# Introduzione alla Ricorsione Computazionale

Prof. Simone Faro
Dispensa didattica per il corso di Algoritmi e Laboratorio
Università di Catania

Anno Accademico 2025–2026

## Indice

# 1 Concetto di Ricorsione

La **ricorsione** rappresenta uno dei concetti più eleganti e profondi della programmazione, oltre che uno dei più antichi nella storia della logica e della matematica. Essa si fonda sull'idea che un problema complesso possa essere affrontato non direttamente nella sua interezza, ma scomponendolo in una serie di *sottoproblemi* più semplici, ciascuno dei quali conserva la stessa natura e struttura del problema originario, ma su scala ridotta. In altre parole, la ricorsione è un modo per definire qualcosa — un processo, una funzione o un concetto — in termini di sé stesso.

Dal punto di vista operativo, un programma ricorsivo è una funzione o procedura che, durante la propria esecuzione, richiama sé stessa per risolvere versioni più piccole dello stesso problema. Questo meccanismo, apparentemente circolare, è in realtà estremamente potente, a condizione che esista un punto di arresto ben definito: una *condizione base* che interrompa la catena di richiami quando il problema raggiunge una dimensione elementare, ovvero un caso la cui soluzione è nota o immediata.

L'idea alla base della ricorsione può essere illustrata con un semplice principio: per risolvere un problema complesso $P$, il programmatore non cerca di affrontarlo direttamente, ma di suddividerlo in $K$ sottoproblemi più piccoli

$$P_1, P_2, \ldots, P_K.$$

Ognuno di questi sottoproblemi viene poi risolto (o, più precisamente, si *assume* che possa essere risolto) attraverso una chiamata alla stessa funzione che sta eseguendo. Le soluzioni parziali così ottenute vengono infine combinate per ricostruire la soluzione complessiva $S$ del problema principale:

$$S = f\big(S_1, S_2, \ldots, S_K\big)$$

dove $S_i$ rappresenta la soluzione del sottoproblema $P_i$.

Questo approccio, che può sembrare controintuitivo a prima vista, si basa su una forma di fiducia logica che il matematico tedesco David Hilbert descriveva come un "principio di completamento induttivo": si presume, cioè, che il problema più piccolo possa essere risolto, e su questa base si costruisce il procedimento generale. Non a caso, la ricorsione in informatica affonda le proprie radici nella stessa idea logica che sta alla base del *principio di induzione matematica*: se sappiamo risolvere un problema nel caso più semplice e sappiamo come passare dal caso $n-1$ al caso $n$, allora possiamo risolvere qualsiasi istanza del problema.

Un esempio classico e intuitivo di questo meccanismo è il calcolo del *fattoriale* di un numero intero non negativo $n$. Definendo

$$n! = \begin{cases} 1 & \text{se } n = 0, \\ n \cdot (n-1)! & \text{se } n > 0, \end{cases}$$

si afferma in sostanza che per calcolare $n!$ non è necessario conoscere direttamente il risultato, ma basta sapere come calcolare il fattoriale di un numero più piccolo, $n-1$. Questa definizione *auto-riferita* è perfettamente lecita e, anzi, diventa computazionalmente potente, purché il caso base $0! = 1$ interrompa la catena di richiami e garantisca la terminazione.

La chiave del pensiero ricorsivo risiede dunque nella *riduzione del problema*: invece di cercare una soluzione diretta e immediata, si riformula il problema in termini più semplici e accessibili. Da questa prospettiva emergono due aspetti fondamentali:

- la **suddivisione** del problema principale in sottoproblemi di minore complessità;
- la **ricombinazione** dei risultati parziali in una soluzione complessiva.

Questa duplice operazione — scomporre e poi ricomporre — è ciò che rende la ricorsione tanto elegante quanto potente. Non sorprende che molti degli algoritmi più noti e studiati, come il *Merge Sort*, il *Quick Sort* o la *ricerca binaria*, si fondino proprio su questa logica di frammentazione e ricostruzione. L'approccio ricorsivo permette di passare da una visione monolitica del calcolo a una visione *gerarchica e modulare*, in cui ogni parte contribuisce alla soluzione complessiva risolvendo una versione ridotta del problema iniziale.

In definitiva, la ricorsione non è soltanto una tecnica di programmazione, ma un vero e proprio *modo di pensare*: un linguaggio concettuale che insegna a vedere la complessità come il risultato di una struttura ripetitiva, e a dominare i problemi difficili riducendoli progressivamente ai loro elementi più semplici.

Nel mondo dell'informatica, e in particolare nella cultura degli *hacker* e dei programmatori più esperti, trovare una soluzione ricorsiva a un problema è spesso considerato un segno di ingegno e di raffinatezza concettuale. Una funzione ricorsiva non solo risolve un compito in modo corretto, ma lo fa rivelando una comprensione più profonda della struttura del problema stesso. Per questo motivo, una soluzione ricorsiva è vista come una forma di *eleganza algoritmica*: essa riduce il codice al suo nucleo essenziale, elimina la necessità di meccanismi iterativi espliciti e mette in luce la logica pura che governa la trasformazione dei dati.

Nella tradizione degli hacker, questa eleganza non è fine a sé stessa, ma rappresenta una manifestazione di intelligenza creativa e di padronanza del linguaggio di programmazione. Scrivere un algoritmo ricorsivo significa saper vedere il problema da un punto di vista astratto, riconoscendo al suo interno un pattern autoripetitivo che può essere espresso in modo compatto e naturale. Molti programmatori esperti provano un senso di soddisfazione estetica nel trovare una soluzione ricorsiva che, pur essendo breve e concettualmente semplice, riesce a risolvere problemi complessi attraverso la pura forza della definizione e dell'autoreferenza.

Non è un caso che in molte comunità informatiche — dai primi laboratori del MIT negli anni Settanta fino alle moderne piattaforme open source — la capacità di individuare e implementare una soluzione ricorsiva sia considerata una prova di *padronanza del pensiero computazionale*. La ricorsione, in questo senso, non è soltanto una tecnica, ma un modo per dimostrare come la chiarezza, la sintesi e la bellezza possano coesistere in un frammento di codice.

## 1.1 Non solo informatica

La ricorsione, tuttavia, non appartiene soltanto al dominio della programmazione o della matematica: essa è un modo generale di pensare, un principio strutturale che si manifesta in numerosi ambiti della conoscenza e della creatività umana. Nella linguistica, ad esempio, Noam Chomsky individuò nella ricorsione la proprietà fondamentale che distingue il linguaggio umano, ovvero la capacità di incorporare frasi all'interno di altre frasi in modo potenzialmente infinito. In biologia, le strutture autoripetitive dei frattali o delle ramificazioni vascolari mostrano come la natura stessa tenda a organizzarsi secondo schemi ricorsivi.

Anche nel mondo dell'informatica e della cultura digitale la ricorsione ha lasciato tracce curiose e affascinanti. Un esempio celebre è il nome del progetto GNU, ideato da Richard Stallman negli anni Ottanta, che sta per *"GNU's Not Unix"* — letteralmente "GNU non è Unix" — dove l'acronimo si riferisce a sé stesso in una definizione che non termina mai, proprio come una funzione ricorsiva. Un altro esempio è il classico messaggio nascosto nei motori di ricerca: digitando la parola "recursion" su Google, il sistema suggerisce ironicamente "Forse cercavi: recursion", invitando l'utente a ripetere la stessa operazione all'infinito.

Questi esempi, per quanto giocosi, rivelano una verità profonda: la ricorsione non è soltanto una tecnica computazionale, ma un modello mentale e culturale, un modo di concepire il mondo come un insieme di strutture che si riflettono su sé stesse. In questo senso, essa rappresenta una delle forme più potenti e universali della riflessione umana, capace di unire logica, linguaggio e creatività in un'unica visione concettuale.

## 1.2 Strutture dati e definizioni ricorsive

La ricorsione non riguarda soltanto le funzioni o gli algoritmi: anche molte **strutture dati** fondamentali in informatica possiedono una natura intrinsecamente ricorsiva. Questo significa che la loro stessa definizione può essere espressa in termini ricorsivi, e che molte delle operazioni che le riguardano — come la ricerca, la scansione o la modifica — possono essere descritte naturalmente attraverso procedure ricorsive.

