# 1. Introduzione

La parola *algoritmo*, un tempo utilizzata quasi esclusivamente da matematici e informatici, è oggi sempre più diffusa anche nel linguaggio comune. Una semplice ricerca sul sito `repubblica.it` restituisce oltre 12 000 articoli contenenti questo termine (dato aggiornato a luglio 2025). La Figura 1.1 mostra la frequenza della parola *algoritmo* nei libri in lingua italiana indicizzati da Google Books, evidenziando un costante aumento a partire dagli anni '60, seguito da una vera e propria esplosione negli ultimi anni.

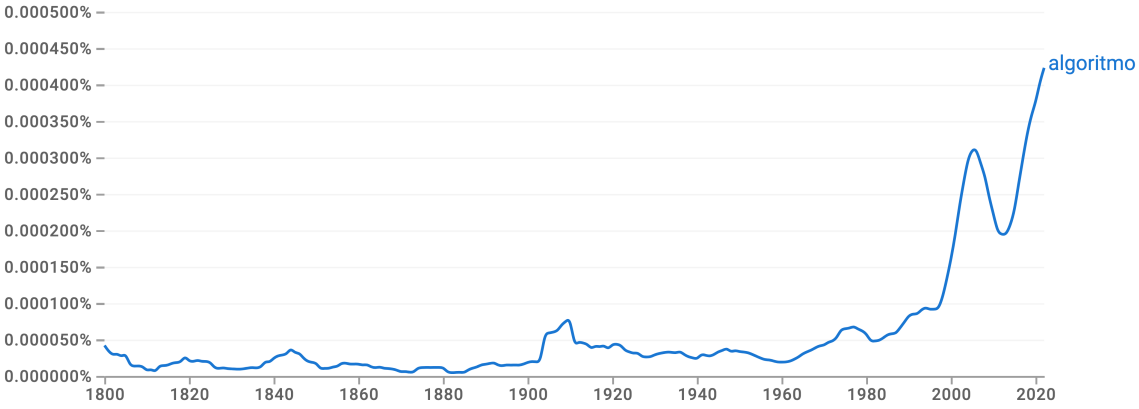



Figura 1.1: La popolarità della parola "algoritmo" nei libri in italiani indicizzati da Google Books

L'uso sempre più diffuso del termine ha portato anche a un certo grado di ambiguità e confusione nel linguaggio comune. Spesso, nei media e nel dibattito pubblico, la parola viene impiegata in modo vago o impreciso per riferirsi a qualunque forma di automazione, decisione informatica o comportamento opaco dei sistemi digitali. Si sente dire, per esempio, che "un algoritmo ha deciso chi doveva ricevere il bonus", o che "l'algoritmo di TikTok premia certi contenuti". In realtà, queste affermazioni non si riferiscono a singoli algoritmi ben definiti, ma a interi sistemi software complessi, che includono molte componenti–dati, euristiche, modelli predittivi, vincoli normativi –spesso non trasparenti nemmeno per chi li sviluppa.

**Un po' di storia** Il termine *algoritmo*, che significa "procedimento di calcolo", è una forma moderna del latino medievale *algorismus*, derivato dal nome del matematico persiano Muḥammad ibn Mūsā *al-Khwārizmī*, vissuto attorno al IX secolo. Le sue opere più celebri sono note in Europa con i titoli *Algoritmi de numero Indorum* (una traduzione del 1126 di un testo arabo oggi perduto) e *Liber algebrae et almucabala* (traduzione del 1145 dell'opera *Al-kitāb al-mukhtasar fī hisāb al-ğabr wa'l-muqābala*, redatta attorno all'820).

Il primo testo introdusse nel mondo arabo, e successivamente in quello occidentale, la numerazione indiana — da noi comunemente chiamata "numerazione araba". Il secondo segna l'inizio dello sviluppo dell'algebra, il cui stesso nome deriva dalla parola araba *al-ğabr*, che significa "completamento" o "unione".

L'accezione moderna della parola *algoritmo*, intesa come procedimento matematico ben definito, si è affermata nel XIX secolo, periodo in cui il termine inizia a comparire nella letteratura scientifica. Nella prima metà del Novecento, in seguito alla formulazione dell'*Entscheidungsproblem* da parte di Hilbert nel 1928, si sono moltiplicati gli sforzi per caratterizzare formalmente i sistemi di calcolo — e con essi, la nozione stessa di algoritmo. Tra i protagonisti di questo fondamentale passaggio storico si ricordano, fra gli altri, Alan Turing, Alonzo Church e Emil Post.

Figura 1.2: Un'opera di al-Khwārizmī

Oltre a definire genericamente sistemi informatici, capita spesso che il termine algoritmo venga usato per qualunque descrizione di un procedimento che risolva un problema, anche la più strampalata e imprecisa. Si attribuisce per esempio valore algoritmico a processi vaghi, solo perché seguono una qualche sequenza di passi. Per esempio, si paragonano spesso gli algoritmi alle ricette di cucina–anche l'autore ha usato questa metafora in un testo precedente.

**Esempio (Ricetta dei canederli)** Taglia 250g di pane raffermo a cubetti e bagnalo con 150ml di latte. Lascialo riposare per mezz'ora finché si ammorbidisce. Intanto fai soffriggere una cipolla tritata con un po' di burro, poi aggiungi 100g di speck a dadini e fai rosolare. Unisci tutto al pane insieme a due uova, un po' di prezzemolo tritato, sale, pepe e un cucchiaio o due di farina, quel tanto che basta a ottenere un impasto morbido ma lavorabile. Forma con le mani bagnate delle palline grandi come un mandarino. Falle cuocere per 15 minuti in acqua bollente salata (o in brodo) a fuoco basso: devono sobbollire e non sfaldarsi.

## 1.1 Algoritmi e problemi

Una definizione così imprecisa di algoritmo può essere adatta per il grande pubblico, ma lascia tuttavia molto a desiderare per chi studia informatica. Nulla viene detto sulle caratteristiche del problema da risolvere, e ancora meno in che modo l'algoritmo debba essere espresso. Il dibattito sulla caratterizzazione precisa di cosa sia un algoritmo è ancora aperto; diamo qui una definizione che collega i concetti di algoritmo e di problema computazionale, proposta da Donald Knuth.

