



SAPIENZA
UNIVERSITÀ DI ROMA

Algoritmi

Volume 1 — Algoritmi di base

La rappresentazione dei numeri e lo pseudocodice, il linguaggio in cui sono scritti tutti gli algoritmi del corso; i primi algoritmi: ordinamento (insertion, exchange e selection sort), ricerca lineare e binaria, algoritmi numerici (cambio di base, test di primalità, crivello di Eratostene, massimo comune divisore). Ogni algoritmo con lo pseudocodice ed esempi eseguiti passo per passo.

Fabio Furini

fabiofurini.github.io

Dipartimento di Ingegneria informatica, automatica e gestionale — Sapienza Università di Roma

Versione online e materiale scaricabile:

fabiofurini.github.io/algoritmi/
github.com/fabiofurini/algoritmi

Anno accademico 2026–2027

Indice

1	Rappresentazione dei numeri	3
	Introduzione	3
1.1	Rappresentazione dei numeri naturali	4
1.2	Rappresentazione dei numeri interi	10
1.3	Rappresentazione dei numeri reali	19
2	Pseudocodice	24
	Introduzione	24
2.1	Introduzione allo pseudocodice	25
2.2	Variabili e assegnamenti	26
2.3	Operazioni di base	28
2.4	Costrutti di base	29
2.5	Array	38
2.6	Funzioni	41
2.7	Insiemi	47
2.8	Convenzioni di stile	49
3	Algoritmi di ordinamento	50
	Introduzione	50
3.1	Il problema dell'ordinamento	50
3.2	Insertion Sort	53
3.3	Exchange Sort	60
3.4	Selection Sort	65
3.5	Confronto dei tre algoritmi	70
4	Ricerca lineare e binaria	72
4.1	Il problema della ricerca	72
4.2	Ricerca lineare	73
4.3	Ricerca binaria	76
5	Algoritmi numerici	83
	Introduzione	83
5.1	Cambio di base	83



5.2	Test di primalità	89
5.3	Crivello di Eratostene	94
5.4	Massimo comun divisore	99



Rappresentazione dei numeri

Introduzione

Un algoritmo manipola dati e, prima di poter descrivere come funziona un algoritmo, dobbiamo stabilire come questi dati vengono scritti. Per i numeri questa domanda ha una risposta precisa: un calcolatore memorizza ogni numero come una sequenza finita di simboli, e con un dato numero di simboli si possono scrivere soltanto un numero finito di numeri. Questo semplice fatto ha conseguenze importanti: ogni rappresentazione ha un **intervallo di rappresentazione**, al di fuori del quale i numeri non possono essere memorizzati, e la rappresentazione dei numeri reali è, in generale, solo approssimata.

In questo capitolo studiamo tre rappresentazioni:

- la rappresentazione dei **numeri naturali** in un sistema di numerazione posizionale in base b , in particolare il sistema binario usato dai calcolatori;
- la rappresentazione dei **numeri interi**, prima in modulo e segno e poi in complemento a due, che è la rappresentazione usata dai calcolatori moderni;
- la rappresentazione dei **numeri reali** in virgola mobile, secondo lo standard IEEE 754.

Per ogni rappresentazione calcoliamo il numero più grande e quello più piccolo che si possono rappresentare con un dato numero di cifre, e il numero minimo di cifre necessarie per rappresentare un dato numero.

1.1 Rappresentazione dei numeri naturali

1.1.1 Sistemi di numerazione posizionali

Siamo abituati a scrivere i numeri nel sistema decimale, dove la stessa cifra ha un valore diverso a seconda della posizione in cui compare: in 555 le tre cifre sono uguali, ma valgono 500, 50 e 5. Questa è l'idea di sistema di numerazione posizionale.

Definizione 1.1: sistema di numerazione posizionale

Un **sistema di numerazione posizionale** è un modo di codificare i numeri naturali mediante una **sequenza di cifre**, in cui ogni cifra (singolo simbolo) contribuisce in modo diverso a seconda della sua posizione.

Definizione 1.2: base di un sistema di numerazione posizionale

Il numero di cifre diverse di un sistema di numerazione posizionale si chiama **base b** del sistema.

- La base b di un sistema di numerazione posizionale deve essere maggiore o uguale a due.
- Il sistema unario non è un sistema di numerazione posizionale: i numeri naturali si rappresentano ripetendo la cifra 1 tante volte quanto necessario (ad esempio, 3 si rappresenta come 111).
- Alcuni dei principali sistemi di numerazione posizionali sono:

Sistema di numerazione posizionale	base b	cifre diverse
Sistema binario	2	0, 1
Sistema ottale	8	0, 1, 2, 3, 4, 5, 6, 7
Sistema decimale	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Sistema esadecimale	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

- Nel sistema esadecimale le sei lettere A, B, C, D, E, F sono cifre, con i valori 10, 11, 12, 13, 14, 15.

Definizione 1.3: sequenza di cifre

Una **sequenza di k cifre** che rappresenta un numero naturale in base b si indica con:

$$(d_{k-1} \ d_{k-2} \ \dots \ d_2 \ d_1 \ d_0)_b \quad (1.1)$$

dove le posizioni si contano da destra a sinistra, partendo da 0 e crescendo fino a $k-1$. La **cifra meno significativa** (LSD, *Least Significant Digit*) è d_0 . La **cifra più significativa** (MSD, *Most Significant Digit*) è d_{k-1} .

La cifra in posizione i viene moltiplicata per la potenza i -esima della base. Sommando tutti questi contributi si ottiene il numero rappresentato dalla sequenza.

Data una sequenza di k cifre in un sistema di numerazione posizionale in base b , il numero naturale che essa rappresenta in base dieci è dato dalla seguente sequenza di cifre:

$$\left(\sum_{i=0}^{k-1} d_i b^i \right)_{10} \quad (1.2)$$

Esempio 1.1: sequenze di cifre

Il numero cento scritto nei quattro sistemi di numerazione posizionali della tabella precedente. Il valore di ogni sequenza si calcola con la formula (1.2), posizione per posizione, partendo dalla cifra meno significativa d_0 .

Sequenza di cifre	b	k	rappresentazione decimale
$(100)_2$	2	3	$(0 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2)_{10} = (4)_{10}$
$(1100100)_2$	2	7	$(0 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 0 \cdot 2^4 + 1 \cdot 2^5 + 1 \cdot 2^6)_{10}$ $= (4 + 32 + 64)_{10} = (100)_{10}$
$(144)_8$	8	3	$(4 \cdot 8^0 + 4 \cdot 8^1 + 1 \cdot 8^2)_{10} = (4 + 32 + 64)_{10} = (100)_{10}$
$(100)_{10}$	10	3	$(0 \cdot 10^0 + 0 \cdot 10^1 + 1 \cdot 10^2)_{10} = (100)_{10}$
$(64)_{16}$	16	2	$(4 \cdot 16^0 + 6 \cdot 16^1)_{10} = (4 + 96)_{10} = (100)_{10}$

La stessa sequenza di cifre 100 rappresenta numeri diversi in basi diverse, e lo stesso numero richiede più cifre quando la base è più piccola.

Esempio 1.2: tutte le sequenze di quattro cifre binarie

Con $k = 4$ cifre binarie possiamo scrivere $2^4 = 16$ sequenze diverse, che rappresentano i numeri naturali da 0 a 15:

Sequenza di cifre	b	k	rappresentazione decimale
$(0000)_2$	2	4	$(0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0)_{10} = (0)_{10}$
$(0001)_2$	2	4	$(0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0)_{10} = (1)_{10}$
$(0010)_2$	2	4	$(0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0)_{10} = (2)_{10}$
$(0011)_2$	2	4	$(0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0)_{10} = (3)_{10}$
$(0100)_2$	2	4	$(0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0)_{10} = (4)_{10}$
$(0101)_2$	2	4	$(0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0)_{10} = (5)_{10}$
$(0110)_2$	2	4	$(0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0)_{10} = (6)_{10}$
$(0111)_2$	2	4	$(0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0)_{10} = (7)_{10}$
$(1000)_2$	2	4	$(1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0)_{10} = (8)_{10}$
$(1001)_2$	2	4	$(1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0)_{10} = (9)_{10}$
$(1010)_2$	2	4	$(1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0)_{10} = (10)_{10}$
$(1011)_2$	2	4	$(1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0)_{10} = (11)_{10}$
$(1100)_2$	2	4	$(1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0)_{10} = (12)_{10}$
$(1101)_2$	2	4	$(1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0)_{10} = (13)_{10}$
$(1110)_2$	2	4	$(1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0)_{10} = (14)_{10}$
$(1111)_2$	2	4	$(1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0)_{10} = (15)_{10}$

Di solito, per le sequenze di cifre in base dieci, le parentesi si omettono e, in assenza di parentesi, la base si assume uguale a dieci.

1.1.2 Intervalli di rappresentazione

La tabella dell'Esempio 1.2 si ferma a 15: con quattro cifre binarie non c'è modo di scrivere 16. In generale, un numero fissato k di cifre può rappresentare solo i numeri naturali fino a un certo limite, che calcoliamo ora.

Osservazione 1.4

Date k cifre in base b , il **più grande numero naturale** rappresentabile è:

$$b^k - 1 \quad (1.3)$$

e il relativo **intervallo di rappresentazione** per i numeri naturali è:

$$[0, b^k - 1] \quad (1.4)$$

Dimostrazione. Il più grande numero naturale in base b con k cifre è formato soltanto dalla cifra più grande $(b - 1)$ ripetuta k volte:

$$\left((b-1)_{k-1} \ (b-1)_{k-2} \ \dots \ (b-1)_1 \ (b-1)_0 \right)_b$$

Di conseguenza, usando la somma della progressione geometrica $\sum_{i=0}^{k-1} b^i = \frac{b^k - 1}{b - 1}$ (valida poiché $b \geq 2$), il più grande numero naturale in decimale è:

$$\sum_{i=0}^{k-1} (b-1) b^i = (b-1) \sum_{i=0}^{k-1} b^i = (b-1) \frac{b^k - 1}{b - 1} = b^k - 1$$

Il più piccolo numero naturale è 0, rappresentato da k cifre uguali a 0. Ogni numero naturale compreso tra 0 e $b^k - 1$ può essere rappresentato: le b^k sequenze diverse di k cifre rappresentano b^k numeri diversi, tutti nell'intervallo $[0, b^k - 1]$, che contiene esattamente b^k numeri naturali. $\square$

Le cifre del sistema binario si chiamano **bit** e 8 bit formano un **byte**.

Esempio 1.3: intervalli di rappresentazione in binario

Per $b = 2$ l'intervallo di rappresentazione con k bit è $[0, 2^k - 1]$. Le ultime righe corrispondono a 1, 2, 4 e 8 byte.

k	b	$b^k - 1$	intervallo di rappresentazione
1	2	$2^1 - 1 = 1$	$[0, 1]$
2	2	$2^2 - 1 = 3$	$[0, 3]$
3	2	$2^3 - 1 = 7$	$[0, 7]$
4	2	$2^4 - 1 = 15$	$[0, 15]$
8	2	$2^8 - 1 = 255$	$[0, 255]$
16	2	$2^{16} - 1 = 65,535$	$[0, 65,535]$
32	2	$2^{32} - 1 = 4,294,967,295$	$[0, 4,294,967,295]^1$
64	2	$2^{64} - 1$	$[0, 2^{64} - 1]^2$

¹ Intervallo di rappresentazione di **unsigned int** in C e C++ (4 byte).

² Intervallo di rappresentazione di **unsigned long long int** in C e C++ (8 byte); $2^{64} = 18,446,744,073,709,551,616$.

L'osservazione si può leggere anche nel verso opposto: dato un numero naturale n , quante cifre servono per scriverlo?

Osservazione 1.5

Il numero minimo di cifre per scrivere un numero naturale $n \geq 1$ in base b è:

$$\lfloor \log_b n \rfloor + 1 \quad (1.5)$$

Dimostrazione. Per l'Osservazione 1.4, il numero n si può scrivere con k cifre se e solo se:

$$n \leq b^k - 1 \quad \Longleftrightarrow \quad n < b^k \quad \Longleftrightarrow \quad \log_b n < k$$

dove la prima equivalenza vale perché n e b^k sono interi, e la seconda perché il logaritmo in base $b \geq 2$ è una funzione crescente. Il più piccolo intero k strettamente maggiore di $\log_b n$ è $\lfloor \log_b n \rfloor + 1$. Di conseguenza, $\lfloor \log_b n \rfloor + 1$ è il numero minimo di cifre per scrivere un numero naturale n in base b . $\square$

Esempio 1.4: numero minimo di cifre di cento

Calcoliamo il numero minimo di cifre di $n = 100$ in base 2 e in base 10.

- Passo 1 (base 2): poiché $2^6 = 64 \leq 100 < 128 = 2^7$, si ha $6 \leq \log_2 100 < 7$ (infatti $\log_2 100 \approx 6.644$), quindi:

$$\lfloor \log_2 100 \rfloor + 1 = 6 + 1 = 7$$

- Passo 2 (base 10): poiché $10^2 = 100$, si ha $\log_{10} 100 = 2$, quindi:

$$\lfloor \log_{10} 100 \rfloor + 1 = 2 + 1 = 3$$

Entrambi i risultati concordano con l'Esempio 1.1: $(1100100)_2$ ha 7 cifre e $(100)_{10}$ ha 3 cifre.

Esempio 1.5: numero minimo di cifre in binario

Per le potenze di due $n = 2^m$ si ha $\log_2 n = m$, quindi il numero minimo di bit è $m + 1$: una cifra 1 seguita da m cifre 0.

n	b	$\log_b n$	$\lfloor \log_b n \rfloor + 1$
$2 = 2^1$	2	$\log_2 2 = 1$	$\lfloor \log_2 2 \rfloor + 1 = 2$
$4 = 2^2$	2	$\log_2 4 = 2$	$\lfloor \log_2 4 \rfloor + 1 = 3$
$8 = 2^3$	2	$\log_2 8 = 3$	$\lfloor \log_2 8 \rfloor + 1 = 4$
$16 = 2^4$	2	$\log_2 16 = 4$	$\lfloor \log_2 16 \rfloor + 1 = 5$
$32 = 2^5$	2	$\log_2 32 = 5$	$\lfloor \log_2 32 \rfloor + 1 = 6$
$64 = 2^6$	2	$\log_2 64 = 6$	$\lfloor \log_2 64 \rfloor + 1 = 7$
$128 = 2^7$	2	$\log_2 128 = 7$	$\lfloor \log_2 128 \rfloor + 1 = 8$
$256 = 2^8$	2	$\log_2 256 = 8$	$\lfloor \log_2 256 \rfloor + 1 = 9$
$512 = 2^9$	2	$\log_2 512 = 9$	$\lfloor \log_2 512 \rfloor + 1 = 10$
$1,024 = 2^{10}$	2	$\log_2 1,024 = 10$	$\lfloor \log_2 1,024 \rfloor + 1 = 11$
$2,048 = 2^{11}$	2	$\log_2 2,048 = 11$	$\lfloor \log_2 2,048 \rfloor + 1 = 12$
$4,096 = 2^{12}$	2	$\log_2 4,096 = 12$	$\lfloor \log_2 4,096 \rfloor + 1 = 13$
$8,192 = 2^{13}$	2	$\log_2 8,192 = 13$	$\lfloor \log_2 8,192 \rfloor + 1 = 14$
$16,384 = 2^{14}$	2	$\log_2 16,384 = 14$	$\lfloor \log_2 16,384 \rfloor + 1 = 15$
$32,768 = 2^{15}$	2	$\log_2 32,768 = 15$	$\lfloor \log_2 32,768 \rfloor + 1 = 16$
$65,536 = 2^{16}$	2	$\log_2 65,536 = 16$	$\lfloor \log_2 65,536 \rfloor + 1 = 17$
$131,072 = 2^{17}$	2	$\log_2 131,072 = 17$	$\lfloor \log_2 131,072 \rfloor + 1 = 18$
$262,144 = 2^{18}$	2	$\log_2 262,144 = 18$	$\lfloor \log_2 262,144 \rfloor + 1 = 19$
$524,288 = 2^{19}$	2	$\log_2 524,288 = 19$	$\lfloor \log_2 524,288 \rfloor + 1 = 20$
$1,048,576 = 2^{20}$	2	$\log_2 1,048,576 = 20$	$\lfloor \log_2 1,048,576 \rfloor + 1 = 21$

Punti chiave sui numeri naturali.

- La sequenza $(d_{k-1} \dots d_1 d_0)_b$ rappresenta il numero naturale $\sum_{i=0}^{k-1} d_i b^i$.
- Con k cifre in base b si rappresentano esattamente i b^k numeri naturali dell'intervallo $[0, b^k - 1]$.
- Un numero naturale $n \geq 1$ richiede almeno $\lfloor \log_b n \rfloor + 1$ cifre in base b .

1.2 Rappresentazione dei numeri interi

I numeri interi comprendono sia valori positivi sia negativi, più lo zero. Con k bit abbiamo ancora soltanto 2^k sequenze diverse, ma ora dobbiamo usarle anche per rappresentare il segno. Si studiano due approcci: il **modulo e segno**, che è il più naturale, e il **complemento a due**, che è lo standard nei calcolatori moderni per i suoi vantaggi aritmetici. In questa sezione la base è sempre $b = 2$.

1.2.1 Rappresentazione in modulo e segno

Definizione 1.6: rappresentazione in modulo e segno

Nella **rappresentazione in modulo e segno**, un numero intero n è rappresentato usando esattamente $k \geq 2$ bit, dove:

- il bit più significativo rappresenta il **segno**: 0 se $n \geq 0$ e 1 se $n < 0$;
- i restanti $k - 1$ bit rappresentano il **modulo** $|n|$ (valore assoluto) in base 2.

