` legge una riga alla volta, senza richiedere di conservare tutto il testo insieme. La lettura per righe è adatta anche a file di grandi dimensioni.

Le righe lette conservano normalmente il fine riga `\n`; l'ultima può esserne priva. Stampare una riga con `print(riga)` può quindi aggiungere un secondo fine riga. `riga.strip()` restituisce una stringa senza spazi bianchi iniziali e finali, incluso il fine riga, ma elimina anche eventuali spazi significativi. Per stampare il testo senza aggiungere un altro fine riga si può usare `print(riga, end="")`.

L'esempio seguente legge il file creato in precedenza, assumendo che ogni riga contenga un intero valido, e ne accumula i valori:

```
1 try:
2     totale = 0
3     with open("voti.txt", "r", encoding="utf-8") as file:
4         for riga in file:
5             totale = totale + int(riga.strip())
6     print("Somma:", totale)
7 except FileNotFoundError:
8     print("File voti.txt non trovato.")
```

Con il file precedente stampa `Somma: 81`; con un file vuoto stampa `Somma: 0`. La conversione `int()` è necessaria perché una riga è una stringa; per dati in virgola mobile si userebbe `float()`. Una riga vuota o non numerica causerebbe `ValueError`: la pulizia con `strip()` non rende valido qualunque contenuto.

Aprire un file inesistente in modalità "r" causa `FileNotFoundError`. Nel frammento, `try` racchiude le operazioni che possono fallire, mentre `except FileNotFoundError` individua il blocco da eseguire per quel particolare errore, senza intercettare gli altri. Se il file manca, il messaggio segnala il problema anziché presentare zero come somma di dati letti.

4 Ricorsione

La ricorsione permette di risolvere un problema mediante soluzioni di problemi della stessa natura, ma più semplici. Una definizione matematica ricorsiva può essere tradotta direttamente in una funzione Python. La concisione del codice non garantisce tuttavia la terminazione né l'efficienza: occorre comprendere come si susseguono le chiamate e quali operazioni rimangono in attesa.¹

4.1 Funzioni ricorsive e terminazione

Definizione 4.1 (Ricorsione). Una funzione è *ricorsiva* quando la sua esecuzione può richiamare la funzione stessa. Nella ricorsione diretta il richiamo compare nel corpo della funzione. Un *caso base* risolve direttamente un caso del problema, senza ulteriori chiamate ricorsive; il *passo ricorsivo* riconduce il problema a uno o più sottoproblemi e ne utilizza i risultati.

La definizione di una funzione è distinta dalle sue invocazioni ([sezione 3.6](#)): una stessa funzione può avere più chiamate non ancora concluse. Richiamarla non modifica la definizione, ma avvia una nuova esecuzione del suo corpo con parametri propri.

Per giustificare la terminazione non basta individuare un caso base: occorre dimostrare che, per ogni ingresso ammissibile, le chiamate ricorsive restano nel dominio previsto e raggiungono un caso base. Un criterio sufficiente consiste nell'associare a ciascun problema una misura intera non negativa che diminuisca strettamente in ogni chiamata ricorsiva. Non esiste una successione infinita strettamente decrescente di interi non negativi. Se i casi terminali sono risolti dai casi base e anche le operazioni non ricorsive terminano, il procedimento termina. Nella ricorsione multipla il progresso va verificato per *ogni* sottoproblema, non soltanto per uno.

Un caso base assente o irraggiungibile può causare una successione illimitata di chiamate nel modello astratto, mentre un programma reale incontra prima i limiti dell'ambiente di esecuzione. Per esempio, sottrarre uno a un numero negativo non permette di raggiungere il caso base $n = 0$. La terminazione va sempre riferita agli ingressi ammissibili, come nella [definizione 1.1](#); la correttezza richiede inoltre casi base corretti e una combinazione dei risultati coerente con la specifica.