Si consideri, ad esempio, una **lista**. Una lista può essere definita in modo ricorsivo come:

$$\text{Lista} = \begin{cases} \text{Lista vuota}, & \text{oppure} \\ (\text{Elemento},\ \text{Lista}) & \text{se contiene almeno un elemento.} \end{cases}$$

In altre parole, una lista non è altro che un *elemento iniziale* seguito da un'altra lista (più piccola) dello stesso tipo. Questa definizione è auto-riferita e mostra immediatamente la struttura ricorsiva del concetto: ogni lista può essere vista come la combinazione di una testa (head) e di una coda (tail), dove la coda è a sua volta una lista.

La stessa idea si applica a strutture apparentemente più semplici, come un **array**. Un array di lunghezza $n$ può essere definito come un elemento iniziale seguito da un array di lunghezza $n-1$:

$$A_n = \{a_0\} \cup A_{n-1}.$$

Molte operazioni sugli array, come il calcolo della somma degli elementi o la ricerca del massimo, possono infatti essere descritte naturalmente come processi ricorsivi che riducono progressivamente la dimensione della struttura fino a raggiungere il caso base $n = 0$.

Un caso ancora più emblematico è rappresentato dagli **alberi**. Un albero è per definizione una struttura dati ricorsiva:

$$\text{Albero} = \begin{cases} \text{nodo vuoto}, & \text{(albero nullo)} \\ (\text{valore},\ \text{sottoalbero sinistro},\ \text{sottoalbero destro}) & \text{(albero non vuoto).} \end{cases}$$

Ogni nodo dell'albero può essere considerato come la radice di un nuovo albero, composto dai suoi sottoalberi. Per questo motivo, la maggior parte delle operazioni sugli alberi — come le visite in *pre-order*, *in-order* e *post-order*, o la ricerca in un *binary search tree* — sono formulate in modo ricorsivo.

Questa caratteristica non è solo un dettaglio tecnico, ma un riflesso profondo della natura gerarchica di molte strutture logiche e computazionali. Ogni elemento di una struttura ricorsiva può essere visto come una copia ridotta dell'intera struttura: un principio di *autosimilarità* che ricorda le costruzioni dei frattali in matematica e in natura. L'albero genealogico, ad esempio, è una forma intuitiva di struttura ricorsiva: ogni persona è un nodo che a sua volta può essere radice di altri due nodi, i propri discendenti, e così via, all'infinito.

Per queste ragioni, la ricorsione non è soltanto un modo per *operare* sulle strutture dati, ma anche per *definirle*. Quando una struttura è definita ricorsivamente, la sua manipolazione ricorsiva risulta non solo naturale, ma anche concettualmente più chiara e formalmente elegante. Questo spiega perché molti algoritmi fondamentali su liste e alberi — dalla somma degli elementi alla ricerca, dalla copia alla cancellazione — assumono in modo quasi spontaneo una forma ricorsiva.

## 1.3 Struttura generale di un programma ricorsivo

Ogni programma ricorsivo si fonda su una struttura concettuale ben precisa, che ne garantisce la correttezza logica e la terminazione. Sebbene le applicazioni possano variare enormemente — dal calcolo di funzioni matematiche all'elaborazione di strutture dati complesse — tutti i programmi ricorsivi condividono tre elementi essenziali e irrinunciabili, che ne costituiscono l'ossatura logica.

**1. Il caso base.** Ogni ricorsione deve prevedere almeno una condizione che ne interrompa la prosecuzione infinita. Questa condizione, detta *caso base*, rappresenta il punto in cui il problema diventa talmente semplice da poter essere risolto direttamente, senza ulteriori richiami ricorsivi. Dal punto di vista matematico, essa corrisponde alla condizione di terminazione dell'equazione definita ricorsivamente. Ad esempio, nella funzione fattoriale,

$$F(n) = \begin{cases} 1 & \text{se } n = 0, \\ n \cdot F(n-1) & \text{se } n > 0, \end{cases}$$

il caso base $F(0) = 1$ impedisce alla funzione di continuare a richiamare sé stessa indefinitamente, chiudendo la catena ricorsiva.

**2. Il passo ricorsivo.** Nel passo ricorsivo, la funzione richiama sé stessa per risolvere uno o più sottoproblemi di dimensione minore. Si tratta del cuore del processo ricorsivo: il problema originario viene progressivamente ridotto a casi più semplici, fino a raggiungere il caso base. Se indichiamo con $P$ il problema principale e con $P_1, P_2, \ldots, P_K$ i sottoproblemi derivati, il passo ricorsivo può essere espresso formalmente come:

$$S = \text{Combina}\big(F(P_1), F(P_2), \ldots, F(P_K)\big),$$

dove $S$ rappresenta la soluzione finale. Questa espressione mostra come la ricorsione permetta di affrontare problemi complessi attraverso la scomposizione sistematica e la ripetizione controllata dello stesso schema risolutivo.

**3. La combinazione dei risultati.** Una volta risolti i sottoproblemi, occorre un meccanismo per ricomporre le soluzioni parziali e ottenere la soluzione complessiva del problema principale. Questo processo, detto *fase di combinazione*, è spesso ciò che distingue una ricorsione semplice da una più sofisticata. Nel *Merge Sort*, ad esempio, la fase di combinazione consiste nel *merging* di due sottoarray ordinati in un singolo array ordinato,

operazione che richiede tempo lineare. In altre applicazioni, come il calcolo del fattoriale o della potenza, la combinazione si riduce a una semplice moltiplicazione o somma.

Possiamo quindi rappresentare formalmente una funzione ricorsiva $F(n)$ nel seguente modo generale:

$$F(n) = \begin{cases} S_0, & \text{se } n \text{ soddisfa la condizione base;} \\ \text{Combina}\big(F(n_1), F(n_2), \ldots, F(n_K)\big), & \text{altrimenti.} \end{cases}$$

Qui $S_0$ rappresenta la soluzione immediata del caso base, mentre i parametri $n_1, n_2, \ldots, n_K$ denotano le versioni ridotte del problema.

Dal punto di vista operativo, l'esecuzione di un programma ricorsivo può essere immaginata come una serie di scatole annidate o di specchi riflettenti: ogni chiamata alla funzione apre un nuovo livello di esecuzione, sospendendo temporaneamente il calcolo corrente fino a quando la funzione invocata non restituisce il suo risultato. Questa dinamica è gestita automaticamente dal *call stack* del linguaggio di programmazione, una struttura a pila (*stack*) che conserva lo stato di ciascuna chiamata. Ogni nuova invocazione della funzione viene accodata in cima alla pila, e le chiamate vengono risolte in ordine inverso, dal caso base verso la superficie. Questo spiega perché, in una funzione ricorsiva ben progettata, la profondità della ricorsione deve sempre essere finita: in caso contrario, la pila si riempirebbe indefinitamente, producendo un errore di *stack overflow*.

In conclusione, la struttura di un programma ricorsivo rappresenta un equilibrio delicato tra astrazione logica e controllo operativo. Il caso base garantisce la terminazione, il passo ricorsivo fornisce la riduzione del problema, e la combinazione dei risultati permette di risalire progressivamente verso la soluzione completa. Insieme, questi tre elementi costituiscono la grammatica universale di ogni algoritmo ricorsivo.

## 1.4 La dimensione del problema

Un aspetto cruciale nella progettazione di un algoritmo ricorsivo riguarda la definizione della **dimensione del problema**, spesso indicata con la variabile $n$. Essa rappresenta il parametro che misura la complessità o la grandezza dell'istanza da risolvere, e che determina quante volte la funzione verrà richiamata su versioni più piccole del problema. Tuttavia, la scelta di cosa debba rappresentare esattamente $n$ non è sempre immediata né univoca.

In alcuni casi, la dimensione del problema è intuitiva: per esempio, nel calcolo del fattoriale o della potenza, $n$ coincide semplicemente con un numero intero che viene progressivamente decrementato fino a raggiungere lo zero. In altri problemi, però, la dimensione può essere interpretata in modi diversi. Nel caso di un algoritmo su un *array*, ad esempio, $n$ può rappresentare la lunghezza dell'array, ma anche l'indice corrente che delimita la porzione dell'array ancora da elaborare. Negli algoritmi su *liste* o *alberi*, la dimensione può essere misurata dal numero di nodi, dall'altezza della struttura, o dal livello di profondità su cui si sta operando.

La scelta del parametro che rappresenta la dimensione del problema non è una semplice formalità: essa influisce profondamente sul comportamento dell'algoritmo e sulla sua *complessità computazionale*. Due soluzioni ricorsive dello stesso problema, formulate sulla base di diverse interpretazioni di $n$, possono avere tempi di esecuzione molto diversi.