**Definizione (Problema computazionale)** Dato un dominio di input e un dominio di output, un *problema computazionale* è rappresentato dalla relazione matematica che associa ad ogni elemento del dominio di input un elemento del dominio di output.

**Definizione (Algoritmo)** Dato un problema computazionale, un *algoritmo* è un procedimento effettivo che risolve tale problema in tempo finito, espresso tramite un numero definito di passi elementari ben specificati in un sistema formale di calcolo.

Nel parlare di *procedimento effettivo* si intende che ogni passo dell'algoritmo deve essere eseguibile in modo meccanico, senza ambiguità o intuizione. L'espressione *in tempo finito* implica che l'algoritmo deve terminare dopo un numero finito di passi, mentre *passi elementari ben specificati* si riferisce a operazioni descritte con precisione, realizzabili da un esecutore automatico.

Per continuare l'esempio delle ricette, l'input è dato dagli ingredienti, mentre l'output è dato dal piatto cucinato. Il sistema formale di calcolo corrisponde al cuoco. Qui la metafora si ferma, perchè è difficile definire la relazione matematica che intercorre fra input e output, e spesso i passi elementari non sono ben specificati: "quel tanto che basta".

Per meglio comprendere la differenza fra problema e algoritmo, si considerino i problemi segunti:

**Problema (MINIMO)** Il minimo di un insieme $A$ è l'elemento di $A$ che è minore o uguale ad ogni elemento di $A$: $\min(A) = a \Leftrightarrow \forall b \in A : a \leq b$

**Problema (RICERCA)** Sia $A = a_0, \ldots, a_{n-1}$ una sequenza di dati ordinati e distinti, $a_0 < a_1 < \cdots < a_{n-1}$. Eseguire una ricerca della posizione di un dato $v$ nella sequenza $A$ consiste nel restituire l'indice corrispondente, se $v$ è presente, oppure $-1$, se $v$ non è presente.

$$lookup(A, v) = \begin{cases} i & \exists i \in \{0, \ldots, n-1\} : a_i = v \\ -1 & \text{altrimenti} \end{cases}$$

Non abbiamo specificato che tipo di valori siano inclusi nell'insieme o nella sequenza; potrebbero essere interi o reali, per esempio. Per i nostri scopi, è sufficiente sapere che esiste una relazione di ordine totale "$<$" su questi valori.

Così come sono, queste definizioni matematiche non rappresentano certo un insieme di istruzioni che possano essere eseguite (da un calcolatore o da un essere umano) per risolvere il problema. Facciamo quindi un primo tentativo, cercando di risolvere i due problemi proposti. Con un po' di fantasia possiamo trarre ispirazione dalle definizioni stesse, inventando i seguenti "algoritmi":

- per trovare il minimo di un insieme, confronta ogni elemento con tutti gli altri; l'elemento che è minore di tutti è il minimo;
- per trovare un valore $v$ nella sequenza $A$, confronta $v$ con tutti gli elementi di $A$, in ordine, e restituisci la posizione corrispondente; restituisci $-1$ se nessuno degli elementi corrisponde.

A prima vista queste soluzioni potrebbero sembrare soddisfacenti; in realtà vi sono diversi problemi.

**Descrizione di algoritmi.** Innanzitutto, le soluzioni proposte sono descritte in linguaggio naturale e fanno uso di una terminologia che andrebbe definita più precisamente (cosa significano esattamente "con tutti gli altri" e "corrispondente"?). Risolvere un problema richiede di saper descrivere il procedimento risolutivo nel modo più formale possibile.

**Valutazione di algoritmi.** Una volta descritto un algoritmo in qualche linguaggio formale, possiamo ritenerci soddisfatti? Oppure esistono degli algoritmi "migliori" del nostro? Ovviamente dovremmo definire esattamente il significato di "migliore"; per il momento, è possibile limitarsi alla seguente osservazione: questi non sembrano algoritmi particolarmente "furbi".

Per esempio, la nostra proposta per la ricerca del minimo richiede che ogni valore sia confrontato con tutti gli altri, per un totale di $n(n-1)$ confronti, dove $n$ è la dimensione dell'insieme. Tuttavia, molti di questi confronti sono inutili: se $a < b$ e $a < c$, è inutile confrontare $b$ e $c$; nessuno dei due potrà essere il minimo. Allo stesso modo, nel problema RICERCA non abbiamo sfruttato l'ipotesi

che i numeri siano ordinati; tutti i confronti effettuati dopo aver incontrato il primo elemento maggiore di $v$ sono inutili, poiché tutti i valori successivi sono sicuramente maggiori di $v$.

**Progettazione di algoritmi**. I due problemi appena proposti sono estremamente semplici; non c'è bisogno di un esperto programmatore per suggerire delle soluzioni (tanto più se sono così inefficienti!). Ma non sempre è così; molti problemi non hanno soluzioni "ispirate dalla definizione", e richiedono un notevole sforzo di progettazione. Fortunatamente, esistono tecniche di progettazione generali che possono venire in aiuto. È fondamentale padroneggiare queste tecniche per essere in grado di risolvere al meglio il maggior numero di problemi.

## 1.2 Descrizione di algoritmi

Nel descrivere un algoritmo, si utilizzano *azioni elementari* che debbono dare in tempo limitato un risultato certo, unico e ripetibile. In altri termini, l'effetto di un'azione elementare su un dato è determinato univocamente dall'azione e dal dato, cioè non dipende da variabili casuali. Se viene ripetuta la stessa azione sullo stesso dato si ottiene sempre lo stesso risultato. Il ritardo per ottenere il risultato è finito e noto a priori. Inoltre, le azioni debbono essere comprensibili ed eseguibili senza possibilità di ambiguità alcuna da parte dell'esecutore.

Il *livello di dettaglio* con cui si specificano le azioni elementari è un aspetto cruciale nella descrizione di un algoritmo. Tale livello dipende dalle capacità dell'esecutore. Nel caso dei calcolatori, è necessario che ogni azione sia descritta in modo estremamente preciso. In effetti, un calcolatore esegue algoritmi espressi in un linguaggio di basso livello, detto *linguaggio macchina*, costituito da istruzioni elementari che operano su registri e celle di memoria. Fortunatamente, per progettare algoritmi non è necessario scrivere direttamente in linguaggio macchina: è possibile utilizzare costrutti di *linguaggi di programmazione ad alto livello*, più vicini al linguaggio umano, che favoriscono chiarezza, concisione e portabilità. Questi costrutti possono essere tradotti automaticamente in linguaggio macchina tramite appositi programmi detti *compilatori* o *interpreti*.