Osservazione 1.7

Dati k bit, il **più grande numero intero** rappresentabile in modulo e segno è:

$$2^{k-1} - 1 \quad (1.6)$$

e il relativo **intervallo di rappresentazione** per i numeri interi è:

$$[-(2^{k-1} - 1), 2^{k-1} - 1] \quad (1.7)$$

Dimostrazione. Il modulo è un numero naturale scritto con $k - 1$ bit. Per l'Osservazione 1.4 con $b = 2$ e $k - 1$ cifre, il modulo più grande è formato dalla cifra 1 ripetuta $k - 1$ volte:

$$\sum_{i=0}^{k-2} 1 \cdot 2^i = \frac{2^{k-1} - 1}{2 - 1} = 2^{k-1} - 1$$

Con bit di segno 0 si ottiene il più grande intero $2^{k-1} - 1$, e con bit di segno 1 il più piccolo intero $-(2^{k-1} - 1)$. Ogni intero intermedio è rappresentato, poiché il suo valore assoluto è al più $2^{k-1} - 1$. $\square$

Esempio 1.6: modulo e segno con 4 bit

Con $k = 4$ bit, il primo bit è il segno e gli ultimi tre bit sono il modulo:

Binario	k	Segno	Decimale
$(0000)_2$	4	0 (non negativo)	$+(0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0) = +0$
$(0001)_2$	4	0 (non negativo)	$+(0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0) = 1$
$(0010)_2$	4	0 (non negativo)	$+(0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0) = 2$
$(0011)_2$	4	0 (non negativo)	$+(0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0) = 3$
$(0100)_2$	4	0 (non negativo)	$+(1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0) = 4$
$(0101)_2$	4	0 (non negativo)	$+(1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0) = 5$
$(0110)_2$	4	0 (non negativo)	$+(1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0) = 6$
$(0111)_2$	4	0 (non negativo)	$+(1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0) = 7$
$(1000)_2$	4	1 (negativo)	$-(0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0) = -0$
$(1001)_2$	4	1 (negativo)	$-(0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0) = -1$
$(1010)_2$	4	1 (negativo)	$-(0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0) = -2$
$(1011)_2$	4	1 (negativo)	$-(0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0) = -3$
$(1100)_2$	4	1 (negativo)	$-(1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0) = -4$
$(1101)_2$	4	1 (negativo)	$-(1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0) = -5$
$(1110)_2$	4	1 (negativo)	$-(1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0) = -6$
$(1111)_2$	4	1 (negativo)	$-(1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0) = -7$

L'intervallo di rappresentazione è $[-7, 7]$, come afferma l'Osservazione 1.7 con $2^{4-1} - 1 = 7$.

La tabella dell'Esempio 1.6 mostra i due punti deboli del modulo e segno.

- Lo zero ha **due rappresentazioni**: $(0000)_2 = +0$ e $(1000)_2 = -0$. Quindi $2^k - 1$ interi diversi sono rappresentati con 2^k sequenze.
- La somma di due interi non può essere eseguita come somma delle due sequenze viste come numeri naturali. Ad esempio, $3 + (-1) = 2$, ma sommando $(0011)_2$ e $(1001)_2$ come numeri naturali si ottiene $3 + 9 = 12 = (1100)_2$, che in modulo e segno rappresenta -4 . I bit di segno vanno trattati separatamente, e questo complica le operazioni aritmetiche.

Per questi motivi il modulo e segno non è la rappresentazione standard degli interi nei calcolatori.

1.2.2 Rappresentazione in complemento a due

Il complemento a due risolve entrambi i problemi: lo zero ha una rappresentazione unica e la somma funziona esattamente come per i numeri naturali.

Definizione 1.8: rappresentazione in complemento a due

Nella **rappresentazione in complemento a due** (base 2), un intero n è rappresentato usando esattamente k bit nel modo seguente:

- se $n \geq 0$, si usa la rappresentazione binaria standard di n con k bit;
- se $n < 0$, la rappresentazione di n è la rappresentazione binaria del numero naturale $2^k + n$ con k bit.

Nel complemento a due, il bit più significativo indica il segno:

- 0: numero non negativo;
- 1: numero negativo.

Per leggere una sequenza il cui bit più significativo è 1, si calcola il suo valore r come numero naturale: l'intero rappresentato è $r - 2^k$.

- Lo zero ha una rappresentazione unica: $(00 \dots 0)_2$.
- Somma e sottrazione si eseguono come per i numeri naturali, modulo 2^k .
- È la rappresentazione standard usata nei calcolatori moderni.

Esempio 1.7: complemento a due con 4 bit

Con $k = 4$ bit, le sequenze con bit più significativo 0 rappresentano $0, 1, \dots, 7$; le sequenze con bit più significativo 1 rappresentano il loro valore naturale meno $2^4 = 16$:

Binario	k	Segno	Decimale
$(0000)_2$	4	non negativo	0
$(0001)_2$	4	non negativo	1
$(0010)_2$	4	non negativo	2
$(0011)_2$	4	non negativo	3
$(0100)_2$	4	non negativo	4
$(0101)_2$	4	non negativo	5
$(0110)_2$	4	non negativo	6
$(0111)_2$	4	non negativo	7
$(1000)_2$	4	negativo	$8 - 16 = -8$
$(1001)_2$	4	negativo	$9 - 16 = -7$
$(1010)_2$	4	negativo	$10 - 16 = -6$
$(1011)_2$	4	negativo	$11 - 16 = -5$
$(1100)_2$	4	negativo	$12 - 16 = -4$
$(1101)_2$	4	negativo	$13 - 16 = -3$
$(1110)_2$	4	negativo	$14 - 16 = -2$
$(1111)_2$	4	negativo	$15 - 16 = -1$

La Figura 1.1 dispone le 16 sequenze dell'Esempio 1.7 su una circonferenza: avanzare di un passo in senso orario aggiunge 1 al valore naturale della sequenza, modulo $2^4 = 16$. Gli interi rappresentati crescono da -8 a 7 , e l'unico salto è tra $(0111)_2 = 7$ e $(1000)_2 = -8$.

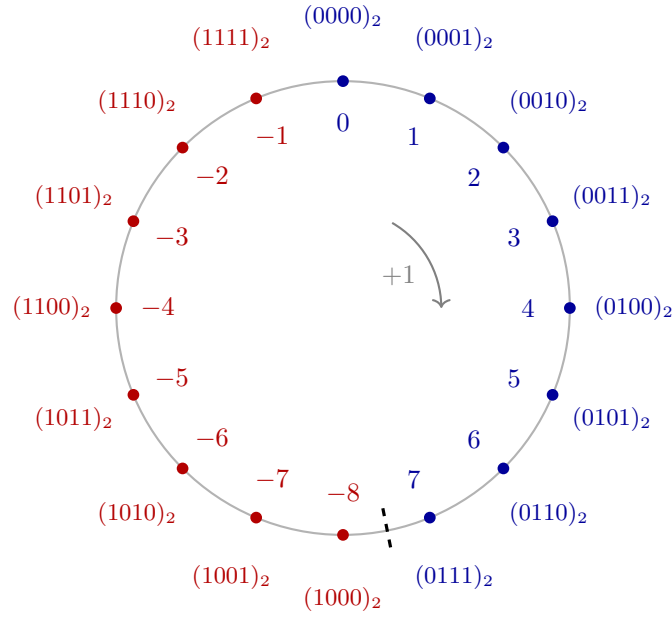


Figura 1.1: Le 16 sequenze di 4 bit su una circonferenza, con gli interi che rappresentano in complemento a due: interi non negativi in blu (bit più significativo 0), interi negativi in rosso (bit più significativo 1). Il segno tratteggiato tra $(0111)_2 = 7$ e $(1000)_2 = -8$ è l'unico punto in cui l'intero rappresentato fa un salto.

Calcolare $2^k + n$ con una sottrazione non è l'unico modo per ottenere la rappresentazione di un intero negativo: la seguente osservazione fornisce una procedura che lavora direttamente sui bit.

Osservazione 1.9

Sia n un intero con $-2^{k-1} \leq n < 0$. La rappresentazione in complemento a due di n con k bit si ottiene dalla rappresentazione binaria di $|n|$ con k bit in due passi:

1. si invertono tutti i bit (ogni 0 diventa 1 e ogni 1 diventa 0);
2. si aggiunge 1 al risultato, come numero naturale.

Dimostrazione. Sia $(d_{k-1} \dots d_1 d_0)_2$ la rappresentazione binaria di $|n|$ con k bit. Invertendo i bit si ottengono le cifre $1 - d_i$, quindi il numero naturale:

$$\sum_{i=0}^{k-1} (1 - d_i) 2^i = \sum_{i=0}^{k-1} 2^i - \sum_{i=0}^{k-1} d_i 2^i = (2^k - 1) - |n|$$

Aggiungendo 1 si ottiene $2^k - |n| = 2^k + n$, che è la rappresentazione in complemento a due di n per la Definizione 1.8. Poiché $1 \leq |n| \leq 2^{k-1}$, si ha $2^{k-1} \leq 2^k + n \leq 2^k - 1$, quindi il risultato ha ancora k bit. $\square$

Esempio 1.8: complemento a due di -5 con 8 bit

Calcoliamo la rappresentazione di $n = -5$ con $k = 8$ bit in entrambi i modi.

- Passo 1: rappresentazione binaria di $|n| = 5$ con 8 bit: $(00000101)_2$.
- Passo 2: si invertono tutti i bit: $(11111010)_2$, il cui valore naturale è $250 = (2^8 - 1) - 5$.
- Passo 3: si aggiunge 1: $(11111010)_2 + 1 = (11111011)_2$.
- Verifica con la Definizione 1.8: $2^8 + (-5) = 256 - 5 = 251 = 128 + 64 + 32 + 16 + 8 + 2 + 1 = (11111011)_2$.

1.2.3 Intervallo di rappresentazione**Osservazione 1.10**

Dati k bit, il più grande intero rappresentabile in complemento a due è:

$$2^{k-1} - 1 \quad (1.8)$$

il più piccolo intero rappresentabile è:

$$-2^{k-1} \quad (1.9)$$

e quindi l'intervallo di rappresentazione è:

$$[-2^{k-1}, 2^{k-1} - 1] \quad (1.10)$$

Dimostrazione. Con k bit ci sono 2^k possibili stringhe binarie, i cui valori naturali sono $0, 1, \dots, 2^k - 1$.

Le stringhe il cui bit più significativo è 0 hanno valori naturali $0, 1, \dots, 2^{k-1} - 1$ e rappresentano gli interi non negativi:

$$0, 1, 2, \dots, 2^{k-1} - 1$$

Le stringhe il cui bit più significativo è 1 hanno valori naturali $r = 2^{k-1}, \dots, 2^k - 1$ e rappresentano gli interi $r - 2^k$, cioè gli interi negativi:

$$-2^{k-1}, -2^{k-1} + 1, \dots, -1$$

Quindi il più grande intero rappresentabile è $2^{k-1} - 1$, il più piccolo è -2^{k-1} , e l'intervallo è:

$$[-2^{k-1}, 2^{k-1} - 1].$$

□

Rispetto al modulo e segno, l'intervallo contiene un intero in più, -2^{k-1} : la sequenza $(10\dots0)_2$, che era -0 in modulo e segno, ora rappresenta -2^{k-1} . Tutte le 2^k sequenze rappresentano interi diversi.

Esempio 1.9: intervalli di rappresentazione in complemento a due

k	-2^{k-1}	$2^{k-1} - 1$	intervallo di rappresentazione
1	-1	0	$[-1, 0]$
2	-2	1	$[-2, 1]$
3	-4	3	$[-4, 3]$
4	-8	7	$[-8, 7]$
8	-128	127	$[-128, 127]$
16	-32,768	32,767	$[-32,768, 32,767]$
32	-2,147,483,648	2,147,483,647	$[-2,147,483,648, 2,147,483,647]^1$
64	-2^{63}	$2^{63} - 1$	$[-2^{63}, 2^{63} - 1]^2$

¹ Intervallo di rappresentazione di `int` in C e C++ (4 byte).

² Intervallo di rappresentazione di `long long int` in C e C++ (8 byte); $2^{63} = 9,223,372,036,854,775,808$.

1.2.4 Numero minimo di bit

Osservazione 1.11

Il numero minimo di bit necessari per rappresentare un intero n in complemento a due è il più piccolo intero $k \geq 1$ tale che

$$-2^{k-1} \leq n \leq 2^{k-1} - 1. \quad (1.11)$$

Dimostrazione. Per l'Osservazione 1.10, un intero rappresentato con k bit in complemento a due appartiene all'intervallo

$$[-2^{k-1}, 2^{k-1} - 1],$$

e ogni intero di questo intervallo può essere rappresentato con k bit. Perciò il numero minimo di bit necessari per rappresentare n è esattamente il più piccolo k per cui

$$-2^{k-1} \leq n \leq 2^{k-1} - 1.$$

□

Come per i numeri naturali, la condizione dell'Osservazione 1.11 si può trasformare in una formula esplicita con i logaritmi.

Corollario 1.12

Il numero minimo di bit necessari per rappresentare un intero n in complemento a due è:

$$k = \begin{cases} 1 & \text{se } n = 0 \text{ o } n = -1 \\ \lfloor \log_2 n \rfloor + 2 & \text{se } n > 0 \\ \lceil \log_2 |n| \rceil + 1 & \text{se } n < 0 \end{cases}$$

Dimostrazione. Con $k = 1$ l'intervallo è $[-1, 0]$, e questo dà il primo caso.

- Se $n > 0$, la condizione dell'Osservazione 1.11 si riduce a $n \leq 2^{k-1} - 1$, cioè $n < 2^{k-1}$, cioè $\log_2 n < k - 1$. Il più piccolo intero $k - 1$ strettamente maggiore di $\log_2 n$ è $\lfloor \log_2 n \rfloor + 1$, quindi $k = \lfloor \log_2 n \rfloor + 2$.
- Se $n < 0$, la condizione si riduce a $-2^{k-1} \leq n$, cioè $|n| \leq 2^{k-1}$, cioè $\log_2 |n| \leq k - 1$. Il più piccolo intero $k - 1$ maggiore o uguale a $\log_2 |n|$ è $\lceil \log_2 |n| \rceil$, quindi $k = \lceil \log_2 |n| \rceil + 1$ (per $n = -1$ si ottiene $k = 1$, in accordo con il primo caso).

□

Esempio 1.10: numero minimo di bit in complemento a due

L'ultima colonna mostra la rappresentazione di n con il numero minimo di bit.

n	numero minimo di bit	rappresentazione
-1	1	1
0	1	0
1	2	01
-2	2	10
2	3	010
-4	3	100
4	4	0100
-8	4	1000
8	5	01000
-16	5	10000

Ad esempio, per $n = 8$ il Corollario 1.12 dà $\lfloor \log_2 8 \rfloor + 2 = 3 + 2 = 5$, e per $n = -16$ dà $\lceil \log_2 16 \rceil + 1 = 4 + 1 = 5$.

1.2.5 Somma e overflow

Il principale vantaggio del complemento a due è che gli interi si sommano esattamente come i numeri naturali: si sommano le due sequenze di k bit colonna per colonna, da destra a sinistra, con i riporti, e si scarta il riporto che esce dalla posizione più significativa.

Osservazione 1.13

Siano n_1 e n_2 due interi rappresentati con k bit in complemento a due. Si sommino le loro rappresentazioni come numeri naturali e si tengano soltanto i k bit meno significativi della somma.

- Se $n_1 + n_2 \in [-2^{k-1}, 2^{k-1} - 1]$, il risultato è la rappresentazione in complemento a due di $n_1 + n_2$.
- Altrimenti, la somma non può essere rappresentata con k bit e si ha un **overflow**. Un overflow si verifica se e solo se n_1 e n_2 hanno lo stesso segno e il risultato ha il bit di segno opposto.

Dimostrazione. Siano r_1 e r_2 i valori naturali delle due rappresentazioni: per la Definizione 1.8, $r_j = n_j$ oppure $r_j = 2^k + n_j$, quindi $r_1 + r_2$ differisce da $n_1 + n_2$ di 0, 2^k o 2^{k+1} . Tenere i k bit meno significativi significa sottrarre 2^k quando $r_1 + r_2 \geq 2^k$ (il riporto scartato). Quindi il risultato r soddisfa $r \in [0, 2^k - 1]$ e $r - (n_1 + n_2)$ è un multiplo di 2^k . Se $n_1 + n_2 \in [-2^{k-1}, 2^{k-1} - 1]$, anche la sua rappresentazione $\bar{r}$ appartiene a $[0, 2^k - 1]$ e $\bar{r} - (n_1 + n_2)$ vale 0 oppure 2^k . Allora $r - \bar{r}$ è un multiplo di 2^k con $|r - \bar{r}| \leq 2^k - 1$, quindi $r = \bar{r}$.

Se n_1 e n_2 hanno segni diversi, $n_1 + n_2$ sta tra n_1 e n_2 , che appartengono all'intervallo: nessun overflow. Se entrambi sono non negativi, $n_1 + n_2 \in [0, 2^k - 2]$, nessun riporto viene scartato e $r = n_1 + n_2$: si ha overflow se e solo se $n_1 + n_2 \geq 2^{k-1}$, cioè se e solo se il bit di segno di r è 1. Se entrambi sono negativi, $r_1 + r_2 = 2^{k+1} + n_1 + n_2 \geq 2^k$, quindi $r = 2^k + n_1 + n_2$: si ha overflow se e solo se $n_1 + n_2 < -2^{k-1}$, cioè se e solo se $r < 2^{k-1}$, cioè il bit di segno di r è 0. $\square$

Negli esempi, la prima riga di ogni somma contiene i riporti: il riporto scritto sopra una colonna è quello che arriva dalla colonna alla sua destra. La colonna più a sinistra riceve il riporto uscente dalla posizione più significativa, che viene scartato.

Esempio 1.11: somma senza overflow

Calcoliamo $3 + (-1)$ con $k = 4$ bit, intervallo $[-8, 7]$.

- Passo 1: rappresentazioni: $3 = (0011)_2$ e $-1 \rightarrow 2^4 - 1 = 15 = (1111)_2$.
- Passo 2: somma in colonna:

riporto	1	1	1	1	
		0	0	1	1
		1	1	1	1
somma	1	0	0	1	0

- Passo 3: si scarta il riporto uscente: il risultato è $(0010)_2 = 2 = 3 + (-1)$.

Esempio 1.12: somma con overflow

Calcoliamo $6 + 3$ con $k = 4$ bit, intervallo $[-8, 7]$. Poiché $6 + 3 = 9 > 7$, ci aspettiamo un overflow.

- Passo 1: rappresentazioni: $6 = (0110)_2$ e $3 = (0011)_2$.
- Passo 2: somma in colonna:

riporto	0	1	1	0	
		0	1	1	0
		0	0	1	1
somma	0	1	0	0	1

- Passo 3: il risultato $(1001)_2$ ha bit di segno 1 e rappresenta $9 - 16 = -7$, benché entrambi gli operandi siano positivi: si è verificato un overflow.

Punti chiave sui numeri interi.

- Modulo e segno con k bit: intervallo $[-(2^{k-1} - 1), 2^{k-1} - 1]$, due rappresentazioni dello zero, la somma richiede un trattamento speciale del segno.
- Complemento a due con k bit: un n negativo è rappresentato da $2^k + n$ (equivalentemente: si invertono i bit di $|n|$ e si aggiunge 1); intervallo $[-2^{k-1}, 2^{k-1} - 1]$; zero unico.
- Nel complemento a due, la somma è la somma di numeri naturali modulo 2^k ; si ha overflow solo quando due operandi con lo stesso segno danno un risultato con il segno opposto.