4.2 Il fattoriale: iterazione e ricorsione

Per $n \in \mathbb{N}$, il *fattoriale* $n!$ è il prodotto degli interi da uno a n , con $0! = 1$. Il prodotto vuoto vale uno, elemento neutro della moltiplicazione. La definizione ricorsiva è

$$n! = \begin{cases} 1, & n = 0, \\ n(n-1)!, & n > 0. \end{cases} \quad (4.1)$$

Ne risultano $1! = 1$, $2! = 2$, $3! = 6$, $4! = 24$ e $5! = 120$. Il caso base è $n = 0$; il passo ricorsivo richiede $(n-1)!$ e lo moltiplica per n . Per $n > 0$, $n-1$ è ancora naturale ed è strettamente minore di n : dopo n riduzioni si raggiunge zero. La relazione giustifica anche la correttezza: partendo da $0! = 1$, se il risultato per $n-1$ è corretto, moltiplicarlo per n produce $n!$.

Le implementazioni seguenti assumono un argomento di tipo `int`. Per i valori negativi, `raise ValueError("n deve essere non negativo")` segnala un argomento non ammissibile e interrompe la funzione.

¹Riferimento didattico: S. Canazza, *Ricorsione*, file 2026_FI_03_canazza_Ricorsione.pdf.

```
1 def fattoriale_iterativo(n):
2     if n < 0:
3         raise ValueError("n deve essere non negativo")
4     risultato = 1
5     for i in range(1, n + 1):
6         risultato = risultato * i
7     return risultato
8
9
10 def fattoriale(n):
11     if n < 0:
12         raise ValueError("n deve essere non negativo")
13     if n == 0:
14         return 1
15     return n * fattoriale(n - 1)
```

Il ciclo conserva un prodotto parziale: dopo il fattore i , risultato vale $i!$. Con $n = 0$, `range(1, 1)` è vuoto e rimane il valore iniziale uno. La versione ricorsiva traduce invece direttamente l'equazione (4.1); non occorre un `else` dopo il caso base, perché `return` conclude la chiamata. Senza il controllo sui negativi, il ciclo restituirebbe uno, mentre la versione ricorsiva continuerebbe con $-1, -2, \dots$ senza raggiungere il caso base. Il controllo di segno non verifica il tipo: la preconditione richiede comunque un numero intero.

```
1 >>> fattoriale(0), fattoriale(1), fattoriale(4)
2 (1, 1, 24)
3 >>> fattoriale_iterativo(5), fattoriale(5)
4 (120, 120)
```

4.3 Invocazioni sospese e pila delle chiamate

Una chiamata ricorsiva segue le regole di una chiamata ordinaria: l'invocante sospende la propria esecuzione, l'invocata esegue il corpo e, quando termina, restituisce il controllo all'invocante nel punto sospeso. Nel fattoriale il prodotto non è ancora completo quando viene avviata la chiamata interna: prima deve essere disponibile il suo risultato.

Ogni invocazione dispone di un proprio *record di attivazione*, o *frame*, che conserva parametri, variabili locali e informazioni necessarie a riprendere il calcolo. Il parametro n di `fattoriale(4)` non viene sostituito dal parametro n di `fattoriale(3)`: i due nomi appartengono a invocazioni distinte.

Le invocazioni non concluse sono organizzate logicamente in una *pila delle chiamate* (*call stack*): l'ultima avviata è la prima a terminare, secondo l'ordine LIFO (*Last In, First Out*). Ogni nuova invocazione aggiunge un frame alla sommità; alla sua conclusione il frame viene rimosso e riprende l'invocazione precedente. Le invocazioni sospese non eseguono simultaneamente istruzioni: ne viene conservato lo stato.

- Esempio 4.1** (Traccia di fattoriale(4)).
1. Con $n = 4$, il caso base è falso. La chiamata attende il risultato di fattoriale(3) per moltiplicarlo per quattro.
 2. Le chiamate con $n = 3, 2, 1$ si sospendono analogamente, chiedendo rispettivamente il fattoriale di 2, 1, 0.
 3. Con $n = 0$, return 1 conclude il caso base.
 4. Riprende $n = 1$: il risultato ricevuto è uno, quindi viene restituito $1 \cdot 1 = 1$.
 5. Riprende $n = 2$, che restituisce $2 \cdot 1 = 2$; poi $n = 3$ restituisce $3 \cdot 2 = 6$, e infine $n = 4$ restituisce $4 \cdot 6 = 24$ al chiamante esterno.