In questo testo, descriveremo gli algoritmi in *pseudocodice*, un linguaggio fittizio non associato a compilatori reali, il cui scopo è rappresentare l'essenza degli algoritmi evitando i dettagli specifici di un particolare linguaggio di programmazione, che possono distrarre. I linguaggi vanno e vengono: in passato si è fatto ampio uso di Pascal, Java e C; oggi è molto diffuso Python, apprezzato per la sua sintassi semplice e leggibile. Tuttavia, i concetti fondamentali alla base degli algoritmi rimangono invariati nel tempo: lo pseudocodice ci permette di esprimerli in modo chiaro e conciso, indipendentemente dalla moda del momento.

Come scritto nella pagina Wikipedia sullo pseudocodice, non esiste una definizione standard e ogni libro definisce il proprio; noi non saremo da meno. Il linguaggio proposto in questo testo è stato definito in base alle seguenti linee guida:

- *Comprensibilità*: utilizzeremo una sintassi meno rigorosa, adottando parole chiavi esplicite ed evitando strutture sintattiche astruse;
- *Leggibilità*: sfrutteremo, per quanto possibile, gli strumenti forniti dal software di composizione di questo libro (Latex) per rendere la struttura del codice la più chiara ed evidente possibile;
- *Universalità*: tutte le volte che è possibile, utilizzeremo le notazioni in comune al maggior numero di linguaggi, evitando quelle troppo specifiche; per esempio, scriveremo $a+=1$ per denotare l'incremento della variabile $a$, ma non $a++$ che non è presente in Python.

Utilizziamo come esempio due algoritmi, scritti in pseudocodice, che risolvono i due problemi computazionali descritti in precedenza, così da introdurre i principali costrutti che impiegheremo. Ulteriori esempi verrano proposti per illustrare costrutti più avanzati.

**Minimo**. Supponiamo che un insieme di $n$ valori interi siano rappresentati da un vettore $A[0 \ldots n-1]$, di tipo **int**[]. Un possibile algoritmo per identificare il minimo è il seguente: si sceglie il primo

elemento di $A$ come minimo parziale, copiandolo in una variabile *minSoFar*. Si confronta *minSoFar* con gli elementi compresi fra le posizioni 1 e $n-1$; ogni volta che si incontra un elemento minore, si aggiorna il minimo parziale. Al termine, *minSoFar* contiene il minimo dell'insieme, che viene restituito al chiamante.

```
int min(int[] A, int n)
    int minSoFar = A[0]                          % Minimo parziale
    for i = 1 to n - 1 do
        if A[i] < minSoFar then
            minSoFar = A[i]                      % Nuovo minimo
    return minSoFar
```

**Ricerca della posizione**. Come sopra, supponiamo che la sequenza sia memorizzata in un vettore. L'algoritmo proposto per risolvere questo problema prende il nome di *ricerca binaria* (o *ricerca dicotomica*) e parte dalla seguente osservazione: se si confronta $v$ con un qualsiasi elemento $A[m]$ del vettore, possono darsi tre casi:

- $A[m] = v$: $v$ è l'$m$-esimo elemento della sequenza;
- $A[m] < v$: gli elementi che precedono $A[m]$ nel vettore non possono contenere $v$, in quanto minori di $A[m]$; la ricerca deve continuare nei soli elementi che seguono $A[m]$;
- $A[m] > v$: simmetrico al caso precedente, la ricerca deve continuare nei soli elementi che precedono $A[m]$.

Con questa idea in mente, come scegliere l'indice $m$? In assenza di ulteriori informazioni sulla distribuzione dei numeri nella sequenza, una scelta conveniente potrebbe essere quella di prendere il mediano fra l'indice iniziale e l'indice finale del sottovettore preso in considerazione. In questo modo, siamo sicuri di scartare ad ogni passo metà degli elementi rimasti, arrivando in pochi passi alla soluzione.

L'algoritmo binarySearch() implementa la ricerca dicotomica invocando una procedura ricorsiva ausiliaria, bsRec(), che realizza la logica dell'algoritmo vero e proprio. binarySearch() richiama bsRec() con gli indici di inizio e fine dell'intervallo da esaminare. L'algoritmo termina quando l'elemento cercato viene trovato, nel qual caso restituisce il relativo indice, oppure quando l'intervallo diventa vuoto (cioè quando $start > end$), nel qual caso restituisce $-1$ per indicare che il valore non è presente nel vettore.

```
int binarySearch(int[] A, int n, int v, )
    return bsRec(A, v, 0, n - 1)
```

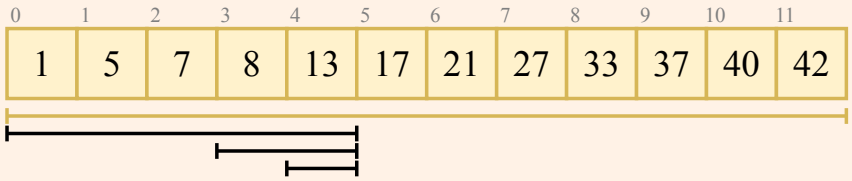
```
int bsRec(int[] A, int v, int start, int end)
    if start > end then
        return -1                                % Vettore vuoto
    else
        int m = ⌊(start + end)/2⌋
        if A[m] == v then
            return m                             % Trovato
        else if A[m] < v then
            return bsRec(A, v, m + 1, end)       % Sottovettore di destra
        else
            return bsRec(A, v, start, m - 1)     % Sottovettore di sinistra
```

**Reality check** L'algoritmo di ricerca binaria è implementato nella funzione `bsearch` della libreria standard del C, nel metodo `Arrays.binarySearch` della libreria Java, e nel modulo `bisect` di Python, spesso usato per mantenere ordinate liste in modo efficiente.