In generale, comprendere cosa rappresenti la dimensione $n$ del problema equivale a comprendere la *natura stessa della ricorsione*. Ogni formulazione ricorsiva è una scelta di prospettiva: essa definisce non solo come il problema viene scomposto, ma anche come il tempo e lo spazio dell'esecuzione vengono distribuiti lungo la catena delle chiamate. Per questo motivo, la capacità di individuare correttamente il parametro di ricorsione — e di valutarne gli effetti sulla complessità — è una delle abilità più importanti nella progettazione di algoritmi efficienti.

## 1.5 L'equazione di ricorrenza

Ogni algoritmo ricorsivo, oltre ad avere una struttura logica ben definita, possiede anche una propria **struttura quantitativa**: essa descrive come il costo del calcolo cresce in funzione della dimensione del problema. Questa struttura prende forma in una *equazione di ricorrenza*, uno strumento matematico che consente di tradurre il comportamento di un algoritmo ricorsivo in un modello analitico, utile per prevederne la complessità.

Un'equazione di ricorrenza esprime il tempo di esecuzione di un algoritmo ricorsivo come funzione del tempo necessario per risolvere uno o più sottoproblemi più piccoli, più un termine che rappresenta il costo delle operazioni non ricorsive (ad esempio la divisione del problema o la combinazione dei risultati). In generale, se indichiamo con $T(n)$ il tempo necessario per risolvere un problema di dimensione $n$, possiamo scrivere:

$$T(n) = \sum_{i=1}^{K} T(n_i) + f(n),$$

dove $T(n_i)$ rappresenta il tempo necessario per risolvere il $i$-esimo sottoproblema, di dimensione $n_i < n$; e $f(n)$ rappresenta il tempo impiegato per la **fase di divisione** e di **riunificazione**, ovvero tutte le operazioni che non comportano ulteriori chiamate ricorsive.

In altre parole, l'equazione di ricorrenza è la *traccia numerica* del processo ricorsivo: ogni livello dell'albero di ricorsione contribuisce a costruire un termine della somma complessiva, e la soluzione finale dell'equazione ci fornisce il tempo totale richiesto dall'algoritmo.

Per costruire l'equazione di ricorrenza corrispondente a un algoritmo, occorre osservare come esso si comporta rispetto alla dimensione del problema. Ogni chiamata ricorsiva può essere vista come un nodo dell'albero di ricorsione, e il numero di nodi totali (ponderato dal costo di ciascun livello) determina il tempo totale.

Nel caso di una funzione come il **fattoriale**, la ricorsione riduce la dimensione del problema di 1 ad ogni chiamata. Poiché ad ogni livello viene eseguita una sola moltiplicazione, possiamo scrivere:

$$T(n) = T(n-1) + O(1),$$

dove il termine $O(1)$ rappresenta il costo costante dell'operazione aritmetica. Risolvendo questa ricorrenza per somma telescopica otteniamo:

$$T(n) = O(n),$$

che riflette il fatto che l'algoritmo effettua una chiamata per ogni valore intero da $n$ a 0.

Se invece consideriamo una ricorsione binaria, come quella che calcola la somma di un array dividendolo in due metà, il tempo di esecuzione soddisfa:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(1),$$

poiché ogni chiamata genera due sottoproblemi di metà dimensione, e la fase di combinazione richiede solo un tempo costante per sommare i risultati. Risolvendola otteniamo ancora $T(n) = O(n)$, ma con una struttura interna profondamente diversa: l'albero di ricorsione, in questo caso, ha altezza $\log_2 n$ e contiene $n$ nodi foglia, ognuno con costo unitario.

Un terzo esempio, più complesso e didatticamente centrale, è quello del **Merge Sort**. In questo algoritmo, ogni livello di ricorsione comporta una divisione in due sottoarray (come nel caso precedente), ma la fase di riunificazione — il merging — richiede un tempo lineare $O(n)$ per combinare i due sottoarray già ordinati. L'equazione di ricorrenza corrispondente diventa quindi:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n),$$

che si risolve, applicando il teorema del maestro, in:

$$T(n) = O(n \log n).$$

Questo risultato, uno dei più noti nell'analisi algoritmica, mostra come la ricorsione possa amplificare o ridurre l'efficienza di un algoritmo a seconda del rapporto fra il numero di sottoproblemi e il costo della combinazione.

Dal punto di vista concettuale, un'equazione di ricorrenza è una **descrizione dinamica** del comportamento di un algoritmo: essa mostra come il problema si trasforma su sé stesso nel tempo.

Ogni termine dell'equazione rappresenta un livello dell'albero di chiamate: la successione delle riduzioni, delle diramazioni e delle ricomposizioni che costituiscono l'ossatura dell'algoritmo. Analizzare la ricorrenza significa quindi "osservare" la ricorsione da un punto di vista quantitativo, misurando quanto lavoro viene svolto a ciascun livello e come esso si accumula globalmente.

In modo più intuitivo, l'equazione di ricorrenza può essere vista come una *metafora aritmetica del pensiero ricorsivo*: mentre la funzione ricorsiva si definisce in termini di sé stessa, l'equazione ne misura l'impatto computazionale, traducendo la logica della decomposizione in una logica di crescita. Il passaggio dal codice alla formula, dalla procedura alla ricorrenza, rappresenta quindi il momento in cui la ricorsione da idea concettuale diventa oggetto di analisi matematica.

Una delle ragioni per cui lo studio delle equazioni di ricorrenza è fondamentale risiede nel loro potere predittivo. Mentre il codice mostra cosa accade *localmente*, passo dopo passo, l'equazione di ricorrenza consente di capire cosa accade *globalmente*, lungo l'intero processo. Essa rivela, per esempio, se un algoritmo cresce in modo lineare, logaritmico o esponenziale, e se la ricorsione scelta è sostenibile in termini di tempo e memoria.

Infine, l'analisi delle equazioni di ricorrenza consente anche di confrontare diverse soluzioni ricorsive dello stesso problema. Due funzioni che producono lo stesso risultato possono avere ricorrenze molto diverse: una lineare, l'altra logaritmica, una con costo costante

di combinazione, l'altra con costo lineare. Saper riconoscere queste differenze è ciò che distingue una ricorsione concettualmente corretta da una ricorsione anche *efficiente*.

In sintesi, l'equazione di ricorrenza è il ponte tra la logica e la misura della ricorsione: un modello che unisce il pensiero algoritmico alla sua analisi formale. Essa consente di comprendere non solo come un problema venga suddiviso e risolto, ma anche quanto "costa" farlo — in termini di tempo, di memoria e, in ultima analisi, di eleganza computazionale.

# 2 Esempi introduttivi di ricorsione

Per comprendere a fondo il funzionamento della ricorsione, è utile analizzare alcuni esempi elementari ma emblematici. Essi mostrano come un problema possa essere suddiviso in versioni più semplici di sé stesso (fase di *divisione*) e come le soluzioni parziali possano essere combinate per ottenere la soluzione finale (fase di *riunificazione*). Le tre funzioni che seguono — fattoriale, moltiplicazione e potenza — rappresentano casi classici e intuitivi di questa dinamica.

## 2.1 Il fattoriale

Il calcolo del **fattoriale** di un numero intero non negativo $n$ è forse l'esempio più iconico di funzione ricorsiva. Il fattoriale, indicato con $n!$, rappresenta il prodotto di tutti i numeri interi positivi minori o uguali a $n$:

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1.$$

La definizione ricorsiva di questa funzione è:

$$n! = \begin{cases} 1 & \text{se } n = 0, \\ n \cdot (n-1)! & \text{se } n > 0. \end{cases}$$

Nel caso del fattoriale, la **fase di divisione** consiste nel ridurre il problema $F(n)$ a un sottoproblema di dimensione minore, $F(n-1)$. Ogni chiamata produce quindi una nuova versione del problema in cui la dimensione $n$ si riduce di uno. La **fase di riunificazione**, invece, avviene durante il ritorno delle chiamate: ciascuna funzione moltiplica il valore ottenuto dalla chiamata successiva per il proprio parametro $n$. In questo modo, la soluzione complessiva si costruisce a ritroso, dal caso base verso l'origine.

```
1 int fattoriale(int n) {
2     if (n == 0)
3         return 1;                    // caso base
4     else
5         return n * fattoriale(n - 1); // divisione e
              riunificazione
6 }
```

L'albero di ricorsione per il calcolo di 4! mostra graficamente questa struttura:


Ogni nodo genera una singola chiamata discendente (fase di divisione) e restituisce poi il risultato moltiplicato per il proprio valore (fase di riunificazione). L'albero ha altezza $n$ e contiene $n + 1$ nodi, uno per ogni chiamata ricorsiva.