1.3 Rappresentazione dei numeri reali

I numeri naturali e interi sono rappresentati esattamente, purché appartengano all'intervallo di rappresentazione. I numeri reali sono diversi: tra due numeri reali qualsiasi ce ne sono infiniti altri, e la maggior parte dei numeri reali (ad esempio $\frac{1}{3}$ o $\sqrt{2}$) ha infinite cifre in qualunque base. Con un numero finito di bit, un numero reale può quindi essere rappresentato, in generale, solo in modo **approssimato**. In questa sezione studiamo la rappresentazione in virgola mobile usata dai calcolatori.

1.3.1 Cifre dopo la virgola

I sistemi di numerazione posizionali si estendono alle cifre dopo la virgola: le cifre a destra della virgola si moltiplicano per le potenze negative della base.

Definizione 1.14: sequenza di cifre con parte frazionaria

Una sequenza di k cifre prima della virgola e h cifre dopo la virgola in base b si indica con:

$$(d_{k-1} \dots d_1 d_0 . d_{-1} d_{-2} \dots d_{-h})_b \quad (1.12)$$

e rappresenta il numero:

$$\sum_{i=-h}^{k-1} d_i b^i \quad (1.13)$$

Esempio 1.13: numeri binari con parte frazionaria

- $(101.11)_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} = 4 + 1 + 0.5 + 0.25 = 5.75$
- $(0.011)_2 = 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} = 0.25 + 0.125 = 0.375$
- Il numero decimale 0.1 non ha una rappresentazione binaria finita: le sue cifre binarie dopo la virgola si ripetono all'infinito,

$$0.1 = (0.0001100110011\dots)_2$$

Scrivere un numero con un numero fissato di cifre prima e dopo la virgola (**virgola fissa**) spreca cifre: i numeri molto grandi richiedono molte cifre prima della virgola, quelli molto piccoli molte cifre dopo. La rappresentazione in virgola mobile lascia “fluttuare” la virgola, come nella notazione scientifica 6.02×10^{23} .

Definizione 1.15: notazione scientifica binaria normalizzata

Ogni numero reale $x \neq 0$ si può scrivere nella **notazione scientifica binaria normalizzata**:

$$x = (-1)^s \cdot (1.f_1 f_2 f_3 \dots)_2 \cdot 2^e \quad (1.14)$$

dove $s \in \{0, 1\}$ è il **segno**, la sequenza $(1.f_1 f_2 f_3 \dots)_2$ è la **mantissa** (o *significand*), i cui bit $f_1, f_2, f_3, \dots$ dopo la virgola formano la **parte frazionaria**, e l'intero e è l'**esponente**.

La mantissa è sempre compresa tra 1 (incluso) e 2 (escluso): la sua prima cifra è sempre 1, quindi non serve memorizzarla. Ad esempio, $5.75 = (101.11)_2 = (1.0111)_2 \cdot 2^2$: la virgola si sposta di due posizioni a sinistra e l'esponente è 2.

1.3.2 Lo standard IEEE 754

I calcolatori memorizzano i numeri in virgola mobile secondo lo standard **IEEE 754** (IEEE: Institute of Electrical and Electronics Engineers). Ne descriviamo i due formati più comuni, concentrandoci su quello a 32 bit.

Definizione 1.16: IEEE 754 singola precisione

Nel formato **IEEE 754 a singola precisione**, un numero reale è memorizzato con 32 bit divisi in tre campi:

- il **segno** s : 1 bit;
- l'**esponente polarizzato** E : 8 bit, che memorizzano il numero naturale $E = e + 127$, dove e è l'esponente e 127 è la **polarizzazione** (*bias*);
- la **parte frazionaria** f : 23 bit, che memorizzano i bit $f_1 f_2 \dots f_{23}$ della mantissa dopo la virgola.

Per $1 \leq E \leq 254$ (**numeri normalizzati**), il numero rappresentato è:

$$x = (-1)^s \cdot (1.f_1 f_2 \dots f_{23})_2 \cdot 2^{E-127} \quad (1.15)$$

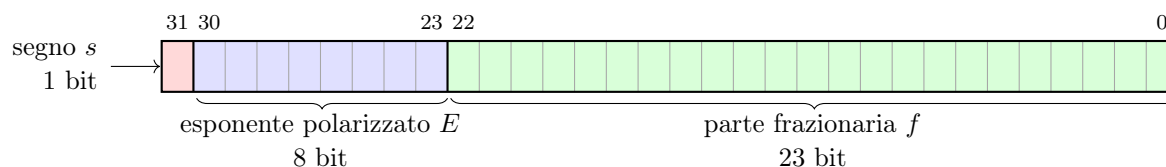


Figura 1.2: I 32 bit del formato IEEE 754 a singola precisione, numerati da 31 (più significativo) a 0 (meno significativo).

- L'esponente è memorizzato con una polarizzazione anziché in complemento a due, in modo che E sia un numero naturale tra 0 e 255.
- I due valori estremi di E sono riservati. $E = 0$ si usa per lo zero (tutti i bit della parte frazionaria uguali a 0; ci sono due zeri, $+0$ e -0 , come nel modulo e segno) e per i numeri molto piccoli detti *subnormali*. $E = 255$ si usa per $+\infty$ e $-\infty$ (parte frazionaria uguale a 0, ad esempio il risultato di $1/0$) e per *NaN*, "Not a Number" (parte frazionaria diversa da 0, ad esempio il risultato di $0/0$).
- Il formato a **doppia precisione** ha la stessa struttura con 64 bit: 1 bit di segno, 11 bit di esponente con polarizzazione 1023 e 52 bit di parte frazionaria.

Formato	bit totali	segno	esponente	parte frazionaria	polarizzazione
singola precisione (C++ <code>float</code>)	32	1	8	23	127
doppia precisione (C++ <code>double</code>)	64	1	11	52	1023

Esempio 1.14: da un numero reale alla sua rappresentazione IEEE 754 a singola precisione

Rappresentiamo $x = 5.75$ in singola precisione.

- Passo 1 (binario): $5.75 = 4 + 1 + 0.5 + 0.25 = (101.11)_2$.
- Passo 2 (normalizzazione): $(101.11)_2 = (1.0111)_2 \cdot 2^2$, quindi $s = 0$, $e = 2$ e la parte frazionaria inizia con 0111.
- Passo 3 (esponente polarizzato): $E = e + 127 = 129 = 128 + 1 = (10000001)_2$.
- Passo 4 (parte frazionaria): i bit dopo la virgola, completati con zeri fino a 23 bit: 01110000000000000000000.

I 32 bit sono:

$$\underbrace{0}_s \quad \underbrace{10000001}_E \quad \underbrace{01110000000000000000000}_f$$

Esempio 1.15: da una rappresentazione IEEE 754 a singola precisione a un numero reale

Leggiamo i 32 bit:

$$\underbrace{1}_s \quad \underbrace{10000010}_E \quad \underbrace{01000000000000000000000}_f$$

- Passo 1 (segno): $s = 1$, il numero è negativo.
- Passo 2 (esponente): $E = (10000010)_2 = 128 + 2 = 130$, quindi $e = E - 127 = 3$.
- Passo 3 (mantissa): $(1.01)_2 = 1 + 0.25 = 1.25$.
- Passo 4 (valore): $x = (-1)^1 \cdot 1.25 \cdot 2^3 = -10$.

Quando le cifre binarie di un numero non entrano nei 23 bit della parte frazionaria, il numero viene **arrotondato** al più vicino numero rappresentabile.

Esempio 1.16: il numero 0.1 non è rappresentabile esattamente

Per l'Esempio 1.13, $0.1 = (0.000110011\dots)_2 = (1.100110011001\dots)_2 \cdot 2^{-4}$, la cui parte frazionaria non termina mai. In singola precisione la parte frazionaria viene arrotondata a

23 bit, e i bit memorizzati sono:

$$\underbrace{0}_s \quad \underbrace{01111011}_{E=123} \quad \underbrace{10011001100110011001101}_f$$

Il numero effettivamente memorizzato è 0.100000001490116119384765625, non 0.1. In doppia precisione l'errore è più piccolo ma comunque presente: il valore memorizzato è 0.10000000000000000055511151231..., e per questo motivo il calcolo $0.1 + 0.2$ in doppia precisione dà 0.300000000000000004 e non il valore in doppia precisione di 0.3.

1.3.3 Campo di variazione e precisione

Il numero finito di bit dell'esponente limita il **campo di variazione** (*range*) dei numeri rappresentabili, e il numero finito di bit della parte frazionaria ne limita la **precisione**.

Osservazione 1.17

Nello standard IEEE 754 a singola precisione:

- il più grande numero finito è $(2 - 2^{-23}) \cdot 2^{127} \approx 3.4028235 \times 10^{38}$;
- il più piccolo numero normalizzato positivo è $2^{-126} \approx 1.1754944 \times 10^{-38}$;
- la distanza tra 1 e il successivo numero rappresentabile è $2^{-23} \approx 1.1920929 \times 10^{-7}$ (detta **epsilon macchina**).

Dimostrazione. Il più grande numero finito ha $s = 0$, il più grande esponente polarizzato normalizzato $E = 254$, cioè $e = 127$, e tutti i 23 bit della parte frazionaria uguali a 1. La sua mantissa è:

$$(1.\underbrace{11\dots1}_{23})_2 = \sum_{i=0}^{23} 2^{-i} = 2 - 2^{-23}$$

quindi il numero è $(2 - 2^{-23}) \cdot 2^{127}$. Il più piccolo numero normalizzato positivo ha $E = 1$, cioè $e = -126$, e tutti i bit della parte frazionaria uguali a 0, quindi vale $1 \cdot 2^{-126}$. Infine, $1 = (1.00\dots0)_2 \cdot 2^0$, e il successivo numero rappresentabile si ottiene ponendo a 1 l'ultimo bit della parte frazionaria, che ha peso 2^{-23} quando $e = 0$: è $1 + 2^{-23}$. $\square$

L'epsilon macchina misura la **precisione relativa** del formato: la singola precisione corrisponde a circa 7 cifre decimali significative, la doppia precisione (epsilon macchina $2^{-52} \approx 2.2 \times 10^{-16}$) a circa 16. Di conseguenza, due espressioni matematicamente uguali possono dare risultati diversi in virgola mobile, e i numeri in virgola mobile non vanno confrontati con l'operatore di uguaglianza senza cautela.

Punti chiave sui numeri reali.

- Un numero in virgola mobile è memorizzato come segno, esponente e parte frazionaria:

$$x = (-1)^s \cdot (1.f)_2 \cdot 2^{E-\text{bias}}$$

- IEEE 754 singola precisione: $1 + 8 + 23$ bit, polarizzazione 127; doppia precisione: $1 + 11 + 52$ bit, polarizzazione 1023.
- I bit dell'esponente determinano il campo di variazione, i bit della parte frazionaria la precisione; la maggior parte dei numeri reali (ad esempio 0.1) è memorizzata con un errore di arrotondamento.

Pseudocodice

Introduzione

Nel capitolo precedente abbiamo visto come sono rappresentati i dati di un algoritmo, i numeri. In questo capitolo fissiamo il linguaggio usato in queste note per descrivere gli algoritmi stessi: lo **pseudocodice**. Lo pseudocodice è abbastanza preciso da descrivere senza ambiguità ogni passo di un algoritmo, e abbastanza semplice da poter essere letto senza conoscere alcun linguaggio di programmazione.

Il capitolo introduce gli elementi dello pseudocodice uno alla volta, dal più semplice al più strutturato:

- variabili, dichiarazioni e assegnamenti, e le operazioni di base su di essi;
- i costrutti di base che controllano l'ordine di esecuzione: istruzioni condizionali, cicli, e le istruzioni di controllo dei cicli **break** e **continue**;
- array con uno, due o più indici;
- funzioni, che danno un nome a un algoritmo, con il suo input e il suo output;
- insiemi.

Ogni costrutto è seguito da esempi, eseguiti passo per passo su piccole istanze. Eseguire un algoritmo a mano, tenendo traccia dei valori delle sue variabili, è il modo migliore per capire che cosa fa.

2.1 Introduzione allo pseudocodice

Definizione 2.1: algoritmo

Un **algoritmo** è una sequenza finita di istruzioni non ambigue che, a partire da alcuni dati forniti come **input**, produce in un numero finito di passi un risultato fornito come **output**.

Definizione 2.2: pseudocodice

Lo **pseudocodice** è una descrizione di alto livello di un algoritmo che usa un misto di linguaggio naturale e di strutture simili a quelle dei linguaggi di programmazione.

- Lo pseudocodice non è fatto per essere eseguito da un calcolatore, ma per comunicare agli esseri umani la logica di un algoritmo.
- Lo pseudocodice permette di concentrarsi sulla logica e sulla struttura di un algoritmo senza preoccuparsi della sintassi di uno specifico linguaggio di programmazione.
- **Convenzione:** in queste note tutte le parole riservate (parole chiave) dello pseudocodice sono scritte in grassetto, ad esempio **if**, **while**, **for**, **return**, ecc.

Vantaggi dello pseudocodice:

- Indipendente dal linguaggio: può essere compreso da programmatori che usano linguaggi diversi
- Facile da scrivere e da modificare
- Si concentra sulla logica dell'algoritmo anziché sui dettagli di implementazione
- Facilita la comunicazione tra sviluppatori e persone non tecniche
- Utile per la pianificazione prima della scrittura effettiva del codice

Principi fondamentali:

- Usare nomi di variabili chiari e descrittivi
- Mantenere le istruzioni semplici e concise
- Usare l'indentazione per mostrare la struttura e l'annidamento
- Usare costrutti standard (**if-then-else**, **while**, **for**, ecc.)
- Inserire commenti quando necessario per chiarire le operazioni complesse

Buone pratiche:

- **Indentazione:** usare un'indentazione coerente per mostrare la struttura a blocchi
- **Variabili:** usare nomi descrittivi in corsivo o in testo normale
- **Commenti:** aggiungere commenti (in linguaggio naturale) per spiegare i passi complessi
- **Input/Output:** specificare chiaramente i parametri di input e l'output (valori restituiti)
- **Numeri di riga:** utili per fare riferimento a righe specifiche
- **Coerenza:** mantenere la stessa notazione e lo stesso stile in tutto il testo

2.2 Variabili e assegnamenti

I dati di un algoritmo sono memorizzati in **variabili**: contenitori dotati di un nome, il cui valore può cambiare durante l'esecuzione. Una variabile viene prima creata con una dichiarazione e poi modificata con degli assegnamenti.

2.2.1 Variabili

- Le variabili devono essere dichiarate prima dell'uso
- L'operatore di dichiarazione è “:=”
- La dichiarazione specifica il nome della variabile e il suo valore iniziale

```
1 variabile := valore_iniziale
```

Esempio 2.1: dichiarazione di variabili

Sette variabili dichiarate con i loro valori iniziali: sono ammessi valori interi, razionali, $-\infty$ e logici (**true/false**).

```
1 n := 5
2 m := 10
3 x := -3/5
4 sum := 0
5 count := 1
6 max :=  $-\infty$ 
7 status := true
```

2.2.2 Assegnamento

- L'operatore di assegnamento è " $\leftarrow$ "
- Assegna il valore di un'espressione a una variabile dichiarata in precedenza: prima si valuta l'espressione a destra, poi il suo valore sostituisce il vecchio valore della variabile

```
1 variabile  $\leftarrow$  espressione
```

Esempio 2.2: assegnamento

Con le variabili dell'Esempio 2.1:

```
1  $n \leftarrow 10$ 
2  $sum \leftarrow n + m$ 
3  $count \leftarrow count + 1$ 
```

Dopo i tre assegnamenti si ha $n = 10$, $sum = 10 + 10 = 20$ e $count = 1 + 1 = 2$. Nell'ultimo assegnamento la stessa variabile compare in entrambi i membri: il vecchio valore 1 viene usato per calcolare il nuovo valore 2.

Esempio 2.3: scambio dei valori di due variabili

Per scambiare i valori di due variabili x e y serve una terza variabile temporanea tmp : il solo assegnamento $x \leftarrow y$ sovrascriverebbe il vecchio valore di x .

```
1  $tmp := x$ 
2  $x \leftarrow y$ 
3  $y \leftarrow tmp$ 
```

Esecuzione passo per passo con $x = 3$ e $y = 8$:

istruzione	x	y	tmp
–	3	8	–
$tmp := x$	3	8	3
$x \leftarrow y$	8	8	3
$y \leftarrow tmp$	8	3	3

2.3 Operazioni di base

Le espressioni combinano variabili e valori con le seguenti operazioni.

Operazioni matematiche:

- $+$, $-$, $\times$, $/$: aritmetica di base
- $a \bmod b$: modulo, cioè il resto della divisione intera di a per b
- $\lfloor x \rfloor$: parte intera inferiore (il più grande intero $\leq x$)
- $\lceil x \rceil$: parte intera superiore (il più piccolo intero $\geq x$)
- $|x|$: valore assoluto
- $\max(a, b)$, $\min(a, b)$: massimo e minimo

Operatori di confronto:

- $=$, $\neq$: uguale, diverso
- $<$, $\leq$, $>$, $\geq$: minore, minore o uguale, maggiore, maggiore o uguale

Operatori logici:

- **and**, **or**, **not**: operazioni logiche

Input/Output:

- **read**(variabile): legge un input dall'utente
- **print**(espressione): stampa un valore

Esempio 2.4: valutazione di espressioni

- $17 \bmod 5 = 2$, poiché $17 = 3 \cdot 5 + 2$; in particolare, un intero i è multiplo di 3 se e solo se $i \bmod 3 = 0$.
- $\lfloor 7/2 \rfloor = \lfloor 3.5 \rfloor = 3$ e $\lceil 7/2 \rceil = \lceil 3.5 \rceil = 4$.
- $|-3| = 3$, $\max(-3, 2) = 2$ e $\min(-3, 2) = -3$.
- $(5 > 3)$ **and** $(2 > 4)$ è **false**, mentre $(5 > 3)$ **or** $(2 > 4)$ è **true**.

2.4 Costrutti di base

Le istruzioni dello pseudocodice vengono eseguite una dopo l'altra, dall'alto verso il basso. Due tipi di costrutti cambiano quest'ordine: le **istruzioni condizionali**, che eseguono un blocco solo quando vale una condizione, e i **cicli**, che eseguono un blocco più volte.

2.4.1 Istruzioni condizionali

2.4.1.1 If-then

- Esegue un blocco di istruzioni solo se una condizione è vera

```

1 if condizione then
2   | istruzioni
3 end

```

Esempio 2.5: if-then

Verifica se un numero x è positivo:

```

1 positive := false
2 if  $x > 0$  then
3   | positive ← true
4   | print( $x$  è positivo)
5 end

```

Esecuzione passo per passo:

Istanza 1: $x = 5$

x	$x > 0?$	<i>positive</i>
–	–	false
5	Sì	true

Output: 5 è positivo

Istanza 2: $x = -3$

x	$x > 0?$	<i>positive</i>
–	–	false
-3	No	false

Output: non viene stampato nulla.