Sebbene il principio della ricerca binaria sia semplice, l'implementazione richiede attenzione. Un classico esempio riguarda il calcolo dell'indice mediano: la formula $m = \lfloor (start + end)/2 \rfloor$ può causare *overflow* se *start* e *end* sono interi molto grandi. Per evitarlo, si preferisce la forma equivalente $m = start + \lfloor (end - start)/2 \rfloor$, più sicura dal punto di vista aritmetico. Questo errore, inizialmente trascurato, è stato segnalato da Josh Bloch nella versione 1.2 di `Arrays.binarySearch` (1998) ed è stato corretto solo nella versione 1.4.2 (2003). Si è ripresentato anche in sistemi reali come motori di ricerca e software embedded. La lezione è evidente: anche gli algoritmi più classici vanno implementati con cura, tenendo conto delle concrete limitazioni del modello computazionale.

**Un po' di storia** L'idea di dimezzare progressivamente un intervallo per localizzare una soluzione risale almeno al 1817, quando Bernard Bolzano formalizzò il *metodo della bisezione* per trovare le radici di una funzione continua. Il primo riferimento esplicito alla *ricerca binaria* in ambito informatico si trova nel 1946, in una nota di John Mauchly — uno dei progettisti dell'ENIAC — che descrive un metodo per trovare rapidamente un numero in una tabella ordinata. Una versione pienamente corretta dell'algoritmo, in grado di evitare problemi di Overlflow, appare solo nel 1960 in un lavoro di Derrick Lehmer.

**Esempio (Ricerca binaria)** Cercando $v = 13$ nel vettore $A$ seguente, si ottengono 4 chiamate:




| Chiamata | $i$ | $j$ | $m$ | $A[m]$ |
|---|---|---|---|---|
| 1 | 0 | 11 | 5 | 17 |
| 2 | 0 | 4 | 2 | 7 |
| 3 | 3 | 4 | 3 | 8 |
| 4 | 4 | 4 | 4 | 13 |

### 1.2.1 Sintassi di base

Lo pseudocodice prende ispirazione dai principali linguaggi di programmazione, come C, Java, Python; quest'ultimo in particolare è fonte di ispirazione.

Utilizzeremo i più comuni costrutti condizionali e iterativi, le cui parole chiave saranno rappresentate in grassetto:

- il costrutto condizionale "**if** *condizione* **then** *istruzione*" comanda l'esecuzione dell'istruzione purché la condizione sia vera; la sua forma più generale "**if** *condizione* **then** *istruzione1* **else** *istruzione2*" comprende anche un'istruzione alternativa, da eseguire qualora la condizione sia falsa;
- il costrutto iterativo illimitato "**while** *condizione* **do** *istruzione*" permette di ripetere iterativamente l'istruzione; la ripetizione prosegue se la condizione è verificata e si arresta non appena la condizione è falsa;

- i costrutti iterativi limitati:
  – "**for** *indice* = *estremoInferiore* **to** *estremoSuperiore* **do** *istruzione*"
  – "**for** *indice* = *estremoSuperiore* **downto** *estremoInferiore* **do** *istruzione*"
  che sono un modo compatto di scrivere:

*indice* = *estremoInferiore*
**while** *indice* ≤ *estremoSuperiore* **do**
  *istruzione*
  *indice* = *indice* + 1

*indice* = *estremoSuperiore*
**while** *indice* ≥ *estremoInferiore* **do**
  *istruzione*
  *indice* = *indice* − 1

La funzione iif$(condizione, v_1, v_2)$ sarà utilizzata per scrivere codice più compatto; restituisce $v_1$ se la condizione è vera, altrimenti restituisce $v_2$. Corrisponde all'operatore ternario di C/C++/Java `condizione ? v1 : v2`.

Utilizzeremo anche il costrutto iterativo limitato "**foreach** *elemento* ∈ *insieme* **do** *istruzione*", che itera l'istruzione assegnando alla variabile *elemento* tutti i valori contenuti in *insieme*, senza un particolare ordine. In un linguaggio reale tale costrutto può essere tradotto in vari modi, a seconda di come l'insieme viene rappresentato; se l'insieme è memorizzato in un vettore, viene sostituito da un ciclo **for**; se l'insieme è una delle `Collections` di Java, l'insieme può essere scandito con un iteratore (si veda il Capitolo 3.4) o con il costrutto "for-each" introdotto in Java 1.5. Il costrutto **foreach** permette di scrivere algoritmi più compatti e più semplici da comprendere.

Un *blocco* è una sequenza di istruzioni che condividono la stessa indentazione (il rientro a destra delle righe), da considerarsi come una sola istruzione composita. Le linee verticali evidenziano in modo grafico dove inizia e finisce un blocco; non sono presenti marcatori espliciti come **begin/end** del Pascal o le parentesi graffe di C/C++/Java.

Infine, i commenti vengono indicati con il simbolo "%".

### 1.2.2 Tipi di dato primitivi

Ogni linguaggio di programmazione fornisce un certo numero di tipi di dato *primitivi*. Nel nostro pseudocodice, utilizzeremo interi, reali e booleani, identificati dalle parole chiave **int**, **real** e **boolean**. Le variabili saranno dichiarate al primo utilizzo, antecedendo il tipo al nome, come nel seguente esempio: **int** *minSoFar* = $A[0]$.

Le variabili utilizzate come contatori nei cicli **for** saranno di tipo intero, e per semplicità non verranno dichiarate; possiamo assumere che il tipo venga inferito dal contesto e dai valori di inizializzazione, come avviene per esempio in linguaggi quali il Go. Analogo discorso vale per il costrutto **foreach**, dove il tipo viene rilevato a partire dall'insieme a cui si applica il costrutto stesso. Se non diversamente specificato, le variabili dichiarate all'interno di una procedura hanno visibilità locale alla procedura stessa.

Utilizzeremo la notazione matematica standard per scrivere espressioni su questi tipi di dato; oltre ai classici $+$, $-$, $\cdot$, $/$, **and**, **or**, **not**, anche le operazioni log per il logaritmo, mod per il modulo, $\lfloor x \rfloor$ e $\lceil x \rceil$ per la parte intera inferiore e superiore, e così via.

Gli operatori booleani **and** e **or** sono corto-circuitati: ovvero, nell'espressione $x$ **and** $y$, viene prima valutato $x$; se questo è falso, $y$ non viene valutato e l'espressione è falsa. Nell'espressione $x$ **or** $y$, viene prima valutato $x$; se questo è vero, $y$ non viene valutato e l'espressione è vera.

Nel resto del libro utilizzeremo il tipo di dato astratto **item** per rappresentare un tipo generico, del quale non è necessario conoscere né la struttura interna né l'insieme completo delle operazioni disponibili. Assumeremo tuttavia che sia definita su di esso una relazione d'ordine totale, indicata con il simbolo "$<$".