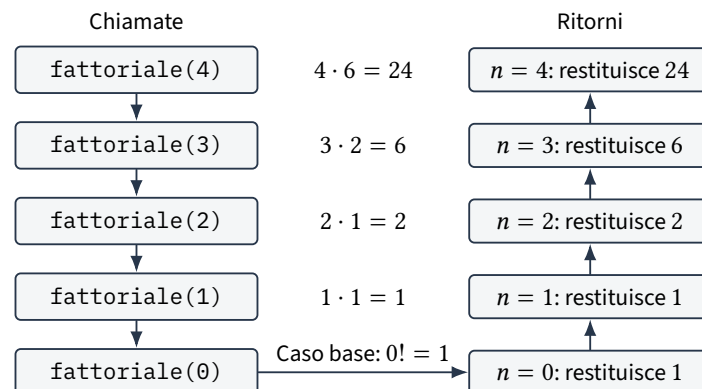


Figura 4.1. Discesa delle chiamate e risalita dei risultati per `fattoriale(4)`. Ogni riga rappresenta un'invocazione distinta; le moltiplicazioni sospese si completano dal basso verso l'alto.

Nella [figura 4.1](#), raggiunto il caso base, coesistono cinque frame del fattoriale: quattro sospesi e uno in esecuzione. Il risultato del caso base raggiunge in successione i chiamanti, ciascuno dei quali completa il proprio prodotto. Per `fattoriale(3)`, quattro invocazioni producono in risalita i valori 1, 1, 2, 6. L'istruzione `return` restituisce un valore senza stamparlo: il REPL lo mostra automaticamente, mentre uno script può visualizzarlo con `print(fattoriale(4))`.

Per un ingresso n , entrambe le versioni eseguono n moltiplicazioni. La versione ricorsiva richiede però $n + 1$ invocazioni e un numero di frame proporzionale a n ; quella iterativa mantiene un solo frame e un numero fisso di variabili. Gestire le chiamate ha un costo aggiuntivo. Questo confronto conta operazioni e variabili, non byte: gli interi che rappresentano i prodotti diventano più grandi e anche il costo delle moltiplicazioni può crescere.

4.4 Ricorsione in coda e limiti di Python

Una chiamata ricorsiva è *in coda* quando il suo risultato viene restituito senza ulteriori operazioni da parte dell'invocante. Non basta che la chiamata compaia nell'ultima riga: in `return n * fattoriale(n - 1)` rimane da eseguire il prodotto. Questa è ricorsione semplice, con una sola chiamata ricorsiva per passo, ma *non* ricorsione in coda.

Per ottenere una versione in coda, si passa il prodotto parziale come parametro. La funzione seguente assume $n \in \mathbb{N}$ e un accumulatore intero `acc`; la chiamata iniziale `fattoriale_coda(n, 1)` calcola $n!$.

```

1 def fattoriale_coda(n, acc):
2     if n < 0:
3         raise ValueError("n deve essere non negativo")
4     if n == 0:
5         return acc
6     return fattoriale_coda(n - 1, acc * n)

```

Il prodotto viene eseguito *prima* della chiamata: non rimane lavoro aritmetico al ritorno. Per esempio, $(4, 1) \rightarrow (3, 4) \rightarrow (2, 12) \rightarrow (1, 24) \rightarrow (0, 24)$. Per ogni valore intero iniziale dell'accumulatore, il risultato è $\text{acc} \cdot n!$. La trasformazione in un ciclo consiste nel sostituire il richiamo con l'aggiornamento dei parametri e nel ripetere il controllo del caso base:

```

1 def fattoriale_ciclo(n, acc):
2     if n < 0:
3         raise ValueError("n deve essere non negativo")
4     while n > 0:
5         acc = acc * n
6         n = n - 1
7     return acc

```

Il ciclo rende esplicita la successione degli stati senza accumulare chiamate. Anche la ricorsione non in coda può essere simulata senza richiami ricorsivi, conservando esplicitamente il lavoro sospeso quando necessario; la trasformazione non è sempre altrettanto immediata. L'esecuzione su un processore non impone al compilatore o all'interprete di sostituire ogni funzione ricorsiva con un ciclo: chiamate, ritorni e stato in memoria sono sufficienti a realizzarla.