## 2.2 Moltiplicazione come somma ripetuta

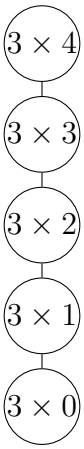
Un secondo esempio classico è la **moltiplicazione** definita come somma ripetuta. Per due numeri interi non negativi $a$ e $b$, possiamo esprimere il prodotto come:

$$a \times b = \begin{cases} 0 & \text{se } b = 0, \\ a + (a \times (b-1)) & \text{se } b > 0. \end{cases}$$

In questo caso, la **fase di divisione** riduce il secondo argomento $b$ di una unità ad ogni chiamata: il problema $P(a, b)$ viene trasformato in un sottoproblema $P(a, b-1)$, di dimensione più piccola. La **fase di riunificazione** consiste nel sommare $a$ al risultato ottenuto dalla chiamata successiva, accumulando progressivamente il valore finale.

```
1 int moltiplica(int a, int b) {
2     if (b == 0)
3         return 0;                          // caso base
4     else
5         return a + moltiplica(a, b - 1); // divisione e
              riunificazione
6 }
```

L'albero di ricorsione per il calcolo di $3 \times 4$ è mostrato di seguito:



Ogni livello dell'albero rappresenta una nuova sottrazione di 1 a $b$, fino al caso base. Durante la risalita, ciascun nodo somma $a$ al risultato parziale ottenuto dal livello sottostante, ricomponendo il valore totale $3 + 3 + 3 + 3 = 12$.

## 2.3 Potenza come moltiplicazione ripetuta

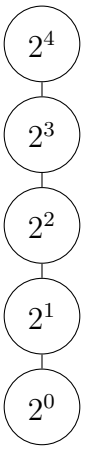
Infine, la **potenza** può essere definita come moltiplicazione ripetuta. Per due numeri $a$ e $n$:

$$a^n = \begin{cases} 1 & \text{se } n = 0, \\ a \times a^{n-1} & \text{se } n > 0. \end{cases}$$

Qui la **fase di divisione** consiste nel ridurre progressivamente l'esponente $n$ fino a raggiungere lo zero, mentre la **fase di riunificazione** avviene moltiplicando ogni valore intermedio per $a$ nel momento in cui le chiamate ritornano. Questo modello è analogo a quello del fattoriale, ma la relazione fra i termini è più astratta.

```
1 int potenza(int a, int n) {
2     if (n == 0)
3         return 1;                        // caso base
4     else
5         return a * potenza(a, n - 1);  // divisione e
              riunificazione
6 }
```

L'albero di ricorsione per il calcolo di $2^4$ ha la seguente forma:




Come nel caso del fattoriale, ogni chiamata genera un unico ramo discendente, e la soluzione complessiva emerge durante la fase di ritorno, moltiplicando i risultati parziali:

$$2^4 = 2 \times (2 \times (2 \times (2 \times 1))).$$

Questi tre esempi mostrano un principio comune: la ricorsione separa il processo in due momenti distinti ma complementari. Durante la *fase di divisione*, il problema viene scomposto in versioni più semplici; durante la *fase di riunificazione*, le soluzioni parziali vengono ricombinate fino a ottenere il risultato finale. Questa struttura logica è ciò che ritroveremo in forma più articolata nei grandi algoritmi ricorsivi di ordinamento e ricerca.

# 3 Problemi di somma e soglia su array

Nel passaggio dai problemi puramente numerici alle strutture dati più complesse, gli **array** offrono un contesto ideale per applicare e comprendere la logica ricorsiva. Un array, infatti, può essere visto come una sequenza di elementi tra loro omogenei, che si presta naturalmente a una suddivisione progressiva in parti più piccole.

Le operazioni definite su un array – come l'elaborazione, il conteggio o il controllo dei suoi valori – possono essere descritte ricorsivamente in molti modi diversi. Ciò che resta invariato è l'idea di fondo: ridurre il problema generale a uno di dimensione minore e combinare i risultati parziali per ottenere la soluzione complessiva.

Nei prossimi esempi vedremo come questa impostazione possa essere utilizzata per risolvere problemi di natura sia quantitativa sia logica, mostrando la versatilità e la potenza espressiva dell'approccio ricorsivo applicato alle sequenze di dati.

## 3.1 Somma dei valori di un array

Dopo aver osservato la ricorsione su problemi puramente numerici, possiamo ora applicare lo stesso principio a strutture dati più complesse, come gli **array**. Un array non è altro che una sequenza di elementi disposti in memoria contigua, e la sua natura ordinata lo rende particolarmente adatto a essere elaborato in modo ricorsivo.

Molti problemi sugli array possono infatti essere risolti secondo due schemi generali: una **ricorsione di coda**, in cui l'array viene ridotto progressivamente di un elemento alla volta; una **doppia ricorsione**, in cui l'array viene suddiviso in due sottoarray di dimensioni più piccole, che vengono risolti separatamente e poi ricombinati.

In questa sezione consideriamo il problema di calcolare la somma di tutti gli elementi di un array $A$ di lunghezza $n$. Per rendere l'idea in modo più concreto, supponiamo di avere un array di interi:

$$A = [3,\ 7,\ 2,\ 9,\ 4].$$

In questo caso, la somma totale degli elementi sarà:

$$3 + 7 + 2 + 9 + 4 = 25.$$

L'obiettivo è dunque quello di scrivere una procedura in grado di restituire, per qualsiasi array $A$ di dimensione $n$, il valore:

$$S(A, n) = \sum_{i=0}^{n-1} A[i].$$

Si tratta di un problema apparentemente semplice, ma molto utile per comprendere la differenza tra un approccio *iterativo* e uno *ricorsivo*. Nel primo caso il programma accumula progressivamente la somma in una variabile durante l'iterazione sugli elementi dell'array; nel secondo, invece, la somma viene ottenuta suddividendo il problema in sottoproblemi più piccoli, ciascuno dei quali calcola la somma di una parte dell'array.

Prima di affrontare il problema con un approccio ricorsivo, è utile ricordare come esso venga risolto tradizionalmente in modo **iterativo**. Supponiamo di avere un array $A$ contenente $n$ elementi numerici. La somma di tutti gli elementi può essere calcolata percorrendo l'array da sinistra verso destra e accumulando progressivamente i valori in

una variabile ausiliaria, che chiameremo `somma`. Al termine del ciclo, la variabile conterrà il risultato desiderato.

```
int sommaIterativa(int A[], int n) {
    int somma = 0;
    for (int i = 0; i < n; i++) {
        somma += A[i];   // accumula il valore corrente
    }
    return somma;
}
```

Questo approccio è lineare, semplice e intuitivo: il programma mantiene esplicitamente lo *stato intermedio* della computazione (la variabile `somma`) e lo aggiorna passo dopo passo. Dal punto di vista concettuale, l'iterazione esprime un processo di tipo sequenziale, in cui ciascun passo dipende dal precedente, ma non si richiama mai sé stesso.

L'idea alla base della **ricorsione** è di natura diversa. Invece di accumulare progressivamente i risultati, la ricorsione tende a scomporre il problema in sottoproblemi più piccoli, ciascuno dei quali viene risolto indipendentemente e poi ricombinato nella fase di ritorno. Nel caso della somma di un array, ciò significa pensare la somma non come un processo iterativo che "scorre" l'array, ma come una funzione che si richiama su versioni ridotte dello stesso array, delegando alle chiamate successive la responsabilità di completare il calcolo.