### 1.2.3 Tipi di dato composti

Nello scrivere una procedura in un linguaggio di programmazione può presentarsi la necessità di voler definire ed utilizzare, oltre ai tipi di dato primitivi, nuovi tipi di dato, che siano in qualche modo più vicini alla natura del problema che si vuole risolvere. Ogni linguaggio mette a disposizione costrutti per comporre tipi di dato esistenti in tipi di dato più complessi.

I *vettori* o *array* permettono di riunire sotto un unico nome un numero prefissato di elementi dello stesso tipo e di individuare i singoli elementi tramite *indici*. Un *record* (Pascal), *struttura* (C), *classe* (Java) è un agglomerato di elementi di tipo diverso. Ogni elemento (*campo*) è individuato con un nome di riferimento, utilizzato per accedere all'elemento in operazioni di lettura o scrittura. Per esempio, il tipo record RECTANGLE contenente due campi è dichiarato nel modo seguente:

| RECTANGLE |
|---|
| **int** *lunghezza* |
| **int** *altezza* |

I vettori possono essere unidimensionali o multidimensionali (*matrici*). Il tipo di un vettore unidimensionale contenente elementi di tipo **item** è denotato con **item**[], mentre il tipo di un vettore multidimensionale (o matrice) viene dichiarato con **item**[]...[], con tante coppie di parentesi quante sono le dimensioni.

È possibile quindi dichiarare variabili di tipo vettore e record in maniera molto semplice:

$$\begin{aligned} \textbf{item}[]\ A &= \textbf{new item}[0\ldots n-1] = \{0\} \\ \textbf{item}[][]\ B &= \textbf{new item}[0\ldots n-1][0\ldots m-1] = \{-1\} \\ \text{RECTANGLE}\ r &= \textbf{new RECTANGLE} \end{aligned}$$

Queste tre righe dichiarano una variabile $A$ di tipo **item**[] inizializzata come vettore di $n$ elementi tutti uguali a zero; una variabile $B$ di tipo **item**[][] come matrice di $n \times m$ elementi tutti uguale a -1; e una variabile $r$ di tipo RECTANGLE. La parte relativa all'inizializzazione è opzionale.

Assumeremo che tutte le variabili di tipo composto siano riferimenti a strutture di dati, ovvero entità che permettono un accesso indiretto agli oggetti memorizzati. Questo modello si avvicina a quello adottato da linguaggi come Java e Python, dove le variabili non contengono direttamente l'oggetto, ma un collegamento (non manipolabile direttamente) alla sua posizione in memoria. In questo libro, useremo i termini riferimento e oggetto in senso astratto, senza entrare nei dettagli implementativi legati a puntatori espliciti come in C o C++. Assumeremo che la memoria venga allocata con il comando **new**, e liberata con **delete**. Un *riferimento "vuoto"* viene indicato da **nil**.

I campi di una variabile $r$ di tipo RECTANGLE vengono acceduti tramite la notazione a punto: *r.lunghezza*. Per specificare un elemento di un vettore o di una matrice, scriveremo il nome del vettore seguito dagli indici dell'elemento fra parentesi quadre: per esempio, $A[i]$ e $B[i][j]$. Sebbene alcuni linguaggi (ad es., Java) permettano di ottenere la lunghezza di un vettore tramite costrutti del tipo `A.length`, espliciteremo sempre questa lunghezza fra i parametri di ingresso delle procedure.

Un'altra utile struttura composta è la *coppia* (o *tupla*, nel caso generale con più di due elementi), che consente di raggruppare più valori, anche di tipo diverso, in un'unica entità logica. Per esempio, la coppia $\langle x, y \rangle$ rappresenta un oggetto composto da due elementi, che possono essere assegnati direttamente a due variabili distinte. La notazione $\langle \ldots \rangle$ verrà utilizzata nel seguito per costruire e restituire tuple da una procedura, in modo simile a quanto avviene in Python. Ad esempio, una procedura che restituisce sia il massimo valore contenuto in un vettore sia il suo indice potrà restituire la tupla $\langle max, pos \rangle$.

Per semplificare molti dei nostri algoritmi, utilizziamo il costrutto $a, b = b, a$ (mutuato da Python) per indicare che il contenuto della variabile $a$ è scambiato con il contenuto della variabile

$b$; corrisponde alla sequenza di istruzioni

$$temp = a; \quad a = b; \quad b = temp$$

dove *temp* è una variabile temporanea opportunamente dichiarata.

Infine, concludiamo con alcune osservazioni:

- Nella descrizione in linguaggio naturale degli algoritmi proposti, faremo riferimento a porzioni di vettori utilizzando la notazione $A[i \ldots j]$, ovvero l'insieme di elementi compresi fra l'indice $i$ e l'indice $j$, estremi inclusi.
- Così come è definito, un record non è altro che un contenitore; se viene associato ad un insieme di procedure che fungono da operatori su di esso, diventa un *tipo di dato astratto*; ulteriori dettagli verranno forniti nel Capitolo 3.
- Come forse il lettore attento ha già notato, per facilitare la lettura utilizziamo fonti tipografiche diverse per i diversi elementi sintattici del nostro pseudocodice. Funzioni e procedure saranno indicate dal font helvetica; le variabili dal font *times corsivo*; i tipi di dato dal MAIUSCOLETTO; le parole chiave del linguaggio in **times bold**.

### 1.2.4 Procedure e funzioni

Un algoritmo descritto per mezzo di costrutti tipici di un linguaggio di programmazione è comunemente detto *procedura* o *funzione*.

Per esempio, la funzione min(**int**[] $A$, **int** $n$) indica un algoritmo che può essere utilizzato con due dati di ingresso arbitrari (i *parametri formali*), chiamati all'interno della procedura col nome $A$ (di tipo **int**[]) e $n$ (di tipo **int**). La funzione può essere richiamata con *parametri attuali* del medesimo tipo:

$$\begin{array}{c} \mathsf{min}(V, 1024) \\ \mathsf{min}(E, h) \end{array}$$

Il testo della procedura che descrive un algoritmo è strutturato come una sequenza di azioni (o passi) elementari tra loro distinti ed ha lunghezza finita e costante per tutti i possibili dati di ingresso. D'altro canto, l'algoritmo è generale ed applicabile ad una infinità di dati d'ingresso, di grandezza arbitraria, ed il tempo necessario per eseguire la procedura può variare a seconda della grandezza dei dati di ingresso.