CPython non elimina le chiamate ricorsive in coda delle funzioni Python: anche `fattoriale_coda` accumula frame. In ambienti che eliminano tali chiamate, invece, questo costo può essere evitato; la minore efficienza non è una proprietà universale della ricorsione in coda.²

La profondità delle chiamate ricorsive in CPython è limitata: superando il limite, si verifica `RecursionError`. Il limite dipende dall'ambiente e dalle chiamate già attive, quindi non stabilisce un valore massimo universale per n . Aumentarlo non corregge una ricorsione priva di progresso e può compromettere la stabilità del processo.³ Un algoritmo matematicamente terminante può dunque non completarsi nell'ambiente concreto per insufficienza di risorse.

4.5 Ricorsione multipla e numeri di Fibonacci

Nella *ricorsione multipla*, una stessa invocazione può effettuare più chiamate ricorsive. Un esempio è la successione di Fibonacci, qui definita per gli indici $n \geq 1$:

$$F_1 = F_2 = 1, \quad F_n = F_{n-1} + F_{n-2} \quad (n > 2). \quad (4.2)$$

²Documentazione Python, *What's New in Python 3.14, A new type of interpreter*: l'ottimizzazione delle chiamate in coda fra funzioni Python non è implementata in CPython.

³Documentazione Python, modulo `sys`, *Recursion limit*.

I primi termini sono 1, 1, 2, 3, 5, 8, 13, ... La funzione ricorsiva traduce la definizione in due casi base e due chiamate per ogni indice maggiore di due:

```

1 def fib(n):
2     if n < 1:
3         raise ValueError("n deve essere almeno 1")
4     if n == 1 or n == 2:
5         return 1
6     return fib(n - 1) + fib(n - 2)

```

L'argomento è assunto di tipo `int`. Per $n > 2$, entrambi gli indici ricorsivi sono positivi e minori di n , quindi ogni ramo raggiunge uno dei casi base. Le due chiamate non avvengono in parallelo: Python valuta prima `fib(n - 1)`, poi `fib(n - 2)` e infine somma i risultati. Anche questa funzione non è in coda.

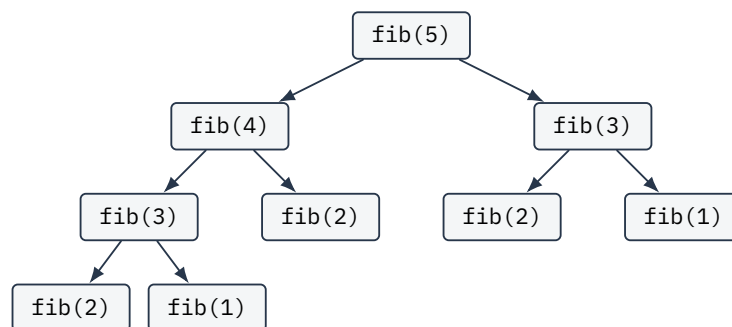


Figura 4.2. Albero completo delle invocazioni di `fib(5)`. I nodi sono chiamate, gli archi collegano invocante e invocata; nodi con lo stesso argomento sono esecuzioni distinte.

La [figura 4.2](#) contiene nove invocazioni: `fib(3)` viene calcolata due volte, `fib(2)` tre e `fib(1)` due. Le invocazioni con lo stesso argomento sono distinte, pur producendo il medesimo risultato. L'albero descrive l'intera esecuzione; la pila conserva invece soltanto le invocazioni non concluse lungo il percorso corrente, non tutti i nodi dell'albero contemporaneamente.