2.4.1.2 If-then-else

- Esegue un blocco se la condizione è vera, e un altro blocco se è falsa

```

1 if condizione then
2   |   istruzioni se vera
3 else
4   |   istruzioni se falsa
5 end

```

Esempio 2.6: if-then-else

Calcolo del valore assoluto di un numero x :

```

1 result := 0
2 if  $x \geq 0$  then
3   |   result  $\leftarrow x$ 
4 else
5   |   result  $\leftarrow -x$ 
6 end

```

Esecuzione passo per passo:

Istanza 1: $x = 7$

x	$x \geq 0?$	<i>result</i>
–	–	0
7	Sì	7

Risultato: $result = 7$

Istanza 2: $x = -4$

x	$x \geq 0?$	<i>result</i>
–	–	0
-4	No	4

Risultato: $result = 4$

2.4.2 Cicli

2.4.2.1 Ciclo while

- Ripete un blocco di istruzioni finché una condizione è vera
- La condizione viene controllata prima di ogni iterazione
- Se la condizione è inizialmente falsa, il corpo del ciclo non viene mai eseguito

```

1 while condizione do
2   | istruzioni
3 end

```

Esempio 2.7: ciclo while

Calcolo della somma dei numeri da 1 a n :

```

1 sum := 0
2 i := 1
3 while  $i \leq n$  do
4   |  $sum \leftarrow sum + i$ 
5   |  $i \leftarrow i + 1$ 
6 end

```

Esecuzione passo per passo:

Istanza: $n = 5$. Ogni riga mostra il valore di i quando la condizione viene controllata, l'esito del controllo, e i valori di sum e i alla fine dell'iterazione.

i	$i \leq n?$	sum	i (dopo)
–	–	0	1
1	Sì	1	2
2	Sì	3	3
3	Sì	6	4
4	Sì	10	5
5	Sì	15	6
6	No	15	–

Risultato: $sum = 15$

Nell'esempio precedente il numero di iterazioni è noto in anticipo. Il ciclo while è necessario quando non lo è, come nell'esempio seguente.

Esempio 2.8: ciclo while

Calcolo di quante volte un numero x va dimezzato prima che diventi minore o uguale a una soglia $\varepsilon > 0$:

```

1 count := 0
2 while  $x > \varepsilon$  do
3   |  $x \leftarrow x/2$ 
4   |  $count \leftarrow count + 1$ 
5 end

```

Esecuzione passo per passo:

Istanza 1: $x = 10, \varepsilon = 1$

x	$count$	$x > \varepsilon?$	x (dopo)	$count$ (dopo)
10	0	Sì	5	1
5	1	Sì	2.5	2
2.5	2	Sì	1.25	3
1.25	3	Sì	0.625	4
0.625	4	No $\Rightarrow$ uscita	–	4

Risultato: $count = 4$

Istanza 2: $x = 100, \varepsilon = 10$

x	$count$	$x > \varepsilon?$	x (dopo)	$count$ (dopo)
100	0	Sì	50	1
50	1	Sì	25	2
25	2	Sì	12.5	3
12.5	3	Sì	6.25	4
6.25	4	No $\Rightarrow$ uscita	–	4

Risultato: $count = 4$

2.4.2.2 Ciclo for

- Ripete un blocco di istruzioni per un numero di volte specificato
- Di solito si usa quando il numero di iterazioni è noto in anticipo

```

1 for variabile := valore_iniziale to valore_finale do
2   | istruzioni
3 end

```

Il ciclo `for` è un modo compatto di scrivere un particolare ciclo `while`: la variabile del ciclo viene dichiarata con il valore iniziale, la condizione la confronta con il valore finale, e viene incrementata di 1 alla fine di ogni iterazione.

Il ciclo a sinistra è equivalente al ciclo a destra:

```

1 for  $i := s$  to  $e$  do
2   | istruzioni
3 end

```

```

1  $i := s$ 
2 while  $i \leq e$  do
3   | istruzioni
4   |  $i \leftarrow i + 1$ 
5 end

```

In particolare, il corpo viene eseguito $e - s + 1$ volte se $s \leq e$, e non viene mai eseguito se $s > e$.

Esempio 2.9: ciclo `for`

Calcolo della somma dei numeri da 1 a n con un ciclo `for` (confrontare con l'Esempio 2.7):

```

1  $sum := 0$ 
2 for  $i := 1$  to  $n$  do
3   |  $sum \leftarrow sum + i$ 
4 end

```

Esecuzione passo per passo:

Istanza: $n = 5$

i	$i \leq n?$	sum
–	–	0
1	Sì	1
2	Sì	3
3	Sì	6
4	Sì	10
5	Sì	15
6	No	15

Risultato: $sum = 15$

Il ciclo `for` può usare facoltativamente un valore di **passo** (*step*), per incrementare la variabile del ciclo di valori diversi da 1.

Esempio 2.10: ciclo for con passoCalcolo della somma dei primi n numeri pari ($2, 4, \dots, 2n$):

```

1  $sum := 0$ 
2 for  $i := 2$  to  $2n$  step 2 do
3   |  $sum \leftarrow sum + i$ 
4 end

```

Esecuzione passo per passo:**Istanza:** $n = 4$ (even numbers: 2, 4, 6, 8)

i	$i \leq 2n?$	sum
–	–	0
2	Sì	2
4	Sì	6
6	Sì	12
8	Sì	20
10	No	20

Risultato: $sum = 20$ **2.4.3 Istruzioni di controllo dei cicli**

A volte un ciclo deve essere interrotto prima della sua fine naturale, oppure una singola iterazione deve essere saltata. A questi scopi servono due istruzioni.

2.4.3.1 Istruzione break

- L'istruzione **break** esce immediatamente dal ciclo corrente
- L'esecuzione continua con la prima istruzione dopo il ciclo
- Utile quando una condizione di terminazione si verifica prima del completamento naturale del ciclo

```

1 while condizione do
2   | istruzioni
3   | if condizione_di_uscita then
4     | | istruzioni
5     | | break
6   | end
7   | istruzioni
8 end

```

```

1 for variabile := valore_iniziale to valore_finale do
2   | istruzioni
3   | if condizione_di_uscita then
4   |   | istruzioni
5   |   | break
6   | end
7   | istruzioni
8 end

```

Esempio 2.11: istruzione break

Calcolo della somma dei numeri da 1 a n , fermandosi non appena la somma supera una soglia t .

Con un ciclo while:

```

1 sum := 0
2 i := 1
3 while  $i \leq n$  do
4   |  $sum \leftarrow sum + i$ 
5   | if  $sum > t$  then
6   |   | break
7   | end
8   |  $i \leftarrow i + 1$ 
9 end

```

Con un ciclo for:

```

1 sum := 0
2 for  $i := 1$  to  $n$  do
3   |  $sum \leftarrow sum + i$ 
4   | if  $sum > t$  then
5   |   | break
6   | end
7 end

```

Esecuzione passo per passo (entrambi i cicli):

Istanza: $n = 6$, $t = 10$

i	sum	$sum > t?$
–	0	–
1	1	No
2	3	No
3	6	No
4	10	No
5	15	Sì $\Rightarrow$ break

Risultato: $sum = 15$; l'iterazione con $i = 6$ non viene mai eseguita.

2.4.3.2 Istruzione continue

- L'istruzione **continue** salta il resto dell'iterazione corrente
- L'esecuzione prosegue subito con l'iterazione successiva del ciclo (in un ciclo while, con il controllo della condizione)
- Utile quando alcune iterazioni vanno saltate in base a una condizione

```

1 while condizione do
2   istruzioni
3   if condizione_di_salto then
4     istruzioni
5     continue
6   end
7   istruzioni
8 end

1 for variabile := valore_iniziale to valore_finale do
2   istruzioni
3   if condizione_di_salto then
4     istruzioni
5     continue
6   end
7   istruzioni
8 end

```

Esempio 2.12: istruzione continue

Calcolo della somma dei numeri da 1 a n , saltando i multipli di 3.

Con un ciclo while:

```

1  $sum := 0$ 
2  $i := 1$ 
3 while  $i \leq n$  do
4   if  $i \bmod 3 = 0$  then
5      $i \leftarrow i + 1$ 
6     continue
7   end
8    $sum \leftarrow sum + i$ 
9    $i \leftarrow i + 1$ 
10 end

```

Con un ciclo **for**:

```

1  $sum := 0$ 
2 for  $i := 1$  to  $n$  do
3   if  $i \bmod 3 = 0$  then
4     continue
5   end
6    $sum \leftarrow sum + i$ 
7 end

```

Nel ciclo **while** la variabile i deve essere incrementata prima di **continue**: altrimenti la condizione $i \bmod 3 = 0$ resterebbe vera per sempre e il ciclo non terminerebbe mai. Nel ciclo **for** l'incremento è automatico.

Esecuzione passo per passo (entrambi i cicli):

Istanza: $n = 7$

i	$i \bmod 3 = 0?$	sum
–	–	0
1	No	1
2	No	3
3	Sì	3
4	No	7
5	No	12
6	Sì	12
7	No	19

Risultato: $sum = 19 = 1 + 2 + 4 + 5 + 7$

Punti chiave sui costrutti di base.

- **if-then** e **if-then-else** scelgono quale blocco eseguire in base a una condizione.
- **while** ripete un blocco finché una condizione è vera, controllandola prima di ogni iterazione; **for** ripete un blocco per ogni valore di una variabile del ciclo da un valore iniziale a un valore finale.

- **break** esce dal ciclo; **continue** passa all'iterazione successiva.
- Eseguire un costrutto passo per passo su una piccola istanza, con una tabella dei valori delle variabili, mostra esattamente che cosa fa.

2.5 Array

Molti algoritmi lavorano su una collezione di valori dello stesso tipo, ad esempio una lista di n numeri. Invece di dichiarare n variabili diverse, si usa un array.

- Gli array sono collezioni indicizzate di variabili e devono essere dichiarati prima dell'uso
- L'operatore di dichiarazione è "new array" seguito dall'intervallo degli indici

```
1 nome_array := new array[inizio..fine]
```

Esempio 2.13: dichiarazione di un array

Creazione di un array di dimensione n , indicizzato da 1 a n :

```
1 a := new array[1..n]
```

- La notazione $a[i]$ indica la variabile di indice i
- Gli array sono di solito indicizzati a partire da 0 o da 1: nello pseudocodice di queste note gli indici partono da 1, quindi un array di n elementi è $a[1..n]$; in molti linguaggi di programmazione, come il C++, gli indici partono da 0
- Ogni variabile dell'array deve essere inizializzata prima di essere letta e usata
- Per inizializzare tutti gli elementi allo stesso valore, si usa un ciclo **for**:

```
1 for  $i := inizio$  to  $fine$  do
2   | nome_array[ $i$ ] ← valore_iniziale
3 end
```

Esempio 2.14: assegnamento in un array

Inizializzazione e accesso alle variabili di un array $a[1..n]$ (con $n \geq 5$):

```

// Inizializza a zero tutte le variabili dell'array
1 for i := 1 to n do
2   | a[i] ← 0
3 end

// Accesso e modifica
4 x := a[5]
5 a[3] ← 42

```

Con $n = 5$ l'array vale $[0, 0, 0, 0, 0]$ dopo il ciclo, $x = 0$, e alla fine l'array vale $[0, 0, 42, 0, 0]$.

2.5.1 Array bidimensionali

- Gli array bidimensionali sono collezioni indicizzate di variabili con due indici e devono essere dichiarati prima dell'uso
- L'operatore di dichiarazione è "new array" seguito dai due intervalli degli indici
- Gli array bidimensionali sono di solito indicizzati a partire da 0 o da 1 (da 1 in queste note)

```

1 nome_array := new array[inizio1..fine1][inizio2..fine2]

```

Esempio 2.15: dichiarazione di un array bidimensionale

Creazione di un array bidimensionale di dimensione $m \times n$ (indicizzato da 1 a m e da 1 a n):

```

1 b := new array[1..m][1..n]

```

- La notazione $b[i][j]$ indica la variabile nella riga i e nella colonna j
- Ogni variabile dell'array deve essere inizializzata prima di essere letta e usata
- Per inizializzare tutti gli elementi allo stesso valore, si usano cicli **for** annidati:

```

1 for i := inizio1 to fine1 do
2   | for j := inizio2 to fine2 do
3     | nome_array[i][j] ← valore_iniziale
4   | end
5 end

```

Esempio 2.16: assegnamento in un array bidimensionale

Inizializzazione e accesso alle variabili di un array bidimensionale $b[1..m][1..n]$ (con $m \geq 2$ e $n \geq 3$):

```
// Inizializza a zero tutte le variabili dell'array bidimensionale
1 for i := 1 to m do
2   | for j := 1 to n do
3   |   | b[i][j] ← 0
4   | end
5 end

// Accesso e modifica
6 x := b[2][3]
7 b[1][1] ← 42
```

Il ciclo esterno scorre le righe e, per ogni riga i , il ciclo interno scorre tutte le colonne: gli elementi vengono visitati nell'ordine $b[1][1], b[1][2], \dots, b[1][n], b[2][1], \dots, b[m][n]$, per un totale di $m \cdot n$ assegnamenti.

2.5.2 Array multidimensionali

- Gli array multidimensionali sono collezioni indicizzate di variabili con più indici e devono essere dichiarati prima dell'uso
- L'operatore di dichiarazione è “new array” seguito dagli intervalli degli indici per ciascuna dimensione

```
1 nome_array := new array[inizio1..fine1][inizio2..fine2]...[inizio_k..fine_k]
```

Esempio 2.17: dichiarazione di un array multidimensionale

Creazione di un array a 3 dimensioni di dimensione $p \times q \times r$ (indicizzato da 1 a p , da 1 a q e da 1 a r):

```
1 t := new array[1..p][1..q][1..r]
```

- La notazione $t[i_1][i_2] \dots [i_k]$ indica la variabile di indici $i_1, i_2, \dots, i_k$
- Gli array multidimensionali sono di solito indicizzati a partire da 0 o da 1 (da 1 in queste note)
- Ogni variabile dell'array deve essere inizializzata prima di essere letta e usata
- Per inizializzare tutti gli elementi allo stesso valore, si usano cicli **for** annidati (uno per dimensione):

```

1 for  $i_1 := \text{inizio}_1$  to  $\text{fine}_1$  do
2   for  $i_2 := \text{inizio}_2$  to  $\text{fine}_2$  do
3     :
4     for  $i_k := \text{inizio}_k$  to  $\text{fine}_k$  do
5       nome_array[ $i_1$ ][ $i_2$ ]  $\cdots$  [ $i_k$ ]  $\leftarrow$  valore_iniziale
6     end
7   end
8 end

```

Esempio 2.18: assegnamento in un array multidimensionale

Inizializzazione e accesso a un array a 3 dimensioni $t[1..p][1..q][1..r]$ (con $p \geq 2$, $q \geq 2$ e $r \geq 3$):

```

// Inizializza a zero tutte le variabili dell'array a 3 dimensioni
1 for  $i := 1$  to  $p$  do
2   for  $j := 1$  to  $q$  do
3     for  $k := 1$  to  $r$  do
4        $t[i][j][k] \leftarrow 0$ 
5     end
6   end
7 end

// Accesso e modifica
8  $x := t[1][2][3]$ 
9  $t[2][1][3] \leftarrow 42$ 

```

2.6 Funzioni

Gli esempi visti finora sono frammenti di pseudocodice che lavorano su variabili supposte esistenti. Una funzione trasforma un frammento di questo tipo in un algoritmo completo, con un nome, un input e un output.

- Una **funzione** è un blocco di codice dotato di nome e riutilizzabile che svolge un compito specifico
- Una funzione riceve dati di **input** attraverso i suoi **parametri** e produce dati di **output**
- L'istruzione **return** termina la funzione e ne specifica i risultati
- Le funzioni si usano di solito per realizzare **algoritmi** o per eseguire calcoli specifici che possono essere riutilizzati in parti diverse di un programma

Sintassi:

Input : descrizione dei parametri di input

Output: descrizione dell'output / del valore restituito

```

1 function FunctionName(parametro1, parametro2, ..., parametrok)
2   |   istruzioni
3   |   return risultato
4 end

```

Nel resto di queste note ogni algoritmo è scritto come una funzione, dentro un riquadro rosso. L'input indica le **precondizioni** sui dati (ad esempio, che un array abbia almeno un elemento): la funzione è garantita funzionare solo quando esse valgono.

Esempio 2.19: valore massimo in un array

Una funzione che restituisce il valore massimo di un array: la variabile *max* parte dal primo elemento e viene aggiornata ogni volta che si trova un elemento più grande.

Algorithm: FindMaximum

Input : Un array $a[1..n]$ di $n \geq 1$ numeri interi

Output: Il valore massimo nell'array a

```

1 function FindMaximum( $a, n$ )
2   |    $max := a[1]$ 
3   |   for  $i := 2$  to  $n$  do
4   |   |   if  $a[i] > max$  then
5   |   |   |    $max \leftarrow a[i]$ 
6   |   |   end
7   |   end
8   |   return  $max$ 
9 end

```

Esecuzione passo per passo:

Istanza 1: $a = [9, 12, 3, 16, 19, 11, 12]$, $n = 7$

i	$a[i]$	$a[i] > max?$	max
–	–	–	9
2	12	Sì	12
3	3	No	12
4	16	Sì	16
5	19	Sì	19
6	11	No	19
7	12	No	19

Restituisce: $max = 19$

Istanza 2: $a = [3, 7, 2, 9, 1]$, $n = 5$

i	$a[i]$	$a[i] > max?$	max
–	–	–	3
2	7	Sì	7
3	2	No	7
4	9	Sì	9
5	1	No	9

Restituisce: $max = 9$

Esempio 2.20: conteggio degli elementi positivi di un array

Una funzione che conta il numero di elementi positivi in un array:

Algorithm: CountPositive

Input : Un array $a[1..n]$ di n numeri interi

Output: Il numero di elementi positivi nell'array a

```

1 function CountPositive( $a, n$ )
2    $count := 0$ 
3   for  $i := 1$  to  $n$  do
4     if  $a[i] > 0$  then
5        $count \leftarrow count + 1$ 
6     end
7   end
8   return  $count$ 
9 end

```

Esecuzione passo per passo:

Istanza 1: $a = [-2, 5, -1, 3, 0]$, $n = 5$

i	$a[i]$	$a[i] > 0?$	$count$
–	–	–	0
1	–2	No	0
2	5	Sì	1
3	–1	No	1
4	3	Sì	2
5	0	No	2

Restituisce: $count = 2$

Istanza 2: $a = [4, -3, 7, 2]$, $n = 4$

i	$a[i]$	$a[i] > 0?$	$count$
—	—	—	0
1	4	Sì	1
2	-3	No	1
3	7	Sì	2
4	2	Sì	3

Restituisce: $count = 3$

Esempio 2.21: verifica se un array è ordinato

Una funzione che verifica se un array è ordinato in ordine non decrescente. L'istruzione **return** dentro il ciclo termina la funzione non appena si trovano due elementi consecutivi nell'ordine sbagliato.