Specificare una procedura permette di definire un algoritmo una volta per tutte, di poterlo individuare con un nome e di poterlo utilizzare più volte, con valori distinti e arbitrari, purché consistenti, dei dati d'ingresso. Ciò ha un duplice vantaggio:

- si evita di riscrivere più volte un algoritmo per effettuare la stessa computazione con valori diversi dei dati;
- è possibile sviluppare algoritmi complessi gradualmente, scrivendo dapprima procedure che, utilizzando costrutti elementari del linguaggio, risolvono sottoproblemi più semplici e successivamente procedure più complesse che utilizzano le procedure scritte precedentemente come se fossero nuove istruzioni elementari.

In questo modo, una procedura $A$ per la risoluzione di un certo problema $P$ può richiamare al suo interno una procedura $B$ che risolve un sottoproblema $Q$, che a sua volta può richiamare $C$ per risolvere $R$, e così via. Al momento della chiamata, l'esecuzione di $A$ viene temporaneamente sospesa per permettere l'esecuzione di $B$, sui dati fornitigli da $A$. L'esecuzione di $A$ viene poi ripresa, al termine dell'esecuzione di $B$, a partire dall'istruzione seguente alla chiamata.

I parametri attuali vengono passati per *valore*: la procedura chiamata riceve una copia dei parametri attuali, ed eventuali modifiche a tali parametri non vengono viste dalla procedura chiamante. Nell'esempio precedente, il valore della variabile $h$ viene copiato nel parametro formale $n$, ed eventuali modifiche ad $n$ nel corpo di min() non modificano il valore di $h$.

Quando viene passato un oggetto di tipo composto, viene passata una copia del riferimento all'oggetto; questo significa che la procedura chiamata potrà modificare gli elementi presenti nell'oggetto, ma non scambiarlo con un altro (per esempio sostituendolo con un vettore più grande).

Quando una procedura restituisce un valore al chiamante è detta *funzione*. Nella dichiarazione, il nome della funzione è preceduto dal tipo di dato che viene restituito; nel codice, la restituzione avviene tramite l'istruzione **return**. Non usiamo parole chiave per identificare una procedura o una funzione, preferendo evidenziare graficamente il nome fra due linee orizzontali.

Infine, un'osservazione sui dati di input alle procedure. In alcuni casi, può essere necessario specificare esplicitamente quali dati siano considerati accettabili e quali no. La condizione di accettabilità è detta *precondizione* ed è introdotta all'inizio delle procedure con la parola chiave **precondition**. Una realizzazione realistica dei nostri algoritmi dovrebbe verificare la precondizione, se presente; nel caso non sia verificata dovrebbe restituire un codice di errore (o un'eccezione, in Java).

### 1.2.5 Ricorsione

Il meccanismo di chiamate tra procedure può essere anche ricorsivo, nel senso che una procedura può richiamare se stessa al suo interno! Ne abbiamo visto un esempio con la procedura binarySearch(). L'apparente circolarità (per eseguire binarySearch() si richiama ancora binarySearch()) è spezzata da una condizione di chiusura, che interrompe l'esecuzione della procedura quando i dati raggiungono opportuni valori (nell'esempio, quando $i > j$). Altrimenti, la procedura richiama se stessa, sospendendo temporaneamente la vecchia computazione e iniziando una nuova computazione su dati che hanno sì lo stesso nome (nell'esempio: $i$ e $j$), ma valori diversi.

La procedura binarySearch() è un esempio di *ricorsione di coda* (*tail recursion*), ovvero la chiamata ricorsiva è l'ultima istruzione eseguita. In questo caso, è semplice scrivere binarySearch(), una versione iterativa della stessa funzione, in quanto non occorre mantenere in memoria i dati della vecchia computazione quando ne viene invocata una nuova. Le versioni iterative sono di solito più efficienti, sebbene non abbiano la medesima chiarezza espositiva delle versioni ricorsive.

```
int binarySearch(int[] A, int start, int end)
int m = ⌊(start + end)/2⌋                          % Calcolo dell'indice centrale
while start < end and A[m] ≠ v do
    if A[m] < v then
        start = m + 1                               % Parte destra
    else
        end = m − 1                                 % Parte sinistra
    m = ⌊(start + end)/2⌋
if start > end then
    return −1                                       % Valore non trovato
else
    return m                                        % Valore trovato in posizione m
```

### 1.2.6 Funzioni predefinite

Per semplificare il codice, assumeremo l'esistenza di alcune funzioni predefinite che eseguono operazioni di base su vettori. Inoltre, daremo per scontata la possibilità di riutilizzare qualunque algoritmo presentato nel libro come sottoprocedura. Nella tabella seguente, usiamo **item** per rappresentare un tipo generico su cui sia definita una relazione d'ordine totale (come **int**, **real** o tipi più complessi confrontabili).

| Firma della funzione | Descrizione |
|---|---|
| **item** min(**item**[] *A*, **int** *n*)<br>**item** min(**item**[] *A*, **int** *start*, **int** *end*) | Restituisce il valore minimo all'interno del vettore *A*. Nel primo caso cerca in $A[0 \dots n-1]$, nel secondo in $A[start \dots end]$. |
| **item** max(**item**[] *A*, **int** *n*)<br>**item** max(**item**[] *A*, **int** *start*, **int** *end*) | Come sopra, ma restituisce il valore massimo. |
| **int** argmin(**item**[] *A*, **int** *n*)<br>**int** argmin(**item**[] *A*, **int** *start*, **int** *end*) | Restituisce l'indice dell'elemento minimo. Nel primo caso cerca in $A[0 \dots n-1]$, nel secondo in $A[start \dots end]$. |
| **int** argmax(**item**[] *A*, **int** *n*)<br>**int** argmax(**item**[] *A*, **int** *start*, **int** *end*) | Come sopra, ma restituisce l'indice dell'elemento massimo. |
| **item** sum(**item**[] *A*, **int** *n*)<br>**item** sum(**item**[] *A*, **int** *start*, **int** *end*) | Restituisce la somma degli elementi contenuti in *A*. Nel primo caso somma $A[0 \dots n-1]$, nel secondo $A[start \dots end]$. |

## 1.3 Valutazione di algoritmi

Una volta stabilito un linguaggio per comunicare gli algoritmi, dobbiamo dotarci di strumenti per valutarli. Due sono i principi cardine che utilizzeremo.