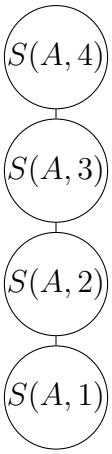
L'approccio ricorsivo più diretto consiste nel ridurre la dimensione dell'array di un elemento ad ogni passo, fino a raggiungere il caso base in cui l'array contiene un solo elemento. Formalmente possiamo scrivere:

$$S(A, n) = \begin{cases} A[0] & \text{se } n = 1, \\ A[n-1] + S(A, n-1) & \text{se } n > 1. \end{cases}$$

In questa formulazione, la **fase di divisione** consiste nel passare da un array di $n$ elementi a uno di $n-1$ elementi, mentre la **fase di riunificazione** consiste nel sommare il valore corrente $A[n-1]$ al risultato ottenuto ricorsivamente dalla parte restante.

```
int somma(int A[], int n) {
    if (n == 1)
        return A[0];                    // caso base
    else
        return A[n-1] + somma(A, n-1); // divisione e
           riunificazione
}
```

L'albero di ricorsione per un array di 4 elementi $(a_0, a_1, a_2, a_3)$ ha la seguente forma:



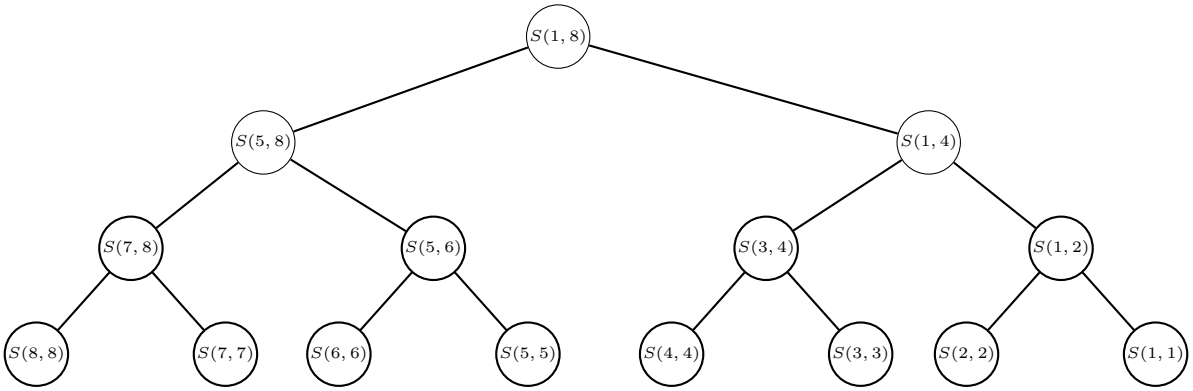

La profondità dell'albero è $n$, e ogni chiamata ricorsiva genera una sola nuova chiamata, per cui la complessità temporale è $O(n)$. Questo tipo di ricorsione, in cui l'ultima operazione della funzione è la chiamata ricorsiva stessa, viene detta **ricorsione di coda** (*tail recursion*). Molti compilatori sono in grado di ottimizzarla automaticamente, riducendo l'uso dello stack e rendendola equivalente a un ciclo iterativo.

Un'alternativa più strutturata consiste nel dividere l'array in due metà, calcolare ricorsivamente la somma di ciascuna parte e poi combinarle. In questo caso, la **fase di divisione** suddivide il problema in due sottoproblemi di dimensione $n/2$, e la **fase di riunificazione** consiste nel sommare i due risultati parziali:

$$S(A,n) = \begin{cases} A[0] & \text{se } n = 1, \\ S(A_{\text{sx}}, n/2) + S(A_{\text{dx}}, n/2) & \text{se } n > 1. \end{cases}$$

```
int sommaDivisa(int A[], int inizio, int fine) {
    if (inizio == fine)
        return A[inizio]; //caso base
    int medio = (inizio + fine) / 2;
    int sinistra = sommaDivisa(A, inizio, medio); //divisione 1
    int destra   = sommaDivisa(A, medio + 1, fine); //divisione 2
    return sinistra + destra; //riunificazione
}
```

L'albero di ricorsione per un array di 8 elementi ha la seguente struttura binaria:



In questo caso, l'albero è bilanciato e completo, con profondità $\log_2 n$ e $n$ nodi foglia. La complessità temporale complessiva soddisfa l'equazione di ricorrenza:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(1),$$

## 3.2 Verificare se la somma supera una soglia

Nel problema ora considerato non vogliamo calcolare il valore della somma, ma soltanto stabilire se la somma degli elementi di un array $A$ di lunghezza $n$ superi una certa soglia $T$. La natura del risultato cambia: non ci serve un numero, bensì un esito logico (`true`/`false`). Questa differenza, apparentemente minima, ha ricadute importanti sia sulla struttura della funzione sia sulla possibilità di interrompere prima la computazione.

Un modo sobrio di pensare la ricorsione consiste nell'evitare parametri superflui. Invece di passare una *somma parziale* esplicita, possiamo far "scorrere" la soglia $T$ verso il basso man mano che consumiamo l'array: a ogni passo sottraiamo l'ultimo elemento e chiediamo se la somma del prefisso rimanente supera la nuova soglia. Se, durante il processo, la soglia "residua" diventa negativa, significa che la somma accumulata ha già superato $T$ e possiamo fermarci.

```
bool supera(int A[], int n, int T) {
    if (T < 0)   return true;   // soglia gia' superata
    if (n == 0)  return false;  // array esaurito, soglia non
       superata
    return supera(A, n - 1, T - A[n - 1]);
}
```

Questa definizione è *essenziale*: i soli parametri necessari sono l'array, la dimensione corrente e la soglia. L'interruzione anticipata emerge naturalmente dal test $T < 0$, senza dover mantenere uno stato ausiliario (nessuna somma parziale esplicita), e i nomi brevi migliorano la resa tipografica in codice e figure.

È naturale chiedersi se un approccio a *doppia ricorsione* (divide et impera su due metà dell'array) possa offrire vantaggi. In questo problema, però, la risposta è per lo più negativa. Se si suddivide l'array in due sottoarray, ciascuna chiamata deve fornire non solo un esito booleano, ma anche l'informazione quantitativa (la somma della propria metà) per permettere al livello superiore di verificare se la somma totale supera $T$. Senza tale informazione, il livello superiore non potrebbe decidere: conoscere che "la sinistra da sola non supera $T$" non dice nulla sull'esito complessivo, che dipende da *sinistra + destra*. Ne consegue che la ricorsione binaria, in forma puramente booleana, o è *insufficiente* (manca la somma per combinare) o deve tornare a propagare anche la somma, complicando l'interfaccia e riducendo le opportunità di arresto anticipato.

```
// Variante "divide et impera": serve restituire ANCHE la somma.
struct Esito { bool over; int sum; };

Esito chk(int A[], int l, int r, int T) {
    if (l == r) {
        int s = A[l];
        return { s > T, s };
```

```
    }
    int m = (l + r) / 2;
    Esito L = chk(A, l,   m, T);
    if (L.over) return { true, L.sum };  // short-circuit locale
    Esito R = chk(A, m+1, r, T);
    int tot = L.sum + R.sum;
    return { tot > T, tot };
}
```

Questa soluzione introduce una struttura di ritorno più pesante (`sum` oltre a `over`) e, nel caso in cui la parte sinistra da sola non superi la soglia, *costringe* a calcolare anche la parte destra prima di poter decidere. In altre parole, l'interruzione anticipata non è globale: può scattare solo se una delle due metà supera di per sé la soglia; in caso contrario, la ricorsione deve esplorare entrambe, perdendo quel vantaggio "passo-passo" che la versione lineare sfrutta con il test $T < 0$.

Dal punto di vista del costo, la ricorsione essenziale `supera` visita al più $n$ elementi e si arresta non appena la soglia è superata; nel peggiore dei casi è $O(n)$, ma spesso termina molto prima. La variante a doppia ricorsione costruisce un albero completo di chiamate (salvo rari short-circuit su sottoalberi), con overhead di combinazione e passaggio di strutture; nel peggiore dei casi, visita comunque tutti gli elementi, senza un reale beneficio rispetto alla forma lineare. In sintesi: quando l'obiettivo è un *esito booleano* con *possibile arresto anticipato*, la ricorsione lineare con soglia residua è la forma più naturale, chiara ed efficace.

# 4 Approccio ricorsivo e approccio iterativo

Dopo aver analizzato vari esempi di funzioni ricorsive e la loro costruzione passo dopo passo, è naturale chiedersi quale sia la differenza sostanziale tra un approccio *ricorsivo* e uno *iterativo*, quando entrambi mirano a risolvere lo stesso problema. A prima vista, le due strategie sembrano alternative equivalenti: in fondo, qualsiasi algoritmo iterativo può essere riscritto in forma ricorsiva, e viceversa. Tuttavia, tra le due modalità di pensiero esistono differenze profonde, sia sul piano concettuale che su quello pratico.

Dal punto di vista **concettuale**, la ricorsione rappresenta un modo di ragionare "dall'alto verso il basso". Essa parte dalla formulazione del problema generale e lo scompone in sottoproblemi più piccoli e più semplici, fino a raggiungere un caso base di immediata risoluzione. Ogni chiamata ricorsiva non è altro che una delega: il problema corrente affida a una versione più semplice di sé il compito di procedere, assumendo che quella saprà risolversi correttamente. In questo senso, la ricorsione non è solo una tecnica di programmazione, ma un vero e proprio *paradigma cognitivo*: affrontare la complessità riducendola progressivamente, e ricomporla soltanto alla fine.