Algorithm: IsSorted

Input : Un array $a[1..n]$ di $n \geq 1$ numeri interi

Output: **true** se i valori dell'array a sono ordinati in ordine non decrescente e
false altrimenti

```

1 function IsSorted( $a, n$ )
2   for  $i := 1$  to  $n - 1$  do
3     if  $a[i] > a[i + 1]$  then
4       return false
5     end
6   end
7   return true
8 end

```

Esecuzione passo per passo:

Istanza 1: $a = [2, 5, 7, 9]$, $n = 4$

i	$a[i]$	$a[i+1]$	$a[i] > a[i+1]?$
1	2	5	No
2	5	7	No
3	7	9	No

Restituisce: **true**

Istanza 2: $a = [3, 1, 4, 2]$, $n = 4$

i	$a[i]$	$a[i+1]$	$a[i] > a[i+1]?$
1	3	1	Sì

Restituisce: **false**, dopo un solo confronto.

Esempio 2.22: prodotto scalare di due array

Una funzione che calcola il prodotto scalare $\sum_{i=1}^n a[i] \cdot b[i]$ di due array:

Algorithm: ScalarProduct

Input : Due array $a[1..n]$ e $b[1..n]$ di dimensione $n \geq 1$

Output: Il prodotto scalare dei due array

```

1 function ScalarProduct(a, b, n)
2   prod := a[1] · b[1]
3   for i := 2 to n do
4     | prod ← prod + a[i] · b[i]
5   end
6   return prod
7 end

```

Esecuzione passo per passo:

Istanza 1: $a = [1, 2, 3]$, $b = [4, 5, 6]$, $n = 3$

i	$a[i]$	$b[i]$	$a[i] \cdot b[i]$	$prod$
–	–	–	–	$1 \cdot 4 = 4$
2	2	5	10	$4 + 10 = 14$
3	3	6	18	$14 + 18 = 32$

Restituisce: $prod = 32$

Istanza 2: $a = [2, 0, -1]$, $b = [3, 4, 5]$, $n = 3$

i	$a[i]$	$b[i]$	$a[i] \cdot b[i]$	$prod$
–	–	–	–	$2 \cdot 3 = 6$
2	0	4	0	$6 + 0 = 6$
3	–1	5	–5	$6 + (–5) = 1$

Restituisce: $prod = 1$

Esempio 2.23: inversione di un array

Una funzione che inverte l'ordine degli elementi di un array: l'elemento $a[i]$ viene scambiato con il suo simmetrico $a[n - i + 1]$, con la variabile temporanea dell'Esempio 2.3. Bastano $\lfloor n/2 \rfloor$ scambi: con n dispari, l'elemento centrale resta al suo posto.

Algorithm: ReverseArray**Input** : Un array $a[1..n]$ di n numeri**Output**: Lo stesso array a con gli elementi in ordine inverso

```

1 function ReverseArray( $a, n$ )
2   for  $i := 1$  to  $\lfloor n/2 \rfloor$  do
3      $tmp := a[i]$ 
4      $a[i] \leftarrow a[n - i + 1]$ 
5      $a[n - i + 1] \leftarrow tmp$ 
6   end
7   return  $a$ 
8 end

```

Esecuzione passo per passo:**Istanza 1:** $a = [1, 2, 3, 4, 5]$, $n = 5$, $\lfloor 5/2 \rfloor = 2$ iterazioni

i	scambio	array a (dopo lo scambio)
–	–	$[1, 2, 3, 4, 5]$
1	$a[1] \leftrightarrow a[5]$	$[5, 2, 3, 4, 1]$
2	$a[2] \leftrightarrow a[4]$	$[5, 4, 3, 2, 1]$

Restituisce: $a = [5, 4, 3, 2, 1]$ **Istanza 2:** $a = [10, 20, 30, 40]$, $n = 4$, $\lfloor 4/2 \rfloor = 2$ iterazioni

i	scambio	array a (dopo lo scambio)
–	–	$[10, 20, 30, 40]$
1	$a[1] \leftrightarrow a[4]$	$[40, 20, 30, 10]$
2	$a[2] \leftrightarrow a[3]$	$[40, 30, 20, 10]$

Restituisce: $a = [40, 30, 20, 10]$

2.7 Insiemi

Gli array memorizzano valori in posizioni. Quando conta soltanto l'appartenenza dei valori, e non la loro posizione o le loro ripetizioni, si usa un insieme.

- Un **insieme** (*set*) è una collezione non ordinata di elementi *distinti*
- Gli insiemi devono essere dichiarati prima dell'uso con l'operatore **new set**; non occorre specificare alcuna dimensione; appena creato, un insieme è vuoto

```
1 nome_insieme := new set
```

I principali operatori sugli insiemi sono:

- **add**: inserisce un elemento nell'insieme (ignorato se già presente)

add x **to** nome_insieme

- **remove**: elimina un elemento dall'insieme (ignorato se non presente)

remove x **from** nome_insieme

- **in**: test di appartenenza, che vale **true** se x appartiene all'insieme

x **in** nome_insieme

- **size**: cardinalità (numero di elementi attualmente nell'insieme)

size(nome_insieme)

- **foreach**: ciclo sugli elementi dell'insieme; per convenzione, in queste note gli elementi vengono visitati nell'ordine di inserimento

```
1 foreach  $x$  in nome_insieme do
  | // elabora  $x$ 
2 end
```

- È possibile scorrere gli elementi di un insieme in un ordine specifico indicando un criterio di ordinamento:

```
1 foreach  $x$  in nome_insieme sorted by criterio do
  | // elabora  $x$ 
2 end
```

Esempio 2.24: dichiarazione e uso di un insieme

```

// Dichiarazione di un insieme vuoto
1 s := new set

// Riempie l'insieme con i numeri pari da 1 a n
2 for i := 1 to n do
3   | if i mod 2 = 0 then
4   |   | add i to s
5   | end
6 end

// Test di appartenenza
7 if 4 in s then
8   | print(4 appartiene a s)
9 end

// Rimuove un elemento
10 remove 4 from s

// Scorre e stampa tutti gli elementi rimasti
11 foreach x in s do
12   | print(x)
13 end

// Stampa la cardinalità
14 print(size(s))

// Stampa gli elementi in ordine non crescente
15 foreach x in s sorted by valore non crescente do
16   | print(x)
17 end

// Stampa gli elementi in ordine non decrescente
18 foreach x in s sorted by valore non decrescente do
19   | print(x)
20 end

```

Esecuzione passo per passo:

Istanza: $n = 8$

- Passo 1: il ciclo for aggiunge 2, 4, 6, 8, quindi $s = \{2, 4, 6, 8\}$.
- Passo 2: 4 appartiene a s : l'output è 4 appartiene a s.
- Passo 3: dopo la rimozione, $s = \{2, 6, 8\}$.
- Passo 4: il primo **foreach** stampa 2, 6, 8 (ordine di inserimento), poi viene stampata la cardinalità 3.
- Passo 5: gli ultimi due cicli stampano 8, 6, 2 e 2, 6, 8.

2.8 Convenzioni di stile

Concludiamo con un riepilogo della notazione usata in queste note, confrontata con altre scelte comuni.

Autori diversi possono usare convenzioni leggermente diverse:

- Dichiarazione: $:=$ (usata in queste note) oppure $=$ (alcuni autori usano lo stesso simbolo sia per la dichiarazione sia per l'assegnamento)
- Assegnamento: $\leftarrow$ (usato in queste note) oppure $=$
- Indicizzazione di un array: $a[i]$ (usata in queste note) oppure a_i
- Indicizzazione di un array bidimensionale: $b[i][j]$ (usata in queste note) oppure b_{ij}
- Commenti: `// commento`, `# commento`, oppure `(* commento *)`

L'importante è essere coerenti e chiari!

Punti chiave del capitolo.

- Le variabili si dichiarano con $:=$ e si modificano con $\leftarrow$.
- Le istruzioni condizionali e i cicli (**if**, **while**, **for**, con **break** e **continue**) controllano l'ordine di esecuzione.
- Gli array $a[1..n]$, $b[1..m][1..n]$, ... sono indicizzati da 1 nello pseudocodice; gli insiemi memorizzano elementi distinti senza posizioni.
- Un algoritmo si scrive come una **funzione** con il suo input (comprese le precondizioni), il suo output e un'istruzione **return**.

Algoritmi di ordinamento

Introduzione

L'ordinamento è uno dei compiti più comuni dell'informatica: un elenco ordinato di nomi, di voti o di prezzi è molto più facile da consultare, da confrontare e da leggere di uno non ordinato. In questo capitolo enunciamo con precisione il problema dell'ordinamento, introduciamo i criteri con cui si confrontano gli algoritmi di ordinamento e studiamo tre algoritmi classici che lo risolvono: **Insertion Sort**, **Exchange Sort** e **Selection Sort**.

- I tre algoritmi sono scritti nello pseudocodice del corso, con gli array indicizzati a partire da 1, e poi implementati in C++, con gli array indicizzati a partire da 0.
- Per ogni algoritmo mostriamo l'esecuzione passo per passo sugli stessi due array, $[5, 2, 4, 1, 3]$ e $[12, 11, 13, 5, 6]$, in modo che i tre metodi possano essere confrontati direttamente.
- Per ogni algoritmo contiamo i confronti tra chiavi e gli spostamenti di chiavi che esegue, e stabiliamo se è in-place e se è stabile.

3.1 Il problema dell'ordinamento

Definizione 3.1: problema dell'ordinamento

Dato un array $a[1..n]$ di $n \in \mathbb{Z}_{\geq 1}$ numeri naturali, il **problema dell'ordinamento** consiste nel riarrangiare gli elementi dell'array in **ordine non decrescente**, cioè in modo che

$$a[1] \leq a[2] \leq \dots \leq a[n]. \quad (3.1)$$

Un **riarrangiamento** (o **permutazione**) dell'array è un array che contiene gli stessi elementi, ciascuno con la stessa molteplicità, in un ordine eventualmente diverso. L'output del problema dell'ordinamento è il riarrangiamento dell'array di input che soddisfa (3.1).

- Ogni elemento $a[i]$ è detto **chiave**: è il valore su cui si basa l'ordinamento, cioè il valore che viene confrontato per decidere la disposizione finale.
- L'ordine è *non decrescente* e non *crescente* perché l'array può contenere chiavi uguali, come in $[3, 1, 3]$, il cui riarrangiamento ordinato è $[1, 3, 3]$.

- L'output deve essere un riarrangiamento dell'input: un array come $[0, 0, 0, 0, 0]$ soddisfa (3.1), ma non è la versione ordinata di $[5, 2, 4, 1, 3]$.

Esempio 3.1: ordinamento di un array

- Array di input: $a = [5, 2, 4, 1, 3]$.
- Array ordinato: $a = [1, 2, 3, 4, 5]$.

Prima di studiare algoritmi specifici, fissiamo le operazioni che essi possono usare e i criteri con cui li confrontiamo.

3.1.1 Ordinamento basato su confronti

Definizione 3.2: ordinamento basato su confronti

Un algoritmo di ordinamento è **basato su confronti** se accede alle chiavi soltanto confrontandole a due a due e copiandole.

- Le sole operazioni consentite sulle chiavi in un algoritmo di ordinamento basato su confronti sono:
 1. **confrontare due chiavi**: stabilire quale tra $a[i]$ e $a[j]$ è la maggiore (o se sono uguali);
 2. **copiare / assegnare una chiave**: spostare o memorizzare il valore di una chiave, ad esempio $tmp \leftarrow a[i]$.
- Operazioni vietate sulle chiavi:
 1. operazioni aritmetiche sulle chiavi (addizione, moltiplicazione, ecc.);
 2. ispezionare le singole cifre o i singoli bit di una chiave;
 3. qualunque operazione che sfrutti la rappresentazione interna della chiave.
- Tutti gli algoritmi di ordinamento studiati in questo capitolo (Insertion Sort, Exchange Sort, Selection Sort) appartengono a questa classe: decidono come riarrangiare gli elementi soltanto in base ai confronti tra coppie di chiavi.

3.1.2 Metriche di prestazione

- Poiché i confronti e le assegnazioni sono le sole operazioni consentite sulle chiavi, valutiamo un algoritmo di ordinamento basato su confronti contando:
 - il **numero di confronti tra chiavi** che esegue;
 - il **numero di assegnazioni di chiavi** (copie) che esegue.
- Il numero di assegnazioni conta nella pratica. Ogni **scambio** (swap) di due elementi dell'array richiede tre assegnazioni:

$$tmp \leftarrow a[i]; \quad a[i] \leftarrow a[j]; \quad a[j] \leftarrow tmp,$$

quindi gli algoritmi che scambiano meno tendono a essere più veloci.

- Entrambi i numeri dipendono non solo da n ma anche dall'ordine iniziale delle chiavi. Per questo motivo li calcoliamo nel **caso migliore** (l'input che li minimizza) e nel **caso peggiore** (l'input che li massimizza), tra tutti gli array di n elementi.
- Nei conteggi usiamo la somma dei primi $n - 1$ interi positivi:

$$\sum_{k=1}^{n-1} k = 1 + 2 + \dots + (n-1) = \frac{n(n-1)}{2}. \quad (3.2)$$

3.1.3 Ordinamento in-place e uso della memoria

- Oltre al tempo, ci interessa anche quanta **memoria aggiuntiva** usa un algoritmo, oltre allo spazio già necessario per memorizzare le n chiavi di input.

Definizione 3.3: ordinamento in-place

Un algoritmo di ordinamento è **in-place** (in loco) se la quantità di memoria aggiuntiva che richiede è **costante**, cioè non cresce con n .

- In modo equivalente, un algoritmo di ordinamento in-place ordina riarrangiando gli elementi *all'interno* dell'array originale, usando al più un numero fisso di variabili ausiliarie (ad esempio, i contatori dei cicli e una variabile temporanea per lo scambio).
- Gli algoritmi in-place riarrangiano tipicamente l'array di input spostando o scambiando elementi, quindi non richiedono spazio aggiuntivo per un altro array che contenga l'output ordinato.

3.1.4 Stabilità

- In molte applicazioni le chiavi fanno parte di **record** più grandi (ad esempio, uno studente identificato da un voto e da un nome), e due record diversi possono avere la stessa chiave. Scriviamo un record come una coppia (*key*, *label*). Si confrontano solo le chiavi, e ogni etichetta viaggia insieme alla sua chiave.

Definizione 3.4: ordinamento stabile

Un algoritmo di ordinamento è **stabile** se i record con chiavi uguali compaiono nell'output nello stesso ordine relativo in cui compaiono nell'input.

Esempio 3.2: ordinamento stabile e non stabile

- Array di record: $a = [(3, A), (1, B), (3, C), (2, D), (1, E)]$.
 - Ordinamento stabile: $a = [(1, B), (1, E), (2, D), (3, A), (3, C)]$. I record con chiave 1 mantengono l'ordine B, E e i record con chiave 3 mantengono l'ordine A, C , come nell'input.
 - Ordinamento non stabile: $a = [(1, E), (1, B), (2, D), (3, C), (3, A)]$. Le chiavi sono ordinate, ma entrambe le coppie di record con chiavi uguali risultano invertite.
- La stabilità conta quando i record vengono ordinati più volte rispetto a chiavi diverse: ad esempio, ordinando gli studenti prima per nome e poi, con un algoritmo stabile, per voto, gli studenti con lo stesso voto risultano in ordine alfabetico.

Un algoritmo di ordinamento basato su confronti si valuta in base al **numero di confronti tra chiavi** e al **numero di assegnazioni di chiavi** che esegue (nel caso migliore e nel caso peggiore), alla **memoria aggiuntiva** che usa (in-place o no) e al fatto di essere **stabile**.

3.2 Insertion Sort

- Insertion Sort funziona come molte persone ordinano una mano di carte da gioco: le carte già in mano si tengono ordinate, e ogni nuova carta viene inserita al suo posto tra di esse.
- All'inizio dell'iterazione i , il sotto-array $a[1..i-1]$ è ordinato. L'algoritmo prende l'elemento successivo $a[i]$, detto *key*, sposta di una posizione a destra tutti gli elementi di $a[1..i-1]$ maggiori di *key* e scrive *key* nella posizione rimasta libera (Figura 3.1).
- Questo è lo pseudocodice dell'algoritmo Insertion Sort, che ordina un array $a[1..n]$ di n numeri naturali in ordine non decrescente.

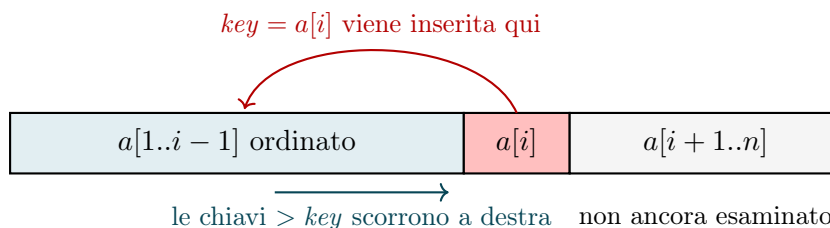


Figura 3.1: Iterazione i di Insertion Sort: $key = a[i]$ viene inserita nel sotto-array ordinato $a[1..i-1]$, dopo aver spostato di una posizione a destra le chiavi maggiori di key .

Algorithm: InsertionSort

Input : Un array $a[1..n]$ di n numeri naturali

Output: L'array a ordinato in ordine non decrescente

```

1 function InsertionSort( $a, n$ )
2    $i := 2$ 
3   while  $i \leq n$  do
4      $key := a[i]$                                 // elemento da inserire
5      $j := i - 1$ 
6     while  $j \geq 1$  and  $a[j] > key$  do
7        $a[j + 1] \leftarrow a[j]$                     // sposta l'elemento a destra
8        $j \leftarrow j - 1$ 
9     end
10     $a[j + 1] \leftarrow key$                         // inserisce key nella sua posizione
11     $i \leftarrow i + 1$ 
12  end
13 end

```

- Spiegazione dell'algoritmo:

- Il ciclo esterno (righe 3–11) scorre l'array a a partire dal secondo elemento ($i = 2$): un singolo elemento, $a[1..1]$, è già ordinato.
- Per ogni elemento $a[i]$ (la key), il ciclo interno (righe 6–8) scorre la parte ordinata $a[1..i-1]$ da destra a sinistra e sposta di una posizione a destra tutti gli elementi maggiori di key .
- Il ciclo interno si ferma quando $j = 0$ (tutti gli elementi di $a[1..i-1]$ erano maggiori di key) oppure quando $a[j] \leq key$. La condizione alla riga 6 è valutata da sinistra a destra: quando $j = 0$ il confronto $a[j] > key$ non viene valutato, perché $a[0]$ non esiste.
- La riga 10 scrive key nella posizione $j + 1$ lasciata libera dagli spostamenti.
- Dopo ogni iterazione del ciclo esterno, il sotto-array $a[1..i]$ è ordinato. Quando l'algoritmo termina, $i = n + 1$ e l'intero array $a[1..n]$ è ordinato.