Indicando con C_n il numero totale di invocazioni per `fib(n)`, inclusa la prima, vale

$$C_1 = C_2 = 1, \quad C_n = 1 + C_{n-1} + C_{n-2} \quad (n > 2).$$

Per induzione si ottiene $C_n = 2F_n - 1$: la formula vale per $n = 1, 2$ e il passo segue dall'identità $1 + (2F_{n-1} - 1) + (2F_{n-2} - 1) = 2F_n - 1$. Per $n = 5$ si hanno nove chiamate; per $n = 10$, 109. Il numero di chiamate cresce esponenzialmente: per $n \geq 2$ valgono $F_{n+2} \geq 2F_n$ e $F_{n+1} \leq 2F_n$. L'aumento è dovuto al calcolo ripetuto degli stessi sottoproblemi. La profondità massima della pila cresce invece linearmente con l'indice.

Un ciclo calcola ciascun nuovo termine una sola volta e conserva soltanto i due precedenti:

```

1 def fib_iterativo(n):
2     if n < 1:
3         raise ValueError("n deve essere almeno 1")
4     precedente = 1
5     corrente = 1
6     for i in range(3, n + 1):
7         nuovo = precedente + corrente
8         precedente = corrente
9         corrente = nuovo
10    return corrente

```

Prima dell'iterazione di indice i , precedente vale F_{i-2} e corrente vale F_{i-1} . Il valore temporaneo nuovo permette di aggiornare entrambi senza perdere un operando. Per $n = 1, 2$ il ciclo è vuoto e restituisce uno; per $n \geq 3$ esegue $n - 2$ somme. Il numero di operazioni è quindi lineare e il numero di variabili è fisso, pur crescendo la dimensione degli interi. La versione ricorsiva segue direttamente l'equazione (4.2), ma quella iterativa evita di ripetere i calcoli già svolti.

```
1 >>> fib(1), fib(2), fib(5), fib(10)
2 (1, 1, 5, 55)
3 >>> fib_iterativo(5), fib_iterativo(10)
4 (5, 55)
```

4.6 Esercizi di riepilogo

Esercizio 4.1 (Somma dei primi naturali). Scrivere una funzione ricorsiva che, per un intero $n \geq 0$, restituisca $1 + 2 + \dots + n$, con somma nulla per $n = 0$. Identificare caso base, passo ricorsivo e misura che garantisce la terminazione.

Soluzione 4.1. Il caso base è la somma vuota; il passo ricorsivo aggiunge n alla somma dei naturali fino a $n - 1$. La misura n diminuisce di un'unità a ogni chiamata, fino a raggiungere zero.

```
1 def somma_naturali(n):
2     if n < 0:
3         raise ValueError("n deve essere non negativo")
4     if n == 0:
5         return 0
6     return n + somma_naturali(n - 1)
```

Per $n = 4$ il risultato è 10; per $n = 0$ è zero. L'addizione rimane sospesa durante la chiamata ricorsiva: la ricorsione non è quindi in coda.

Esercizio 4.2 (Conto alla rovescia). Scrivere una funzione ricorsiva che, per un intero $n \geq 0$, stampi gli interi da n a zero in ordine decrescente, uno per riga. Spiegare perché il problema diventa più semplice e distinguere i valori stampati dal valore restituito.

Soluzione 4.2. Ogni invocazione stampa il proprio argomento e, se questo è positivo, richiama la funzione con $n - 1$. La misura n diminuisce di un'unità a ogni chiamata, fino a raggiungere zero e la chiamata ricorsiva è in coda.

```
1 def conto_rovescia(n):
2     if n < 0:
3         raise ValueError("n deve essere non negativo")
4     print(n)
5     if n == 0:
6         return
7     return conto_rovescia(n - 1)
```

Con `conto_rovescia(3)` vengono stampati 3, 2, 1, 0, uno per riga e in quest'ordine; con zero viene stampato soltanto zero. Il valore restituito è `None`, non la sequenza stampata. Anche questa ricorsione in coda rimane soggetta ai limiti di profondità di CPython.