L'iterazione, al contrario, procede "dal basso verso l'alto". Essa non delega, ma *accumula*. La soluzione viene costruita passo dopo passo, mantenendo esplicitamente lo stato intermedio della computazione — per esempio attraverso variabili di controllo, contatori o accumulatori. Nel caso della somma di un array, l'iterazione scorre gli elementi e aggiorna una variabile `somma`; nel caso del calcolo del fattoriale, moltiplica progressivamente i numeri da 1 a $n$. La ricorsione, invece, si affida implicitamente allo *stack delle chiamate*

per ricordare i passi precedenti: ogni livello della funzione sospende la propria esecuzione fino a quando le chiamate più profonde non restituiscono il risultato.

Dal punto di vista **computazionale**, le differenze diventano più tangibili. Un algoritmo iterativo, in generale, utilizza una quantità di memoria costante $O(1)$, poiché le variabili che descrivono lo stato sono riutilizzate a ogni passo. Un algoritmo ricorsivo, invece, richiede una quantità di memoria proporzionale alla profondità della ricorsione, tipicamente $O(n)$ per problemi lineari. Ogni chiamata comporta la creazione di un nuovo contesto di esecuzione, con copie locali di variabili e indirizzi di ritorno: questi vengono mantenuti nello *stack* fino al completamento della computazione. Per questo motivo, le versioni iterative sono spesso più efficienti in termini di spazio, e in alcuni casi anche in termini di tempo, poiché evitano il costo di molteplici invocazioni di funzione.

Tuttavia, la ricorsione ha dalla sua parte un vantaggio altrettanto significativo: la **chiarezza espressiva**. In molti casi, la formulazione ricorsiva riflette direttamente la definizione matematica del problema. La funzione che calcola il fattoriale, ad esempio, rispecchia la stessa relazione $n! = n \cdot (n-1)!$ che la definisce; la funzione che somma un array ne segue la struttura lineare o binaria; e molti algoritmi su strutture dati come alberi, grafi o liste assumono una forma ricorsiva del tutto naturale. Scrivere un algoritmo ricorsivo significa spesso *pensare il problema nella sua essenza*, isolandone la struttura logica più profonda, anche a costo di sacrificare qualche ottimizzazione.

Un altro vantaggio, meno evidente ma altrettanto importante, riguarda la **modularità** e la **componibilità** delle soluzioni. Le funzioni ricorsive tendono a essere più brevi, più leggibili e più facilmente generalizzabili. Molti algoritmi avanzati — come il Merge Sort, la ricerca binaria, il Quick Sort o le procedure su alberi binari — derivano direttamente da una formulazione ricorsiva che ne chiarisce il principio di funzionamento. La possibilità di arresto anticipato, come abbiamo visto nel caso del controllo di soglia, rappresenta un ulteriore esempio di flessibilità: la ricorsione può modellare in modo elegante anche processi che si interrompono in condizioni variabili, cosa che spesso richiede maggiore attenzione in un ciclo iterativo.

Naturalmente, la ricorsione presenta anche dei **limiti pratici**. Quando la profondità della chiamata è molto grande (come nel caso di dati di dimensione elevata o di ricorsioni mal definite), si rischia il superamento dello spazio disponibile nello stack, con conseguente *stack overflow*. Alcuni linguaggi di programmazione ottimizzano le *ricorsioni di coda*, eliminando la necessità di mantenere più livelli attivi, ma non tutti i casi si prestano a tale ottimizzazione. Inoltre, la gestione implicita dello stato e dei contesti di esecuzione può rendere più difficile il debug o il tracciamento manuale del programma, specialmente per studenti alle prime armi.

In sintesi, possiamo dire che la ricorsione e l'iterazione rappresentano due prospettive complementari su uno stesso problema. L'approccio iterativo privilegia l'efficienza e la concretezza del controllo esplicito, mentre la ricorsione privilegia l'astrazione, la chiarezza concettuale e la naturale corrispondenza con la definizione logica del problema. Scegliere tra i due significa decidere se valorizzare la *trasparenza della logica* o la *parsimonia dell'esecuzione*. In molti casi, il miglior compromesso consiste nel partire da una definizione ricorsiva — più vicina all'intuizione e più facile da verificare — e successivamente trasformarla in una versione iterativa ottimizzata, preservando la chiarezza del ragionamento originario.

In ultima analisi, la ricorsione non è solo un modo per scrivere codice, ma un modo per *pensare* i problemi computazionali: un linguaggio concettuale che, quando ben compreso, permette di vedere la struttura nascosta dietro la complessità apparente. L'iterazione, dal canto suo, rappresenta la traduzione di quella struttura in un meccanismo efficiente e diretto. Nell'equilibrio tra i due approcci si trova una delle lezioni più profonde della programmazione: la tensione costante tra eleganza e concretezza, tra astrazione e controllo.

# 5 Esercizi

## Esercizi numerici di ricorsione

1. **Somma dei primi numeri dispari.** Scrivere una funzione ricorsiva che calcoli la somma dei primi $n$ numeri dispari positivi.
   *Suggerimento (da considerare solo in caso di difficoltà):* l'ultimo termine può essere espresso come $2n - 1$.

2. **Conteggio delle cifre di un numero.** Scrivere una funzione ricorsiva che, dato un numero intero positivo $n$, restituisca il numero di cifre che lo compongono.
   *Suggerimento (da considerare solo in caso di difficoltà):* dividere $n$ per 10 a ogni passo riduce di una cifra il numero.

3. **Somma delle cifre di un numero.** Scrivere una funzione ricorsiva che calcoli la somma delle cifre di un numero intero positivo $n$.
   *Suggerimento (da considerare solo in caso di difficoltà):* l'ultima cifra di $n$ può essere ottenuta con $n \bmod 10$.

4. **Massimo comune divisore (MCD).** Scrivere una funzione ricorsiva che calcoli il massimo comune divisore tra due numeri interi positivi $a$ e $b$.
   *Suggerimento (da considerare solo in caso di difficoltà):* utilizzare la relazione $\text{MCD}(a, b) = \text{MCD}(b, a \bmod b)$.

5. **Conteggio dei divisori di un numero.** Scrivere una funzione ricorsiva che conti quanti interi positivi dividono esattamente un dato numero $n$.
   *Suggerimento (da considerare solo in caso di difficoltà):* verificare se $n \bmod k = 0$ per un divisore candidato $k$ decrescente.

6. **Prodotto dei primi numeri dispari.** Scrivere una funzione ricorsiva che calcoli il prodotto dei primi $n$ numeri dispari positivi.
   *Suggerimento (da considerare solo in caso di difficoltà):* l'ultimo termine della sequenza è $2n - 1$.

7. **Successione di Fibonacci.** Scrivere una funzione ricorsiva che calcoli il termine $n$-esimo della successione di Fibonacci.
   *Suggerimento (da considerare solo in caso di difficoltà):* la successione è definita da $F(0) = 0$, $F(1) = 1$ e $F(n) = F(n-1) + F(n-2)$.

8. **Somma alternata dei primi $n$ interi positivi.** Scrivere una funzione ricorsiva che calcoli la somma alternata dei primi $n$ interi positivi:

$$S(n) = 1 - 2 + 3 - 4 + 5 - 6 + \cdots \pm n.$$

   *Suggerimento (da considerare solo in caso di difficoltà):* la regola può essere espressa come $S(n) = S(n-1) + (-1)^{n+1} \cdot n$.

## Esercizi di ricorsione su array

1. **Conteggio dei valori negativi.** Scrivere una funzione ricorsiva che conti quanti elementi di un array $A$ di lunghezza $n$ sono minori di zero.
   *Suggerimento (da considerare solo in caso di difficoltà):* sommare 1 se $A[i] < 0$ e procedere con l'array rimanente.

2. **Prodotto dei valori pari.** Scrivere una funzione ricorsiva che calcoli il prodotto di tutti gli elementi pari di un array $A$ di lunghezza $n$. Se l'array non contiene elementi pari, la funzione deve restituire 1.
   *Suggerimento (da considerare solo in caso di difficoltà):* verificare la condizione $A[i] \bmod 2 = 0$ prima di moltiplicare.