- Nelle tabelle di traccia di questo capitolo, ogni riga mostra lo stato dell'array *dopo* l'operazione descritta nell'ultima colonna. La chiave corrente è evidenziata in **rosso**, e le celle

modificate da uno spostamento, da un inserimento o da uno scambio sono evidenziate in blu.

Esempio 3.3: Insertion Sort su $[5, 2, 4, 1, 3]$

Array di input: $a = [5, 2, 4, 1, 3]$, con $n = 5$. Il ciclo esterno esegue le iterazioni $i = 2, 3, 4, 5$.

- Traccia dettagliata (ciclo interno):

i	j	key	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	Azione
–	–	–	5	2	4	1	3	stato iniziale
2	1	2	5	2	4	1	3	$key \leftarrow a[2] = 2; j \leftarrow 1$
	0	2	5	5	4	1	3	$a[1] = 5 > 2$: sposta $a[2] \leftarrow 5; j \leftarrow 0$
	–	2	2	5	4	1	3	$j = 0$: esci; inserisci $a[1] \leftarrow 2$
3	2	4	2	5	4	1	3	$key \leftarrow a[3] = 4; j \leftarrow 2$
	1	4	2	5	5	1	3	$a[2] = 5 > 4$: sposta $a[3] \leftarrow 5; j \leftarrow 1$
	–	4	2	4	5	1	3	$a[1] = 2 \leq 4$: esci; inserisci $a[2] \leftarrow 4$
4	3	1	2	4	5	1	3	$key \leftarrow a[4] = 1; j \leftarrow 3$
	2	1	2	4	5	5	3	$a[3] = 5 > 1$: sposta $a[4] \leftarrow 5; j \leftarrow 2$
	1	1	2	4	4	5	3	$a[2] = 4 > 1$: sposta $a[3] \leftarrow 4; j \leftarrow 1$
	0	1	2	2	4	5	3	$a[1] = 2 > 1$: sposta $a[2] \leftarrow 2; j \leftarrow 0$
	–	1	1	2	4	5	3	$j = 0$: esci; inserisci $a[1] \leftarrow 1$
5	4	3	1	2	4	5	3	$key \leftarrow a[5] = 3; j \leftarrow 4$
	3	3	1	2	4	5	5	$a[4] = 5 > 3$: sposta $a[5] \leftarrow 5; j \leftarrow 3$
	2	3	1	2	4	4	5	$a[3] = 4 > 3$: sposta $a[4] \leftarrow 4; j \leftarrow 2$
	–	3	1	2	3	4	5	$a[2] = 2 \leq 3$: esci; inserisci $a[3] \leftarrow 3$

- Stato dell'array dopo ogni iterazione del ciclo esterno:

i	key	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	Azione
–	–	5	2	4	1	3	stato iniziale
2	2	2	5	4	1	3	inserisci 2 all'inizio
3	4	2	4	5	1	3	inserisci 4 tra 2 e 5
4	1	1	2	4	5	3	inserisci 1 all'inizio
5	3	1	2	3	4	5	inserisci 3 tra 2 e 4

- L'array ordinato è $[1, 2, 3, 4, 5]$. L'algoritmo esegue 9 confronti tra chiavi e 7 spostamenti.

Esempio 3.4: Insertion Sort su $[12, 11, 13, 5, 6]$

Array di input: $a = [12, 11, 13, 5, 6]$, con $n = 5$.

- Traccia dettagliata (ciclo interno):

i	j	key	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	Azione
–	–	–	12	11	13	5	6	stato iniziale
2	1	11	12	11	13	5	6	$key \leftarrow a[2] = 11; j \leftarrow 1$
	0	11	12	12	13	5	6	$a[1] = 12 > 11$: sposta $a[2] \leftarrow 12; j \leftarrow 0$
	–	11	11	12	13	5	6	$j = 0$: esci; inserisci $a[1] \leftarrow 11$
3	2	13	11	12	13	5	6	$key \leftarrow a[3] = 13; j \leftarrow 2$
	–	13	11	12	13	5	6	$a[2] = 12 \leq 13$: esci; inserisci $a[3] \leftarrow 13$
4	3	5	11	12	13	5	6	$key \leftarrow a[4] = 5; j \leftarrow 3$
	2	5	11	12	13	13	6	$a[3] = 13 > 5$: sposta $a[4] \leftarrow 13; j \leftarrow 2$
	1	5	11	12	12	13	6	$a[2] = 12 > 5$: sposta $a[3] \leftarrow 12; j \leftarrow 1$
	0	5	11	11	12	13	6	$a[1] = 11 > 5$: sposta $a[2] \leftarrow 11; j \leftarrow 0$
	–	5	5	11	12	13	6	$j = 0$: esci; inserisci $a[1] \leftarrow 5$
5	4	6	5	11	12	13	6	$key \leftarrow a[5] = 6; j \leftarrow 4$
	3	6	5	11	12	13	13	$a[4] = 13 > 6$: sposta $a[5] \leftarrow 13; j \leftarrow 3$
	2	6	5	11	12	12	13	$a[3] = 12 > 6$: sposta $a[4] \leftarrow 12; j \leftarrow 2$
	1	6	5	11	11	12	13	$a[2] = 11 > 6$: sposta $a[3] \leftarrow 11; j \leftarrow 1$
	–	6	5	6	11	12	13	$a[1] = 5 \leq 6$: esci; inserisci $a[2] \leftarrow 6$

- Stato dell'array dopo ogni iterazione del ciclo esterno:

i	key	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	Azione
–	–	12	11	13	5	6	stato iniziale
2	11	11	12	13	5	6	inserisci 11 all'inizio
3	13	11	12	13	5	6	13 già al suo posto
4	5	5	11	12	13	6	inserisci 5 all'inizio
5	6	5	6	11	12	13	inserisci 6 tra 5 e 11

- L'array ordinato è $[5, 6, 11, 12, 13]$. Nell'iterazione $i = 3$ la chiave 13 è già al suo posto: il ciclo interno si ferma al primo confronto e non viene eseguito alcuno spostamento. L'algoritmo esegue 9 confronti tra chiavi e 7 spostamenti.

3.2.1 Proprietà di Insertion Sort

- I due esempi mostrano che il lavoro svolto da Insertion Sort dipende dall'ordine iniziale delle chiavi: una chiave già al suo posto costa un confronto e nessuno spostamento, mentre una chiave più piccola di tutte le precedenti viene spostata lungo l'intera parte ordinata.

Osservazione 3.5: confronti e spostamenti di Insertion Sort

Dato un array $a[1..n]$ di n numeri naturali, InsertionSort esegue almeno $n - 1$ e al più $\frac{n(n-1)}{2}$ confronti tra chiavi, e al più $\frac{n(n-1)}{2}$ spostamenti. Le limitazioni inferiori sono raggiunte quando l'array è già ordinato (non viene eseguito alcuno spostamento), quelle superiori quando è ordinato in senso decrescente con chiavi distinte.

Dimostrazione. Consideriamo l'iterazione i del ciclo esterno, con $i \in \{2, 3, \dots, n\}$, e siano s_i e c_i il numero di spostamenti e di confronti tra chiavi che essa esegue. Il ciclo interno parte con $j = i - 1 \geq 1$, quindi $c_i \geq 1$. Ogni spostamento (righe 7–8) segue un confronto $a[j] > key$ vero e diminuisce j di 1; quindi $s_i \leq i - 1$. Il ciclo interno si ferma poi in uno di due modi:

- con $j = 0$, dopo $s_i = i - 1$ spostamenti e senza un ulteriore confronto: $c_i = s_i = i - 1$;
- dopo un confronto $a[j] > key$ falso, con $j \geq 1$: allora $s_i = i - 1 - j \leq i - 2$ e $c_i = s_i + 1 \leq i - 1$.

In entrambi i casi $1 \leq c_i \leq i - 1$ e $0 \leq s_i \leq i - 1$. Sommando su $i \in \{2, 3, \dots, n\}$ e usando (3.2):

$$n - 1 \leq \text{confronti tra chiavi} \leq \sum_{i=2}^n (i - 1) = \frac{n(n - 1)}{2}, \quad \text{spostamenti} \leq \frac{n(n - 1)}{2}.$$

Se l'array è ordinato, a ogni iterazione $a[i - 1] \leq key$: il ciclo interno si ferma al primo confronto, senza spostamenti, e l'algoritmo esegue esattamente $n - 1$ confronti. Se le chiavi sono distinte e in ordine decrescente, $key = a[i]$ è minore di tutte le chiavi di $a[1..i - 1]$: vengono spostate tutte e il ciclo si ferma con $j = 0$, quindi l'iterazione i esegue $i - 1$ confronti e $i - 1$ spostamenti, e i totali sono entrambi $\frac{n(n-1)}{2}$. $\square$

- Ogni iterazione esegue inoltre due assegnazioni di chiavi (righe 4 e 10), quindi il numero di assegnazioni di chiavi è $2(n - 1)$ più il numero di spostamenti: tra $2(n - 1)$ e $2(n - 1) + \frac{n(n-1)}{2}$.
- Insertion Sort non scambia mai due elementi: uno spostamento è una singola assegnazione, il che rende il metodo più economico di una sequenza di scambi che muoverebbe key di una posizione alla volta.

Osservazione 3.6: Insertion Sort è in-place e stabile

InsertionSort è un algoritmo di ordinamento in-place e stabile.

Dimostrazione. Oltre all'array, l'algoritmo usa soltanto le tre variabili i , j e key , indipendentemente da n : è in-place. Una chiave viene spostata a sinistra soltanto oltre chiavi *strettamente* maggiori di essa (condizione $a[j] > key$ alla riga 6): il ciclo interno si ferma alla prima chiave $a[j] \leq key$, quindi key viene inserita dopo ogni chiave uguale che la precede. Dunque due record con chiavi uguali non vengono mai scambiati, e il loro ordine relativo nell'output è lo stesso dell'input: l'algoritmo è stabile. $\square$

3.2.2 Implementazione in C++

- Questa è un'implementazione in C++ dell'algoritmo Insertion Sort. In C++ l'array ha indici da 0 a $n-1$: il ciclo esterno parte da $i = 1$, e il ciclo interno si ferma quando j diventa -1 . L'operatore `&&` valuta il suo operando destro solo quando quello sinistro è vero, quindi `a[j]` non viene mai letto con $j = -1$.

```
1 // Ordina un array in-place in ordine non decrescente
2
3 void InsertionSort(int n, int *a)
4 {
5     int i, j, key;
6
7     for (i = 1; i < n; i++)
8     {
9         key = a[i]; // Elemento da inserire
10        j = i - 1;
11
12        // Sposta a destra gli elementi maggiori di key
13        while (j >= 0 && a[j] > key)
14        {
15            a[j + 1] = a[j];
16            j = j - 1;
17        }
18
19        // Inserisce key nella posizione corretta
20        a[j + 1] = key;
21    }
22 }
```

Esempio 3.5: esecuzione del codice C++ su [5, 2, 4, 1, 3]

La funzione C++ viene chiamata con $n = 5$ e l'array $a = [5, 2, 4, 1, 3]$, memorizzato in $a[0..4]$. L'esecuzione è quella dell'Esempio 3.3, con tutti gli indici diminuiti di 1:

i	j	key	$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	Azione
–	–	–	5	2	4	1	3	stato iniziale
1	0	2	5	2	4	1	3	$key \leftarrow a[1] = 2; j \leftarrow 0$
	-1	2	5	5	4	1	3	$a[0] = 5 > 2$: sposta $a[1] \leftarrow 5; j \leftarrow -1$
	–	2	2	5	4	1	3	$j = -1$: esci; inserisci $a[0] \leftarrow 2$
2	1	4	2	5	4	1	3	$key \leftarrow a[2] = 4; j \leftarrow 1$
	0	4	2	5	5	1	3	$a[1] = 5 > 4$: sposta $a[2] \leftarrow 5; j \leftarrow 0$
	–	4	2	4	5	1	3	$a[0] = 2 \leq 4$: esci; inserisci $a[1] \leftarrow 4$
3	2	1	2	4	5	1	3	$key \leftarrow a[3] = 1; j \leftarrow 2$
	1	1	2	4	5	5	3	$a[2] = 5 > 1$: sposta $a[3] \leftarrow 5; j \leftarrow 1$
	0	1	2	4	4	5	3	$a[1] = 4 > 1$: sposta $a[2] \leftarrow 4; j \leftarrow 0$
	-1	1	2	2	4	5	3	$a[0] = 2 > 1$: sposta $a[1] \leftarrow 2; j \leftarrow -1$
	–	1	1	2	4	5	3	$j = -1$: esci; inserisci $a[0] \leftarrow 1$
4	3	3	1	2	4	5	3	$key \leftarrow a[4] = 3; j \leftarrow 3$
	2	3	1	2	4	5	5	$a[3] = 5 > 3$: sposta $a[4] \leftarrow 5; j \leftarrow 2$
	1	3	1	2	4	4	5	$a[2] = 4 > 3$: sposta $a[3] \leftarrow 4; j \leftarrow 1$
	–	3	1	2	3	4	5	$a[1] = 2 \leq 3$: esci; inserisci $a[2] \leftarrow 3$

La funzione termina con l'array ordinato $[1, 2, 3, 4, 5]$.

Esempio 3.6: esecuzione del codice C++ su $[12, 11, 13, 5, 6]$

La funzione C++ viene chiamata con $n = 5$ e l'array $a = [12, 11, 13, 5, 6]$, memorizzato in $a[0..4]$:

i	j	key	$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	Azione
–	–	–	12	11	13	5	6	stato iniziale
1	0	11	12	11	13	5	6	$key \leftarrow a[1] = 11; j \leftarrow 0$
	-1	11	12	12	13	5	6	$a[0] = 12 > 11$: sposta $a[1] \leftarrow 12; j \leftarrow -1$
	–	11	11	12	13	5	6	$j = -1$: esci; inserisci $a[0] \leftarrow 11$
2	1	13	11	12	13	5	6	$key \leftarrow a[2] = 13; j \leftarrow 1$
	–	13	11	12	13	5	6	$a[1] = 12 \leq 13$: esci; inserisci $a[2] \leftarrow 13$
3	2	5	11	12	13	5	6	$key \leftarrow a[3] = 5; j \leftarrow 2$
	1	5	11	12	13	13	6	$a[2] = 13 > 5$: sposta $a[3] \leftarrow 13; j \leftarrow 1$
	0	5	11	12	12	13	6	$a[1] = 12 > 5$: sposta $a[2] \leftarrow 12; j \leftarrow 0$
	-1	5	11	11	12	13	6	$a[0] = 11 > 5$: sposta $a[1] \leftarrow 11; j \leftarrow -1$
	–	5	5	11	12	13	6	$j = -1$: esci; inserisci $a[0] \leftarrow 5$
4	3	6	5	11	12	13	6	$key \leftarrow a[4] = 6; j \leftarrow 3$
	2	6	5	11	12	13	13	$a[3] = 13 > 6$: sposta $a[4] \leftarrow 13; j \leftarrow 2$
	1	6	5	11	12	12	13	$a[2] = 12 > 6$: sposta $a[3] \leftarrow 12; j \leftarrow 1$
	0	6	5	11	11	12	13	$a[1] = 11 > 6$: sposta $a[2] \leftarrow 11; j \leftarrow 0$
	–	6	5	6	11	12	13	$a[0] = 5 \leq 6$: esci; inserisci $a[1] \leftarrow 6$

La funzione termina con l'array ordinato $[5, 6, 11, 12, 13]$.

3.3 Exchange Sort

- Insertion Sort costruisce il prefisso ordinato inserendo ogni nuova chiave tra le precedenti. Exchange Sort segue l'idea opposta: riempie le posizioni $1, 2, \dots, n - 1$ una alla volta, ciascuna con la più piccola tra le chiavi rimaste.
- Nell'iterazione i , l'algoritmo confronta $a[i]$ con ogni elemento $a[j]$ con $j > i$ e scambia i due elementi ogni volta che $a[i] > a[j]$. Alla fine dell'iterazione $a[i]$ è il minimo di $a[i..n]$ (Figura 3.2).

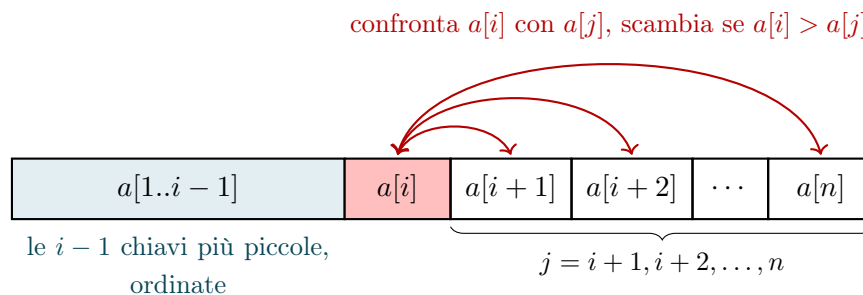


Figura 3.2: Iterazione i di Exchange Sort: $a[i]$ viene confrontato con ogni $a[j]$, $j > i$, e i due elementi vengono scambiati ogni volta che $a[i] > a[j]$; alla fine, $a[i] = \min(a[i..n])$.

- Questo è lo pseudocodice dell'algoritmo Exchange Sort, che ordina un array $a[1..n]$ di n numeri naturali in ordine non decrescente.