**Correttezza**. L'algoritmo che abbiamo proposto è corretto? Risolve correttamente un particolare problema computazionale?

Provare la correttezza di un algoritmo richiede una dimostrazione matematica. Si considerino, per esempio, gli algoritmi min() e binarySearch(). Nel caso di min(), una proprietà matematica che può essere dimostrata è la seguente: all'inizio di ogni iterazione del ciclo **for**, la variabile *minSoFar* contiene il minimo parziale degli elementi $A[0 \dots i-1]$. Un'affermazione di questo tipo prende il nome di *invariante di ciclo* o, più semplicemente, *invariante*. Con il termine "all'inizio" intendiamo nel momento in cui la variabile *i* assume il valore che avrà durante l'iterazione, prima di eseguire il codice dell'istruzione associata al ciclo **for**.

Dobbiamo dimostrare che l'invariante è vera in due casi specifici:

(1) **passo base**: l'affermazione è vera all'inizio della prima iterazione del ciclo;
(2) **passo induttivo**: se è vera all'inizio di un'iterazione, deve essere vera al termine dell'iterazione stessa, e quindi all'inizio dell'iterazione successiva.

Nel caso di *minSoFar*, è facile dimostrare il passo base, in quanto all'inizio del primo ciclo $i = 1$ e $A[0 \dots 0]$ contiene il solo primo elemento $A[0]$, che è stato precedentemente copiato in *minSoFar*. Per quanto riguarda il passo induttivo, se *minSoFar* contiene il minimo di $A[0 \dots i-1]$ all'inizio di un'iterazione, possono darsi due casi: se $A[i]$ è minore di *minSoFar*, esso è il minimo di $A[0 \dots i]$, e viene correttamente copiato in *minSoFar*; altrimenti *minSoFar* rimane invariato. In entrambi i casi, il passo induttivo è dimostrato.

Dimostrare un'invariante di questo tipo ha senso se fornisce informazioni sulla correttezza dell'algoritmo in generale; infatti, se si considera l'invariante al termine del ciclo, quando $i = n$, vediamo che *minSoFar* contiene il minimo di $A[0 \dots n-1]$, e quindi può essere correttamente restituita come valore finale. In questo caso, e in quelli che seguono, assumiamo tacitamente che la variabile *i* "esista" al termine del ciclo e contenga il valore successivo all'estremo superiore, che ha causato la terminazione del ciclo (come nei linguaggi derivati dal C).

Nel caso di binarySearch(), non utilizziamo un'invariante, ma adottiamo comunque un approccio induttivo per dimostrare la correttezza dell'algoritmo, qualunque sia la dimensione $n = end - start + 1$ della porzione di vettore in cui cercare. Se $n = 0$, il vettore è vuoto e $start = end + 1$; la condizione del primo **if** è vera, quindi binarySearch() restituisce correttamente il valore $-1$, ad indicare che $v$ non è presente nella porzione in cui cercare. Questo dimostra il caso base. Supponendo che binarySearch() termini correttamente per tutte le dimensioni fino ad $n-1$ (ipotesi induttiva), vogliamo dimostrare che termina correttamente anche per $n$. La dimostrazione segue banalmente dai tre casi dell'algoritmo: se si trova il valore nell'indice mediano, l'algoritmo termina restituendo questo indice; altrimenti il valore (se presente) si trova sicuramente prima o dopo l'indice mediano, a causa dell'ordinamento. Il vettore in cui cercare avrà quindi dimensione massima $\lfloor n/2 \rfloor$, e la chiamata ricorsiva verrà eseguita correttamente per l'ipotesi induttiva.

Entrambe le dimostrazioni sono basate sull'induzione matematica. Si è detto che gli informatici sono matematici che sanno solo produrre dimostrazioni basate sull'induzione; questo non è totalmente vero (anche le dimostrazioni per assurdo e le riduzioni hanno il loro ruolo, come vedremo) ma è certamente vero che molti degli algoritmi che proporremo qui sono o ricorsivi o incrementali, e si prestano bene a dimostrazioni di questo tipo.

Nel seguito, dimostreremo la correttezza dei nostri algoritmi ogni volta che non sia immediatamente chiara dalla definizione dell'algoritmo stesso.

**Efficienza**. Gli algoritmi che abbiamo proposto sono "veloci" o "efficienti"? Utilizzano al meglio le risorse del calcolatore? Abbiamo già discusso alcune semplici considerazioni sull'efficienza introducendo i nostri problemi computazionali; cerchiamo ora di formalizzare meglio le nostre idee, ma la domanda è molto complessa, ed una risposta approfondita è rimandata al Capitolo 2.

Per quanto riguarda MINIMO, abbiamo descritto informalmente un algoritmo in cui ogni valore veniva confrontato con tutti gli altri, richiedendo un totale di $n(n-1)/2$ confronti. Ovviamente, l'algoritmo min() è preferibile, in quanto richiede solo $n-1$ confronti.

Per quanto riguarda RICERCA, abbiamo descritto informalmente un algoritmo che richiede $n$ confronti, ma la ricerca binaria è più efficiente. Ogni chiamata richiede infatti 2 confronti fra elementi del vettore (nel costrutto **if-else if**). Quindi, la domanda diventa: quante volte la funzione richiamerà se stessa? Ad ogni chiamata, il vettore da chiamare si dimezza. Quindi il numero di chiamate richieste nel caso il valore cercato non sia presente è $\lfloor \log n \rfloor$. Se invece è presente, potrebbe essere incontrato prima. Quindi il numero totale di confronti sarà al massimo $2\lfloor \log n \rfloor$.

Nel nostro pseudocodice, abbiamo assunto che i dati appartengano al tipo **int**, per cui le operazioni di confronto sono primitive e possono essere eseguite in modo estremamente veloce — tipicamente in meno di un miliardesimo di secondo. Tuttavia, in contesti reali, i dati potrebbero essere strutture complesse, come stringhe molto lunghe o record composti, in cui confrontare due elementi può richiedere un tempo significativamente maggiore. Per questo motivo, quando analizziamo un algoritmo, non ci limitiamo a considerare il tempo di calcolo in senso assoluto (secondi, minuti, ecc.), ma preferiamo contare operazioni elementari come i confronti, che rappresentano una misura più stabile e indipendente dalla piattaforma.