3. **Conteggio dei valori superiori alla media.** Scrivere una funzione ricorsiva che conti quanti elementi di un array $A$ di lunghezza $n$ hanno valore superiore alla media aritmetica dell'array stesso.
   *Suggerimento (da considerare solo in caso di difficoltà):* calcolare prima la media con una funzione separata e utilizzarla come parametro nella chiamata ricorsiva principale.

4. **Differenza tra somma dei pari e somma dei dispari.** Scrivere una funzione ricorsiva che calcoli la differenza tra la somma degli elementi pari e la somma degli elementi dispari di un array $A$ di lunghezza $n$.
   *Suggerimento (da considerare solo in caso di difficoltà):* usare una ricorsione unica in cui ogni elemento contribuisce con segno positivo o negativo a seconda della sua parità.

5. **Verifica di simmetria.** Scrivere una funzione ricorsiva che determini se un array $A$ di lunghezza $n$ è simmetrico rispetto al suo centro, cioè se $A[i] = A[n-1-i]$ per ogni $i$. La funzione deve restituire `true` o `false`.
   *Suggerimento (da considerare solo in caso di difficoltà):* confrontare gli elementi alle estremità e ridurre il problema escludendo la prima e l'ultima posizione.

6. **Conteggio dei picchi locali.** Scrivere una funzione ricorsiva che conti quanti elementi di un array $A$ sono *picchi locali*, ossia valori tali che $A[i] > A[i-1]$ e $A[i] > A[i+1]$.
   *Suggerimento (da considerare solo in caso di difficoltà):* limitare la ricorsione agli indici interni dell'array e verificare la condizione di massimo relativo.

7. **Calcolo del valore medio ricorsivo.** Scrivere una funzione ricorsiva che calcoli la media aritmetica degli elementi di un array $A$ di lunghezza $n$.
   *Suggerimento (da considerare solo in caso di difficoltà):* restituire la media come combinazione della somma parziale e del numero di elementi elaborati.

8. **Verifica della presenza di una sottosequenza crescente.** Scrivere una funzione ricorsiva che verifichi se all'interno di un array $A$ esiste una sequenza di almeno tre elementi consecutivi in ordine strettamente crescente. La funzione deve restituire `true` o `false`.
   *Suggerimento (da considerare solo in caso di difficoltà):* mantenere un contatore che si incrementa finché la sequenza è crescente e si azzera in caso contrario.

9. **Conteggio dei valori univoci.** Scrivere una funzione ricorsiva che conti quanti elementi di un array $A$ non si ripetono. Un elemento è detto univoco se compare una sola volta nell'array.
   *Suggerimento (da considerare solo in caso di difficoltà):* confrontare ogni elemento con i restanti e proseguire riducendo l'array.

10. **Prodotto alternato degli elementi.** Scrivere una funzione ricorsiva che calcoli il prodotto alternato degli elementi di un array $A$, moltiplicando il primo, dividendo per il secondo, moltiplicando per il terzo, e così via.
    *Suggerimento (da considerare solo in caso di difficoltà):* applicare a ciascun elemento un segno alternato $(-1)^i$ nel calcolo logaritmico o equivalente.

## Esercizi di ricorsione su stringhe

1. **Conteggio dei caratteri.** Scrivere una funzione ricorsiva che, data una stringa $S$, restituisca il numero totale dei suoi caratteri.
   *Suggerimento (da considerare solo in caso di difficoltà):* ridurre la lunghezza della stringa di un carattere a ogni passo e incrementare un contatore.

2. **Conteggio di una lettera specifica.** Scrivere una funzione ricorsiva che conti quante volte un carattere $c$ compare all'interno di una stringa $S$.
   *Suggerimento (da considerare solo in caso di difficoltà):* confrontare il primo carattere di $S$ con $c$ e proseguire sul resto della stringa.

3. **Verifica di palindromo.** Scrivere una funzione ricorsiva che determini se una stringa $S$ è palindroma, cioè se può leggersi allo stesso modo da sinistra a destra e da destra a sinistra. La funzione deve restituire `true` o `false`.
   *Suggerimento (da considerare solo in caso di difficoltà):* confrontare i caratteri alle estremità e ridurre la stringa escludendo il primo e l'ultimo carattere.

4. **Rimozione di un carattere.** Scrivere una funzione ricorsiva che elimini tutte le occorrenze di un carattere $c$ da una stringa $S$ e restituisca la nuova stringa risultante.
   *Suggerimento (da considerare solo in caso di difficoltà):* costruire la nuova stringa aggiungendo i caratteri diversi da $c$ e proseguire sulla parte rimanente.

5. **Conteggio delle vocali.** Scrivere una funzione ricorsiva che conti quante vocali $(a, e, i, o, u)$ contiene una stringa $S$.
   *Suggerimento (da considerare solo in caso di difficoltà):* controllare se il primo carattere appartiene all'insieme delle vocali e proseguire sul resto della stringa.

6. **Inversione di una stringa.** Scrivere una funzione ricorsiva che restituisca la stringa $S$ invertita.
   *Suggerimento (da considerare solo in caso di difficoltà):* spostare il primo carattere in fondo alla stringa e proseguire sull'insieme dei restanti.

7. **Conteggio delle maiuscole.** Scrivere una funzione ricorsiva che conti quanti caratteri maiuscoli contiene una stringa $S$.
   *Suggerimento (da considerare solo in caso di difficoltà):* utilizzare il confronto con gli intervalli di codici ASCII o funzioni di libreria equivalenti.

8. **Verifica della presenza di una sottostringa.** Scrivere una funzione ricorsiva che determini se una stringa $T$ è contenuta all'interno di una stringa $S$. La funzione deve restituire `true` o `false`.
   *Suggerimento (da considerare solo in caso di difficoltà):* confrontare progressivamente il prefisso di $S$ con $T$ e scorrere ricorsivamente di un carattere.

9. **Rimozione delle cifre numeriche.** Scrivere una funzione ricorsiva che elimini da una stringa $S$ tutti i caratteri numerici (0–9). La funzione deve restituire la stringa risultante.
   *Suggerimento (da considerare solo in caso di difficoltà):* verificare a ogni passo se il primo carattere è una cifra numerica e, in caso contrario, mantenerlo nella costruzione della nuova stringa.

10. **Conteggio delle parole.** Scrivere una funzione ricorsiva che calcoli quante parole contiene una stringa $S$, assumendo che le parole siano separate da uno spazio singolo.
    *Suggerimento (da considerare solo in caso di difficoltà):* incrementare il contatore quando si incontra una transizione da spazio a carattere non vuoto.

## Esercizi di ricorsione su BST

1. **Conteggio dei nodi.** Scrivere una funzione ricorsiva che restituisca il numero totale di nodi presenti in un albero binario di ricerca.
   *Suggerimento (da considerare solo in caso di difficoltà):* la dimensione di un albero è pari a 1 più la somma delle dimensioni dei due sottoalberi.

2. **Somma dei valori memorizzati.** Scrivere una funzione ricorsiva che calcoli la somma di tutti i valori numerici contenuti nei nodi di un BST.
   *Suggerimento (da considerare solo in caso di difficoltà):* la somma complessiva si ottiene sommando il valore del nodo corrente con le somme dei due sottoalberi.

3. **Ricerca di un valore.** Scrivere una funzione ricorsiva che determini se un certo valore $x$ è presente nel BST. La funzione deve restituire `true` o `false`.
   *Suggerimento (da considerare solo in caso di difficoltà):* confrontare $x$ con il valore del nodo corrente e proseguire a sinistra o a destra in base al risultato del confronto.

4. **Calcolo dell'altezza.** Scrivere una funzione ricorsiva che calcoli l'altezza di un BST, cioè la lunghezza del cammino più lungo dalla radice a una foglia.
   *Suggerimento (da considerare solo in caso di difficoltà):* l'altezza è pari a 1 più il massimo tra le altezze dei due sottoalberi.

5. **Conteggio delle foglie.** Scrivere una funzione ricorsiva che determini quanti nodi foglia (senza figli) contiene il BST.
   *Suggerimento (da considerare solo in caso di difficoltà):* un nodo è foglia se entrambi i sottoalberi sono nulli.

6. **Verifica di BST valido.** Scrivere una funzione ricorsiva che verifichi se un albero binario generico rispetta le proprietà di un BST. La funzione deve restituire `true` se per ogni nodo $v$ tutti i valori nel sottoalbero sinistro sono minori di $v$, e tutti quelli nel sottoalbero destro sono maggiori.
   *Suggerimento (da considerare solo in caso di difficoltà):* mantenere intervalli di validità $(\min, \max)$ aggiornati a ogni chiamata.