Algorithm: ExchangeSort

Input : Un array $a[1..n]$ di n numeri naturali

Output: L'array a ordinato in ordine non decrescente

```

1 function ExchangeSort( $a, n$ )
2    $i := 1$ 
3   while  $i \leq n - 1$  do
4      $j := i + 1$ 
5     while  $j \leq n$  do
6       if  $a[i] > a[j]$  then
7          $tmp := a[i]$                                 // scambia  $a[i]$  e  $a[j]$ 
8          $a[i] \leftarrow a[j]$ 
9          $a[j] \leftarrow tmp$ 
10      end
11       $j \leftarrow j + 1$ 
12    end
13     $i \leftarrow i + 1$ 
14  end
15 end

```

- Spiegazione dell'algoritmo:

- Il ciclo esterno scorre con l'indice i da 1 a $n - 1$.
- Per ogni i , il ciclo interno confronta $a[i]$ con ogni elemento $a[j]$ per $j \in \{i + 1, i + 2, \dots, n\}$.
- Ogni volta che $a[i] > a[j]$, i due elementi vengono scambiati (righe 7–9), quindi $a[i]$ contiene sempre la più piccola chiave incontrata finora in $a[i..n]$.
- Dopo l'iterazione i del ciclo esterno, $a[i]$ è il minimo di $a[i..n]$ e il sotto-array $a[1..i]$ contiene le i chiavi più piccole dell'array, ordinate. Dopo l'iterazione $n - 1$, l'ultimo elemento $a[n]$ è la chiave maggiore, e l'intero array è ordinato.

Esempio 3.7: Exchange Sort su $[5, 2, 4, 1, 3]$

Array di input: $a = [5, 2, 4, 1, 3]$, con $n = 5$. Il ciclo esterno esegue le iterazioni $i = 1, 2, 3, 4$.

- Traccia dettagliata (ciclo interno):

i	j	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	Azione
–	–	5	2	4	1	3	stato iniziale
1	2	2	5	4	1	3	$a[1] = 5 > a[2] = 2$: scambia $a[1] \leftrightarrow a[2]$
	3	2	5	4	1	3	$a[1] = 2 \leq a[3] = 4$: nessuno scambio
	4	1	5	4	2	3	$a[1] = 2 > a[4] = 1$: scambia $a[1] \leftrightarrow a[4]$
	5	1	5	4	2	3	$a[1] = 1 \leq a[5] = 3$: nessuno scambio
2	3	1	4	5	2	3	$a[2] = 5 > a[3] = 4$: scambia $a[2] \leftrightarrow a[3]$
	4	1	2	5	4	3	$a[2] = 4 > a[4] = 2$: scambia $a[2] \leftrightarrow a[4]$
	5	1	2	5	4	3	$a[2] = 2 \leq a[5] = 3$: nessuno scambio
3	4	1	2	4	5	3	$a[3] = 5 > a[4] = 4$: scambia $a[3] \leftrightarrow a[4]$
	5	1	2	3	5	4	$a[3] = 4 > a[5] = 3$: scambia $a[3] \leftrightarrow a[5]$
4	5	1	2	3	4	5	$a[4] = 5 > a[5] = 4$: scambia $a[4] \leftrightarrow a[5]$

- Stato dell'array dopo ogni iterazione del ciclo esterno:

i	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	Azione
–	5	2	4	1	3	stato iniziale
1	1	5	4	2	3	$a[1] = \min(a[1..5]) = 1$ (2 scambi)
2	1	2	5	4	3	$a[2] = \min(a[2..5]) = 2$ (2 scambi)
3	1	2	3	5	4	$a[3] = \min(a[3..5]) = 3$ (2 scambi)
4	1	2	3	4	5	$a[4] = \min(a[4..5]) = 4$ (1 scambio)

- L'array ordinato è $[1, 2, 3, 4, 5]$. L'algoritmo esegue 10 confronti tra chiavi e 7 scambi.

Esempio 3.8: Exchange Sort su $[12, 11, 13, 5, 6]$

Array di input: $a = [12, 11, 13, 5, 6]$, con $n = 5$.

- Traccia dettagliata (ciclo interno):

i	j	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	Azione
–	–	12	11	13	5	6	stato iniziale
1	2	11	12	13	5	6	$a[1] = 12 > a[2] = 11$: scambia $a[1] \leftrightarrow a[2]$
	3	11	12	13	5	6	$a[1] = 11 \leq a[3] = 13$: nessuno scambio
	4	5	12	13	11	6	$a[1] = 11 > a[4] = 5$: scambia $a[1] \leftrightarrow a[4]$
	5	5	12	13	11	6	$a[1] = 5 \leq a[5] = 6$: nessuno scambio
2	3	5	12	13	11	6	$a[2] = 12 \leq a[3] = 13$: nessuno scambio
	4	5	11	13	12	6	$a[2] = 12 > a[4] = 11$: scambia $a[2] \leftrightarrow a[4]$
	5	5	6	13	12	11	$a[2] = 11 > a[5] = 6$: scambia $a[2] \leftrightarrow a[5]$
3	4	5	6	12	13	11	$a[3] = 13 > a[4] = 12$: scambia $a[3] \leftrightarrow a[4]$
	5	5	6	11	13	12	$a[3] = 12 > a[5] = 11$: scambia $a[3] \leftrightarrow a[5]$
4	5	5	6	11	12	13	$a[4] = 13 > a[5] = 12$: scambia $a[4] \leftrightarrow a[5]$

- Stato dell'array dopo ogni iterazione del ciclo esterno:

i	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	Azione
–	12	11	13	5	6	stato iniziale
1	5	12	13	11	6	$a[1] = \min(a[1..5]) = 5$ (2 scambi)
2	5	6	13	12	11	$a[2] = \min(a[2..5]) = 6$ (2 scambi)
3	5	6	11	13	12	$a[3] = \min(a[3..5]) = 11$ (2 scambi)
4	5	6	11	12	13	$a[4] = \min(a[4..5]) = 12$ (1 scambio)

- L'array ordinato è $[5, 6, 11, 12, 13]$. L'algoritmo esegue 10 confronti tra chiavi e 7 scambi.

3.3.1 Proprietà di Exchange Sort

- A differenza di Insertion Sort, Exchange Sort confronta ogni coppia di posizioni $i < j$ esattamente una volta, qualunque sia l'ordine iniziale delle chiavi: solo il numero di scambi dipende dall'input.

Osservazione 3.7: confronti e scambi di Exchange Sort

Dato un array $a[1..n]$ di n numeri naturali, **ExchangeSort** esegue esattamente $\frac{n(n-1)}{2}$ confronti tra chiavi e al più $\frac{n(n-1)}{2}$ scambi. Non viene eseguito alcuno scambio quando l'array è già ordinato, e vengono eseguiti $\frac{n(n-1)}{2}$ scambi quando è ordinato in senso decrescente con chiavi distinte.

Dimostrazione. Nell'iterazione i del ciclo esterno, $i \in \{1, 2, \dots, n-1\}$, il ciclo interno esegue la riga 6 una volta per ogni $j \in \{i+1, i+2, \dots, n\}$, cioè $n-i$ volte. Per (3.2), il numero totale di confronti tra chiavi è

$$\sum_{i=1}^{n-1} (n-i) = (n-1) + (n-2) + \dots + 1 = \frac{n(n-1)}{2}.$$

Ogni confronto provoca al più uno scambio, quindi gli scambi sono al più $\frac{n(n-1)}{2}$.
 Se l'array è ordinato, $a[i] \leq a[j]$ per ogni $i < j$: il primo confronto non provoca alcuno scambio, l'array non cambia, e lo stesso vale per tutti i confronti successivi.
 Se le chiavi sono distinte e in ordine decrescente, sia $b[1..n]$ l'array di input, quindi $b[1] > b[2] > \dots > b[n]$. Nell'iterazione 1, prima del confronto con $j \in \{2, 3, \dots, n\}$ il valore corrente di $a[1]$ è $b[j-1]$ (l'ultima chiave scambiata in posizione 1), che è maggiore di $a[j] = b[j]$: ogni confronto provoca uno scambio, che porta $b[j-1]$ in posizione j . Alla fine dell'iterazione 1, $a[1] = b[n]$ e $a[2..n] = [b[1], b[2], \dots, b[n-1]]$, che è di nuovo in ordine decrescente: ad esempio, $[6, 5, 4, 3, 2, 1]$ diventa $[1, 6, 5, 4, 3, 2]$. Ripetendo il ragionamento su $a[i..n]$ per $i = 2, 3, \dots, n-1$, tutti i $\frac{n(n-1)}{2}$ confronti provocano uno scambio. $\square$

- Su un array in ordine decrescente con chiavi distinte, Exchange Sort esegue $\frac{n(n-1)}{2}$ scambi, cioè $\frac{3n(n-1)}{2}$ assegnazioni di chiavi, mentre Insertion Sort sposta le chiavi con $\frac{n(n-1)}{2}$ spostamenti, di una assegnazione ciascuno.

Osservazione 3.8: Exchange Sort è in-place ma non stabile

ExchangeSort è un algoritmo di ordinamento in-place, ma non è stabile.

Dimostrazione. Oltre all'array, l'algoritmo usa soltanto le tre variabili i , j e tmp : è in-place. Per mostrare che non è stabile basta esibire un input su cui l'ordine relativo di due chiavi uguali cambia. Consideriamo l'array di record $[(2, A), (2, B), (1, C)]$. Per $i = 1$: il confronto con $j = 2$ dà $2 > 2$ falso, quindi nessuno scambio; il confronto con $j = 3$ dà $2 > 1$, quindi $a[1]$ e $a[3]$ vengono scambiati, ottenendo $[(1, C), (2, B), (2, A)]$. Per $i = 2$: $2 > 2$ è falso, quindi nessuno scambio. L'output è $[(1, C), (2, B), (2, A)]$: il record $(2, A)$, che nell'input precedeva $(2, B)$, ora lo segue. $\square$

3.3.2 Implementazione in C++

- Questa è un'implementazione in C++ dell'algoritmo Exchange Sort, con indici da 0 a $n-1$.

```

1 // Ordina un array in-place in ordine non decrescente
2
3 void ExchangeSort(int n, int *a)
4 {
5     for (int i = 0; i < n-1; i++)
6     {
7         for (int j = i+1; j < n; j++)
8         {
9             if (a[i] > a[j])
10            {
11                int tmp = a[i];
12                a[i] = a[j];
13                a[j] = tmp;
14            }
15        }
16    }
17 }
```

Esempio 3.9: esecuzione del codice C++ su [5, 2, 4, 1, 3]

La funzione C++ viene chiamata con $n = 5$ e l'array $a = [5, 2, 4, 1, 3]$, memorizzato in $a[0..4]$. La tabella mostra soltanto i confronti che provocano uno scambio, con lo stato dell'array dopo ogni scambio:

i	j	$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	Azione
–	–	5	2	4	1	3	stato iniziale
0	1	2	5	4	1	3	$a[0] = 5 > a[1] = 2$: scambia $a[0] \leftrightarrow a[1]$
	3	1	5	4	2	3	$a[0] = 2 > a[3] = 1$: scambia $a[0] \leftrightarrow a[3]$
1	2	1	4	5	2	3	$a[1] = 5 > a[2] = 4$: scambia $a[1] \leftrightarrow a[2]$
	3	1	2	5	4	3	$a[1] = 4 > a[3] = 2$: scambia $a[1] \leftrightarrow a[3]$
2	3	1	2	4	5	3	$a[2] = 5 > a[3] = 4$: scambia $a[2] \leftrightarrow a[3]$
	4	1	2	3	5	4	$a[2] = 4 > a[4] = 3$: scambia $a[2] \leftrightarrow a[4]$
3	4	1	2	3	4	5	$a[3] = 5 > a[4] = 4$: scambia $a[3] \leftrightarrow a[4]$

La funzione termina con l'array ordinato [1, 2, 3, 4, 5].

Esempio 3.10: esecuzione del codice C++ su [12, 11, 13, 5, 6]

La funzione C++ viene chiamata con $n = 5$ e l'array $a = [12, 11, 13, 5, 6]$, memorizzato in $a[0..4]$. Anche qui sono mostrati soltanto i confronti che provocano uno scambio:

i	j	$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	Azione
–	–	12	11	13	5	6	stato iniziale
0	1	11	12	13	5	6	$a[0] = 12 > a[1] = 11$: scambia $a[0] \leftrightarrow a[1]$
	3	5	12	13	11	6	$a[0] = 11 > a[3] = 5$: scambia $a[0] \leftrightarrow a[3]$
1	3	5	11	13	12	6	$a[1] = 12 > a[3] = 11$: scambia $a[1] \leftrightarrow a[3]$
	4	5	6	13	12	11	$a[1] = 11 > a[4] = 6$: scambia $a[1] \leftrightarrow a[4]$
2	3	5	6	12	13	11	$a[2] = 13 > a[3] = 12$: scambia $a[2] \leftrightarrow a[3]$
	4	5	6	11	13	12	$a[2] = 12 > a[4] = 11$: scambia $a[2] \leftrightarrow a[4]$
3	4	5	6	11	12	13	$a[3] = 13 > a[4] = 12$: scambia $a[3] \leftrightarrow a[4]$

La funzione termina con l'array ordinato [5, 6, 11, 12, 13].

3.4 Selection Sort

- Exchange Sort può scambiare molte volte nella stessa iterazione solo per portare il minimo di $a[i..n]$ nella posizione i . Selection Sort ottiene lo stesso risultato con al più uno scambio per iterazione: prima *seleziona* la posizione del minimo di $a[i..n]$, e poi lo scambia con $a[i]$ (Figura 3.3).

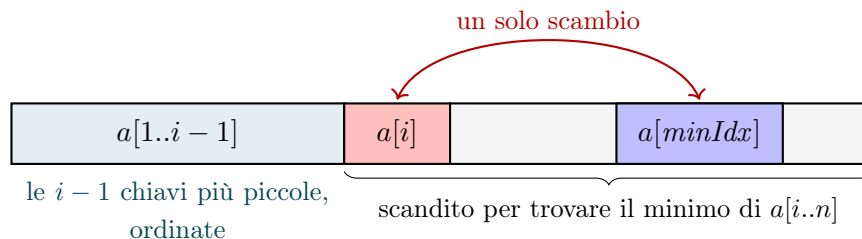


Figura 3.3: Iterazione i di Selection Sort: il ciclo interno trova la posizione $minIdx$ del minimo di $a[i..n]$, e poi $a[i]$ e $a[minIdx]$ vengono scambiati (se $minIdx \neq i$).

- Questo è lo pseudocodice dell'algoritmo Selection Sort, che ordina un array $a[1..n]$ di n numeri naturali in ordine non decrescente.

Algorithm: SelectionSort

Input : Un array $a[1..n]$ di n numeri naturali

Output: L'array a ordinato in ordine non decrescente

```

1 function SelectionSort( $a, n$ )
2    $i := 1$ 
3   while  $i \leq n - 1$  do
4      $minIdx := i$                                 // indice del minimo in  $a[i..n]$ 
5      $j := i + 1$ 
6     while  $j \leq n$  do
7       if  $a[j] < a[minIdx]$  then
8          $minIdx \leftarrow j$                       // trovato un nuovo minimo
9       end
10       $j \leftarrow j + 1$ 
11    end
12    if  $minIdx \neq i$  then
13       $tmp := a[i]$                                 // scambia  $a[i]$  e  $a[minIdx]$ 
14       $a[i] \leftarrow a[minIdx]$ 
15       $a[minIdx] \leftarrow tmp$ 
16    end
17     $i \leftarrow i + 1$ 
18  end
19 end

```

- Spiegazione dell'algoritmo:

- Il ciclo esterno scorre con l'indice i da 1 a $n - 1$.
- Per ogni i , il ciclo interno scandisce $a[i + 1..n]$ e mantiene in $minIdx$ l'indice della più piccola chiave incontrata finora. Poiché il confronto alla riga 7 è stretto, in caso di parità $minIdx$ è l'indice del *primo* minimo.
- Dopo il ciclo interno, se $minIdx \neq i$, il minimo $a[minIdx]$ viene scambiato con $a[i]$: al più **uno scambio** per iterazione del ciclo esterno, a differenza di Exchange Sort.
- Dopo l'iterazione i del ciclo esterno, il sotto-array $a[1..i]$ è ordinato e contiene le i chiavi più piccole dell'array.

Esempio 3.11: Selection Sort su $[5, 2, 4, 1, 3]$

Array di input: $a = [5, 2, 4, 1, 3]$, con $n = 5$. Il ciclo esterno esegue le iterazioni $i = 1, 2, 3, 4$.

- Traccia dettagliata (ciclo interno):

i	j	$minIdx$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	Azione
–	–	–	5	2	4	1	3	stato iniziale
1	–	1	5	2	4	1	3	$minIdx := 1$ ($a[1] = 5$)
	2	2	5	2	4	1	3	$a[2] = 2 < a[1] = 5$: $minIdx \leftarrow 2$
	3	2	5	2	4	1	3	$a[3] = 4 \geq a[2] = 2$: nessun aggiornamento
	4	4	5	2	4	1	3	$a[4] = 1 < a[2] = 2$: $minIdx \leftarrow 4$
	5	4	5	2	4	1	3	$a[5] = 3 \geq a[4] = 1$: nessun aggiornamento
	–	–	1	2	4	5	3	$minIdx = 4 \neq 1$: scambia $a[1] \leftrightarrow a[4]$
2	–	2	1	2	4	5	3	$minIdx := 2$ ($a[2] = 2$)
	3	2	1	2	4	5	3	$a[3] = 4 \geq a[2] = 2$: nessun aggiornamento
	4	2	1	2	4	5	3	$a[4] = 5 \geq a[2] = 2$: nessun aggiornamento
	5	2	1	2	4	5	3	$a[5] = 3 \geq a[2] = 2$: nessun aggiornamento
	–	–	1	2	4	5	3	$minIdx = 2$: nessuno scambio
	–	–	1	2	4	5	3	
3	–	3	1	2	4	5	3	$minIdx := 3$ ($a[3] = 4$)
	4	3	1	2	4	5	3	$a[4] = 5 \geq a[3] = 4$: nessun aggiornamento
	5	5	1	2	4	5	3	$a[5] = 3 < a[3] = 4$: $minIdx \leftarrow 5$
	–	–	1	2	3	5	4	$minIdx = 5 \neq 3$: scambia $a[3] \leftrightarrow a[5]$
	–	–	1	2	3	5	4	
4	–	4	1	2	3	5	4	$minIdx := 4$ ($a[4] = 5$)
	5	5	1	2	3	5	4	$a[5] = 4 < a[4] = 5$: $minIdx \leftarrow 5$
	–	–	1	2	3	4	5	$minIdx = 5 \neq 4$: scambia $a[4] \leftrightarrow a[5]$

- Stato dell'array dopo ogni iterazione del ciclo esterno:

i	$minIdx$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	Azione
–	–	5	2	4	1	3	stato iniziale
1	4	1	2	4	5	3	scambia $a[1] \leftrightarrow a[4]$: $a[1] = \min(a[1..5]) = 1$
2	2	1	2	4	5	3	nessuno scambio: $a[2] = \min(a[2..5]) = 2$
3	5	1	2	3	5	4	scambia $a[3] \leftrightarrow a[5]$: $a[3] = \min(a[3..5]) = 3$
4	5	1	2	3	4	5	scambia $a[4] \leftrightarrow a[5]$: $a[4] = \min(a[4..5]) = 4$

- L'array ordinato è $[1, 2, 3, 4, 5]$. L'algoritmo esegue 10 confronti tra chiavi e 3 scambi.