In secondo luogo, il tempo di calcolo effettivo dipende fortemente dalla piattaforma utilizzata per la misura; la frequenza del processore, la memoria cache di primo, secondo e terzo livello, la velocità del bus e così via. Ognuno di questi parametri può influenzare il tempo di calcolo, in maniera sottile.

Concludiamo con una domanda, a cui daremo risposta nel prossimo capitolo: possiamo fare meglio di così? Esistono degli algoritmi "migliori" (più efficienti) per i problemi proposti?

**Ulteriori proprietà**. Correttezza ed efficienza esauriscono le proprietà "desiderabili" degli algoritmi? Da un punto di vista più ampio, proprio dell'ingegneria del software, potremmo aggiungere proprietà sicuramente importanti quali semplicità, modularità, manutenibilità, espandibilità; ma anche aspetti particolari della correttezza quali sicurezza e robustezza (capacità di gestire dati in

ingresso non previsti). Cercheremo di realizzare algoritmi che soddisfino queste proprietà, ma non ne discuteremo direttamente, ritenendo che un corso di ingegneria del software sia la sede più adatta.

Una considerazione è tuttavia importante. Spesso, molte di queste proprietà hanno un costo, in termini di prestazioni. Scrivere codice modulare può richiedere un costo aggiuntivo per la gestione delle chiamate di procedura; utilizzare un linguaggio robusto ma interpretato come Java può rallentare l'esecuzione del codice; e la lista di esempi potrebbe continuare. Progettare algoritmi efficienti è un prerequisito fondamentale per poter "pagare" questi costi. Se il vostro algoritmo utilizza un numero $\log n$ di passi invece che $n$, il fatto che l'esecuzione sia rallentata di un piccolo fattore costante perde completamente importanza.

**Un po' di storia**

La Macchina Analitica, progettata da Babbage a partire dal 1833 e mai effettivamente completata, doveva essere il primo apparecchio meccanico per il calcolo programmabile, dotato di unità aritmetico-logica, memoria interna, controllo condizionale e cicli — anticipando di oltre un secolo l'era dei computer moderni. Nel suo saggio *Passages from the Life of a Philosopher*, Charles Babbage scriveva già nel 1864:

> *"As soon as an Analytical Engine exists, it will necessarily guide the future course of science. Whenever any result is sought by its aid, the question will then arise — By what course of calculation can these results be arrived at by the machine in the shortest time?"*

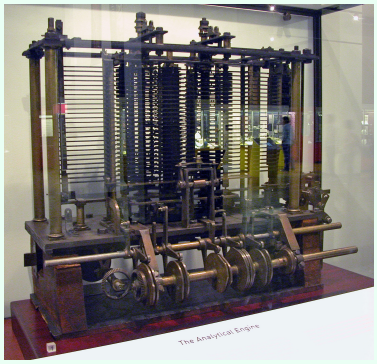


Figura 1.3: Modello della macchina analitica, Museo di Londra, foto Bruno Barral

## 1.4 Progettazione di algoritmi

Per progettare algoritmi corretti ed efficienti, un buon progettista di algoritmi deve saper padroneggiare due distinti insiemi di conoscenze:

- deve essere in grado di identificare il problema da risolvere fra una classe più ampia di problemi, e utilizzare l'enorme catalogo di soluzioni sviluppate negli ultimi cinquant'anni per evitare di partire ogni volta da zero; come si dice, deve "salire sulle spalle dei giganti". Non conoscere la ricerca binaria, per esempio, è un grave peccato per chiunque voglia scrivere algoritmi efficienti;
- deve saper utilizzare le principali tecniche algoritmiche, quali *divide-et-impera*, *programmazione dinamica*, *tecnica greedy*, *backtrack*, *ricerca locale* e *approcci probalistici*. Come primo assaggio, sappiate che la ricerca binaria appartiene alla classe *divide-et-impera*.

In questo testo, svilupperemo tali argomenti in due filoni principali. La Parte II discuterà l'organizzazione di grandi quantità di dati nella memoria di un calcolatore affinché possano essere reperiti facilmente ed efficientemente. Questo è un problema fondamentale, per il quale tante soluzioni differenti (strutture di dati) sono state proposte, ognuna specializzata per un diverso utilizzo. La Parte III si concentrerà sulle principali tecniche di progettazione, mentre la Parte IV complementerà la Parte III, discutendo tecniche speciali per risolvere in modo subottimo problemi inerentemente intrattabili per i quali non si conoscono algoritmi efficienti.

**Un po' di storia** Il primo algoritmo di cui si abbia documentazione si trova nel papiro egizio di Ahmes (Figura 1.4), risalente circa al 1650 a.C. (ma tratto per dichiarazione di Ahmes da un papiro ancora più antico) e conservato al British Museum di Londra.

Il papiro contiene un algoritmo per la moltiplicazione. Siano $a$ e $b$ i due fattori (interi). Partendo da $p = 0$,

- si somma $b$ a $p$ se $a$ è dispari;
- si dimezza $a$ (prendendone la parte intera);
- si raddoppia $b$.

ripetendo finché non risulti $a = 0$. Al termine del procedimento $p$ ha valore $a \cdot b$.

Figura 1.4: Papiro di Ahmes (British Museum; foto Paul James Cowie).

Per provare la correttezza, basta considerare le due seguenti proprietà: se $a$ è un numero pari allora $a \cdot b = (a/2) \cdot 2b$, se viceversa $a$ è dispari $a \cdot b = ((a-1)/2) \cdot 2b + b$. Verificando che l'algoritmo altro non fa se non passare per rappresentazioni equivalenti del problema in cui $a$ decresce continuamente fino a giungere certamente a zero in un numero finito di passi, si prova la correttezza.

Il fatto sorprendente è che tale algoritmo di moltiplicazione viene usato tuttoggi nei circuiti delle unità aritmetico/logiche dei calcolatori elettronici. Infatti, rappresentando gli interi in base 2, le operazioni di verifica della parità, dimezzamento e raddoppio sono elementari. Per verificare la parità basta controllare il valore del bit meno significativo, mentre per dimezzare (o raddoppiare) basta traslare a destra (o a sinistra) di una posizione tutti i bit.