7. **Ricerca del valore minimo.** Scrivere una funzione ricorsiva che restituisca il valore minimo memorizzato nel BST.
   *Suggerimento (da considerare solo in caso di difficoltà):* in un BST il minimo si trova nel nodo più a sinistra.

8. **Ricerca del valore massimo.** Scrivere una funzione ricorsiva che restituisca il valore massimo memorizzato nel BST.
   *Suggerimento (da considerare solo in caso di difficoltà):* in un BST il massimo si trova nel nodo più a destra.

9. **Conteggio dei nodi con un solo figlio.** Scrivere una funzione ricorsiva che conti quanti nodi del BST hanno esattamente un figlio non nullo.
   *Suggerimento (da considerare solo in caso di difficoltà):* un nodo ha un solo figlio se uno dei due sottoalberi è nullo e l'altro no.

10. **Verifica della presenza di un percorso somma.** Scrivere una funzione ricorsiva che verifichi se esiste un cammino dalla radice a una foglia la cui somma dei valori dei nodi sia pari a un valore dato $k$. La funzione deve restituire `true` o `false`.
    *Suggerimento (da considerare solo in caso di difficoltà):* a ogni passo sottrarre dal valore $k$ il valore del nodo corrente e verificare la condizione al raggiungimento di una foglia.

11. **Somma dei nodi di un livello specifico.** Scrivere una funzione ricorsiva che calcoli la somma dei valori dei nodi presenti a un livello $L$ specificato. Il livello della radice è 0.
    *Suggerimento (da considerare solo in caso di difficoltà):* decrementare $L$ a ogni passo fino a raggiungere il livello richiesto.

12. **Verifica di struttura identica.** Scrivere una funzione ricorsiva che determini se due BST hanno la stessa struttura (indipendentemente dai valori contenuti). La funzione deve restituire `true` se i due alberi hanno nodi disposti nello stesso modo.
    *Suggerimento (da considerare solo in caso di difficoltà):* confrontare simultaneamente i due alberi visitando in parallelo i nodi corrispondenti.

13. **Somma dei valori compresi in un intervallo.** Scrivere una funzione ricorsiva che calcoli la somma dei valori dei nodi di un BST compresi tra due estremi $a$ e $b$.
    *Suggerimento (da considerare solo in caso di difficoltà):* sfruttare le proprietà del BST per evitare di visitare rami non necessari.

14. **Verifica di bilanciamento.** Scrivere una funzione ricorsiva che verifichi se un BST è bilanciato, ossia se per ogni nodo la differenza tra le altezze dei due sottoalberi non supera 1. La funzione deve restituire `true`/`false`.
    *Suggerimento (da considerare solo in caso di difficoltà):* calcolare l'altezza dei sottoalberi durante la discesa e interrompere anticipatamente in caso di squilibrio.

15. **Copia del BST.** Scrivere una funzione ricorsiva che costruisca e restituisca una copia indipendente di un BST dato.
    *Suggerimento (da considerare solo in caso di difficoltà):* creare un nuovo nodo con lo stesso valore del nodo corrente e ricopiare ricorsivamente i due sottoalberi.

## Esercizi di ricorsione avanzati

1. **Conteggio delle inversioni in un array.** Dato un array $A$ di lunghezza $n$, calcolare ricorsivamente il numero di coppie $(i, j)$ con $i < j$ e $A[i] > A[j]$.
   *Suggerimento (da considerare solo in caso di difficoltà):* integrare il conteggio nella fase di unione di un divide-et-impera stile merge.

2. **Selezione dell'$k$-esimo elemento (Quickselect ricorsivo).** Dato un array $A$ e un indice $k$ ($0 \leq k < n$), progettare una procedura ricorsiva che restituisca l'elemento di ordine $k$ secondo l'ordinamento crescente.
   *Suggerimento (da considerare solo in caso di difficoltà):* partizionare rispetto a un pivot e ricorrere solo nel sottoarray che contiene l'indice $k$.

3. **Partizionamento palindromico minimo di una stringa.** Data una stringa $S$, determinare ricorsivamente il minimo numero di tagli per suddividerla in sottostringhe palindrome.
   *Suggerimento (da considerare solo in caso di difficoltà):* usare due indici per i confini e memorizzare sottoproblemi ripetuti.

4. **Corrispondenza con espressioni regolari semplificate.** Data una stringa $S$ e un pattern $P$ che può contenere i simboli speciali . (un carattere qualsiasi) e $*$ (zero o più ripetizioni del precedente), stabilire ricorsivamente se $P$ corrisponde interamente a $S$.
   *Suggerimento (da considerare solo in caso di difficoltà):* implementare i casi $*$ consumando $0, 1, \ldots$ caratteri compatibili e sfruttare pruning quando il match è impossibile.

5. **Sottosequenza palindroma massima (LPS).** Data una stringa $S$, calcolare ricorsivamente la lunghezza della più lunga sottosequenza palindroma.
   *Suggerimento (da considerare solo in caso di difficoltà):* due indici $i, j$ con ricorrenza che confronta $S[i]$ e $S[j]$ e memorizza i risultati.

6. **Numero di partizioni intere.** Per un intero $n \geq 1$, calcolare ricorsivamente quante partizioni di $n$ esistono in parti non crescenti (ordine delle parti irrilevante).
   *Suggerimento (da considerare solo in caso di difficoltà):* usare stato $(n, m)$ con $m$ massimo addendo consentito, riducendo $n$ e/o $m$.

7. **Conteggio di sottoinsiemi con somma esatta.** Dato un array di interi $A$ e un valore $T$, determinare ricorsivamente quanti sottoinsiemi di $A$ sommano esattamente a $T$.
   *Suggerimento (da considerare solo in caso di difficoltà):* scelte include/esclude sull'elemento corrente; pruning quando $T$ diventa impossibile.

8. **Conteggio di cammini con somma esatta in un BST.** Dato un BST con valori interi e un intero $K$, contare ricorsivamente i cammini dalla radice alle foglie la cui somma è esattamente $K$.
   *Suggerimento (da considerare solo in caso di difficoltà):* sottrarre il valore del nodo corrente da $K$ e sommare i conteggi dei sottoalberi.

9. **Antenato comune più basso (LCA) in un BST.** Dati un BST e due chiavi distinte $x, y$ presenti nell'albero, progettare una procedura ricorsiva che restituisca il loro LCA.

*Suggerimento (da considerare solo in caso di difficoltà):* usare il confronto con il valore del nodo corrente per scegliere sinistra/destra o fermarsi.

10. **Equivalenza con ribaltamento di sottoalberi (flip-equivalence).** Date le radici di due BST, determinare ricorsivamente se sono equivalenti a meno di scambi tra figli sinistro e destro in nodi arbitrari.
*Suggerimento (da considerare solo in caso di difficoltà):* confrontare ricorsivamente sia la coppia ($L \leftrightarrow L$, $R \leftrightarrow R$) sia ($L \leftrightarrow R$, $R \leftrightarrow L$).

11. **Ricostruzione di un BST dal preorder.** Data una sequenza di visite in preorder di un BST con chiavi distinte, ricostruire ricorsivamente l'albero.
*Suggerimento (da considerare solo in caso di difficoltà):* passare limiti $(\min, \max)$ e un indice globale/di riferimento per consumare i nodi validi.

12. **Conteggio delle coppie con differenza limitata.** Dato un array $A$ di lunghezza $n$ e un intero $D \geq 0$, contare ricorsivamente quante coppie $(i, j)$ con $i < j$ soddisfano $|A[i] - A[j]| \leq D$.
*Suggerimento (da considerare solo in caso di difficoltà):* schema divide-et-impera con unione che conta coppie inter-partizione in modo ordinato.

13. **Minimo numero di rimozioni per rendere una stringa palindroma.** Data una stringa $S$, calcolare ricorsivamente il minimo numero di caratteri da rimuovere per ottenere un palindromo.
*Suggerimento (da considerare solo in caso di difficoltà):* ricorrenza su due indici $i, j$; collegare alla LPS per derivare il risultato.

14. **Partizioni con limiti di cardinalità.** Dato un intero $n$ e un limite $k$, contare ricorsivamente in quanti modi $n$ può essere scritto come somma di *al più* $k$ interi positivi non crescenti.
*Suggerimento (da considerare solo in caso di difficoltà):* estendere lo stato a $(n, m, k)$ con massimo addendo $m$ e parti residue $k$.