Esempio 3.12: Selection Sort su [12, 11, 13, 5, 6]

Array di input: $a = [12, 11, 13, 5, 6]$, con $n = 5$.

- Traccia dettagliata (ciclo interno):

i	j	$minIdx$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	Azione
–	–	–	12	11	13	5	6	stato iniziale
1	–	1	12	11	13	5	6	$minIdx := 1$ ($a[1] = 12$)
	2	2	12	11	13	5	6	$a[2] = 11 < a[1] = 12$: $minIdx \leftarrow 2$
	3	2	12	11	13	5	6	$a[3] = 13 \geq a[2] = 11$: nessun aggiornamento
	4	4	12	11	13	5	6	$a[4] = 5 < a[2] = 11$: $minIdx \leftarrow 4$
	5	4	12	11	13	5	6	$a[5] = 6 \geq a[4] = 5$: nessun aggiornamento
	–	–	5	11	13	12	6	$minIdx = 4 \neq 1$: scambia $a[1] \leftrightarrow a[4]$
2	–	2	5	11	13	12	6	$minIdx := 2$ ($a[2] = 11$)
	3	2	5	11	13	12	6	$a[3] = 13 \geq a[2] = 11$: nessun aggiornamento
	4	2	5	11	13	12	6	$a[4] = 12 \geq a[2] = 11$: nessun aggiornamento
	5	5	5	11	13	12	6	$a[5] = 6 < a[2] = 11$: $minIdx \leftarrow 5$
	–	–	5	6	13	12	11	$minIdx = 5 \neq 2$: scambia $a[2] \leftrightarrow a[5]$
3	–	3	5	6	13	12	11	$minIdx := 3$ ($a[3] = 13$)
	4	4	5	6	13	12	11	$a[4] = 12 < a[3] = 13$: $minIdx \leftarrow 4$
	5	5	5	6	13	12	11	$a[5] = 11 < a[4] = 12$: $minIdx \leftarrow 5$
	–	–	5	6	11	12	13	$minIdx = 5 \neq 3$: scambia $a[3] \leftrightarrow a[5]$
4	–	4	5	6	11	12	13	$minIdx := 4$ ($a[4] = 12$)
	5	4	5	6	11	12	13	$a[5] = 13 \geq a[4] = 12$: nessun aggiornamento
	–	–	5	6	11	12	13	$minIdx = 4$: nessuno scambio

- Stato dell'array dopo ogni iterazione del ciclo esterno:

i	$minIdx$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	Azione
–	–	12	11	13	5	6	stato iniziale
1	4	5	11	13	12	6	scambia $a[1] \leftrightarrow a[4]$: $a[1] = \min(a[1..5]) = 5$
2	5	5	6	13	12	11	scambia $a[2] \leftrightarrow a[5]$: $a[2] = \min(a[2..5]) = 6$
3	5	5	6	11	12	13	scambia $a[3] \leftrightarrow a[5]$: $a[3] = \min(a[3..5]) = 11$
4	4	5	6	11	12	13	nessuno scambio: $a[4] = \min(a[4..5]) = 12$

- L'array ordinato è [5, 6, 11, 12, 13]. L'algoritmo esegue 10 confronti tra chiavi e 3 scambi.

3.4.1 Proprietà di Selection Sort

Osservazione 3.9: confronti e scambi di Selection Sort

Dato un array $a[1..n]$ di n numeri naturali, **SelectionSort** esegue esattamente $\frac{n(n-1)}{2}$ confronti tra chiavi e al più $n - 1$ scambi. Non viene eseguito alcuno scambio quando l'array è già ordinato, e vengono eseguiti $n - 1$ scambi, ad esempio, sull'array $[2, 3, \dots, n, 1]$.

Dimostrazione. Come per Exchange Sort, nell'iterazione $i \in \{1, 2, \dots, n-1\}$ il confronto alla riga 7 viene eseguito una volta per ogni $j \in \{i+1, i+2, \dots, n\}$, quindi il numero totale di confronti tra chiavi è $\sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2}$ per (3.2). Ciascuna delle $n-1$ iterazioni del ciclo esterno esegue al più uno scambio (righe 13–15), quindi gli scambi sono al più $n-1$.

Se l'array è ordinato, il minimo di $a[i..n]$ è $a[i]$ e, per il confronto stretto alla riga 7, $minIdx$ resta uguale a i : non viene eseguito alcuno scambio. Su $[2, 3, \dots, n, 1]$, l'iterazione 1 trova il minimo 1 in posizione n e lo scambia con $a[1] = 2$, ottenendo $[1, 3, 4, \dots, n, 2]$; in generale, all'inizio dell'iterazione i l'array è $[1, 2, \dots, i-1, i+1, \dots, n, i]$, il minimo i di $a[i..n]$ è in posizione $n \neq i$, e lo scambio produce $[1, 2, \dots, i, i+2, \dots, n, i+1]$. Dunque ciascuna delle $n-1$ iterazioni esegue uno scambio. $\square$

- Selection Sort esegue al più $3(n-1)$ assegnazioni di chiavi, il minor numero di assegnazioni di chiavi nel caso peggiore tra i tre algoritmi di questo capitolo, mentre il suo numero di confronti, $\frac{n(n-1)}{2}$, non dipende dall'input.

Osservazione 3.10: Selection Sort è in-place ma non stabile

SelectionSort è un algoritmo di ordinamento in-place, ma non è stabile.

Dimostrazione. Oltre all'array, l'algoritmo usa soltanto le quattro variabili i , j , $minIdx$ e tmp : è in-place. Consideriamo l'array di record $[(2, A), (2, B), (1, C)]$. Per $i = 1$, il minimo è $(1, C)$, con $minIdx = 3$, e lo scambio di $a[1]$ e $a[3]$ dà $[(1, C), (2, B), (2, A)]$. Per $i = 2$, il confronto $2 < 2$ è falso, quindi $minIdx = 2$ e non viene eseguito alcuno scambio. Nell'output $(2, A)$ segue $(2, B)$, mentre nell'input lo precedeva: lo scambio ha spostato $(2, A)$ oltre la chiave uguale $(2, B)$. $\square$

3.4.2 Implementazione in C++

- Questa è un'implementazione in C++ dell'algoritmo Selection Sort, con indici da 0 a $n-1$.

```

1 // Ordina un array in-place in ordine non decrescente
2
3 void SelectionSort(int n, int *a)
4 {
5     for (int i = 0; i < n-1; i++)
6     {
7         int minIdx = i;
8         for (int j = i+1; j < n; j++)
9         {
10             if (a[j] < a[minIdx])
11             {
12                 minIdx = j;
13             }
14         }
15         if (minIdx != i)
16         {
17             int tmp = a[i];
18             a[i] = a[minIdx];
19             a[minIdx] = tmp;
20         }
21     }
22 }

```

Esempio 3.13: esecuzione del codice C++ su [5, 2, 4, 1, 3]

La funzione C++ viene chiamata con $n = 5$ e l'array $a = [5, 2, 4, 1, 3]$, memorizzato in $a[0..4]$. L'esecuzione è quella dell'Esempio 3.11, con tutti gli indici diminuiti di 1:

i	j	$minIdx$	$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	Azione
—	—	—	5	2	4	1	3	stato iniziale
0	—	0	5	2	4	1	3	$minIdx := 0$ ($a[0] = 5$)
	1	1	5	2	4	1	3	$a[1] = 2 < a[0] = 5$: $minIdx \leftarrow 1$
	2	1	5	2	4	1	3	$a[2] = 4 \geq a[1] = 2$: nessun aggiornamento
	3	3	5	2	4	1	3	$a[3] = 1 < a[1] = 2$: $minIdx \leftarrow 3$
	4	3	5	2	4	1	3	$a[4] = 3 \geq a[3] = 1$: nessun aggiornamento
	—	—	1	2	4	5	3	$minIdx = 3 \neq 0$: scambia $a[0] \leftrightarrow a[3]$
1	—	1	1	2	4	5	3	$minIdx := 1$ ($a[1] = 2$)
	2	1	1	2	4	5	3	$a[2] = 4 \geq a[1] = 2$: nessun aggiornamento
	3	1	1	2	4	5	3	$a[3] = 5 \geq a[1] = 2$: nessun aggiornamento
	4	1	1	2	4	5	3	$a[4] = 3 \geq a[1] = 2$: nessun aggiornamento
	—	—	1	2	4	5	3	$minIdx = 1$: nessuno scambio
2	—	2	1	2	4	5	3	$minIdx := 2$ ($a[2] = 4$)
	3	2	1	2	4	5	3	$a[3] = 5 \geq a[2] = 4$: nessun aggiornamento
	4	4	1	2	4	5	3	$a[4] = 3 < a[2] = 4$: $minIdx \leftarrow 4$
	—	—	1	2	3	5	4	$minIdx = 4 \neq 2$: scambia $a[2] \leftrightarrow a[4]$
3	—	3	1	2	3	5	4	$minIdx := 3$ ($a[3] = 5$)
	4	4	1	2	3	5	4	$a[4] = 4 < a[3] = 5$: $minIdx \leftarrow 4$
	—	—	1	2	3	4	5	$minIdx = 4 \neq 3$: scambia $a[3] \leftrightarrow a[4]$

La funzione termina con l'array ordinato $[1, 2, 3, 4, 5]$.

Esempio 3.14: esecuzione del codice C++ su $[12, 11, 13, 5, 6]$

La funzione C++ viene chiamata con $n = 5$ e l'array $a = [12, 11, 13, 5, 6]$, memorizzato in $a[0..4]$. Qui mostriamo soltanto lo stato dell'array dopo ogni iterazione del ciclo esterno:

i	$minIdx$	$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	Azione
—	—	12	11	13	5	6	stato iniziale
0	3	5	11	13	12	6	scambia $a[0] \leftrightarrow a[3]$: $a[0] = \min(a[0..4]) = 5$
1	4	5	6	13	12	11	scambia $a[1] \leftrightarrow a[4]$: $a[1] = \min(a[1..4]) = 6$
2	4	5	6	11	12	13	scambia $a[2] \leftrightarrow a[4]$: $a[2] = \min(a[2..4]) = 11$
3	3	5	6	11	12	13	nessuno scambio: $a[3] = \min(a[3..4]) = 12$

La funzione termina con l'array ordinato $[5, 6, 11, 12, 13]$.

3.5 Confronto dei tre algoritmi

- La Tabella 3.1 raccoglie i risultati delle sei Osservazioni di questo capitolo, per un array di n chiavi. Gli “spostamenti” sono gli spostamenti (shift) per Insertion Sort e gli scambi (swap) per gli altri due algoritmi.

algoritmo	confronti tra chiavi		spostamenti		in-place	stabile
	migliore	peggiore	migliore	peggiore		
Insertion Sort	$n - 1$	$\frac{n(n-1)}{2}$	0 shift	$\frac{n(n-1)}{2}$ shift	sì	sì
Exchange Sort	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$	0 scambi	$\frac{n(n-1)}{2}$ scambi	sì	no
Selection Sort	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$	0 scambi	$n - 1$ scambi	sì	no

Tabella 3.1: Confronti tra chiavi, spostamenti di chiavi, memoria e stabilità dei tre algoritmi di ordinamento del capitolo, per un array di n chiavi.

Esempio 3.15: i tre algoritmi sugli stessi array

Sui due array usati in tutto il capitolo, i tre algoritmi eseguono i seguenti numeri di confronti tra chiavi e di spostamenti.

algoritmo	[5, 2, 4, 1, 3]		[12, 11, 13, 5, 6]	
	confronti	spostamenti	confronti	spostamenti
Insertion Sort	9	7 shift	9	7 shift
Exchange Sort	10	7 scambi	10	7 scambi
Selection Sort	10	3 scambi	10	3 scambi

- Insertion Sort, Exchange Sort e Selection Sort sono basati su confronti e in-place, e nel caso peggiore tutti e tre eseguono $\frac{n(n-1)}{2}$ confronti tra chiavi.
- Insertion Sort si adatta all'input: su un array già ordinato esegue soltanto $n - 1$ confronti e nessuno spostamento. È l'unico algoritmo stabile dei tre.
- Exchange Sort e Selection Sort eseguono sempre $\frac{n(n-1)}{2}$ confronti, ma Selection Sort esegue al più $n - 1$ scambi, contro i $\frac{n(n-1)}{2}$ scambi di Exchange Sort nel caso peggiore.

Ricerca lineare e binaria

4.1 Il problema della ricerca

- Cercare un valore in una collezione di dati è una delle operazioni più frequenti eseguite da un calcolatore: un nome in una rubrica, un codice in una tabella di prodotti, una parola in un dizionario.
- In questo capitolo la collezione è un array di numeri naturali, e presentiamo due algoritmi che risolvono il problema:
 - la **ricerca lineare**, che funziona su ogni array;
 - la **ricerca binaria**, che funziona solo su array ordinati in modo non decrescente, ma che è molto più veloce della ricerca lineare su array lunghi.

Definizione 4.1: problema della ricerca

Dati un array $a[1..n]$ di n numeri naturali e un numero naturale m da cercare, il **problema della ricerca** consiste nel determinare se m è uno degli elementi dell'array, cioè se esiste un indice $k \in \{1, 2, \dots, n\}$ tale che

$$a[k] = m.$$

Un algoritmo per il problema della ricerca restituisce tale indice k se esiste, e il valore -1 altrimenti.

- Il valore -1 non è mai un indice valido dell'array $a[1..n]$ (né di un array C++, i cui indici partono da 0): per questo motivo può essere usato per segnalare che il valore cercato non è stato trovato.

4.2 Ricerca lineare

Dati un array $a[1..n]$ di n numeri naturali e un numero naturale m da cercare, consideriamo il problema di determinare se m è uno degli elementi dell'array.

- L'idea più semplice è confrontare il valore cercato m con gli elementi dell'array uno dopo l'altro, dal primo all'ultimo, e fermarsi non appena si trova un elemento uguale a m .
- Questo è lo pseudocodice dell'algoritmo di ricerca lineare (Linear Search) che determina se un valore m è presente in un array $a[1..n]$ di n numeri naturali:

Algorithm: LinearSearch

Input : Un array $a[1..n]$ di n numeri naturali e un valore m da cercare

Output: Indice di m in a , oppure -1 se non trovato

```

1 function LinearSearch( $a, n, m$ )
2    $i := 1$ 
3   while  $i \leq n$  do
4     if  $a[i] = m$  then
5       return  $i$                                 // Elemento trovato all'indice  $i$ 
6     end
7      $i \leftarrow i + 1$ 
8   end
9   return  $-1$                                     // Elemento non trovato
10 end

```

- Spiegazione dell'algoritmo:
 - L'algoritmo scorre l'array sequenzialmente dall'indice 1 a n .
 - A ogni iterazione confronta l'elemento $a[i]$ con il valore cercato m (riga 4).
 - Se $a[i] = m$, l'elemento è trovato e viene restituito l'indice i (riga 5).
 - Se il ciclo termina senza trovare corrispondenze, l'elemento non è nell'array e viene restituito -1 (riga 9).
- Se m compare più di una volta nell'array, l'algoritmo si ferma alla prima corrispondenza: l'indice restituito è il **più piccolo** indice k tale che $a[k] = m$.

Esempio 4.1: Ricerca lineare

Consideriamo l'array $a = [4, 2, 7, 1, 5, 9, 3]$ con $n = 7$.

- Ricerca di $m = 5$:

iterazione	i	$a[i]$	azione
1	1	4	$a[i] \neq m$
2	2	2	$a[i] \neq m$
3	3	7	$a[i] \neq m$
4	4	1	$a[i] \neq m$
5	5	5	$a[i] = m \Rightarrow$ Trovato!

Quindi, l'elemento 5 si trova all'indice 5.

- Ricerca di $m = 6$:

iterazione	i	$a[i]$	azione
1	1	4	$a[i] \neq m$
2	2	2	$a[i] \neq m$
3	3	7	$a[i] \neq m$
4	4	1	$a[i] \neq m$
5	5	5	$a[i] \neq m$
6	6	9	$a[i] \neq m$
7	7	3	$a[i] \neq m$
—	8	—	$i > n \Rightarrow$ il ciclo termina, non trovato!

Quindi, l'elemento 6 non è presente nell'array (restituisce -1).

- Il lavoro svolto dalla ricerca lineare si misura con il numero di confronti $a[i] = m$ alla riga 4: nell'esempio servono 5 confronti per trovare $m = 5$, e tutti gli $n = 7$ elementi vengono confrontati per concludere che $m = 6$ non è nell'array.

Osservazione 4.2: numero di confronti della ricerca lineare

Dati un array $a[1..n]$ di n numeri naturali e un valore m da cercare:

- se k è il più piccolo indice tale che $a[k] = m$, allora **LinearSearch** esegue esattamente k confronti alla riga 4 e restituisce k ;
- se m non è nell'array, allora **LinearSearch** esegue esattamente n confronti alla riga 4 e restituisce -1 .

Dimostrazione. La variabile i assume i valori $1, 2, 3, \dots$ in quest'ordine, e per ogni valore $i \leq n$ viene eseguito esattamente un confronto $a[i] = m$.

- Se k è il più piccolo indice con $a[k] = m$, allora $a[i] \neq m$ per $i \in \{1, 2, \dots, k-1\}$, quindi l'algoritmo non si ferma per questi valori di i ; per $i = k$ il confronto è vero e l'algoritmo restituisce k . I confronti sono quelli con $i = 1, 2, \dots, k$, cioè k confronti.
- Se m non è nell'array, ogni confronto è falso, quindi il ciclo viene eseguito per $i = 1, 2, \dots, n$ e si ferma quando $i = n + 1$. I confronti sono n , e l'algoritmo

restituisce -1 alla riga 9.



La ricerca lineare non fa alcuna ipotesi sull'ordine degli elementi. Il suo prezzo è che, quando il valore cercato è assente (o si trova nell'ultima posizione), tutti gli n elementi dell'array devono essere esaminati.

4.2.1 Implementazione in C++

- Questa è un'implementazione in C++ dell'algoritmo di ricerca lineare. In C++ gli indici di un array di n elementi vanno da 0 a $n - 1$: l'indice restituito dalla funzione è quindi l'indice dello pseudocodice meno 1.

```

1 // Restituisce l'indice di m nell'array a, oppure -1 se m non c'è
2
3 int LinearSearch(int n, int *a, int m)
4 {
5     for (int i = 0; i < n; i++)
6     {
7         if (a[i] == m)
8         {
9             return i; // Elemento trovato all'indice i
10        }
11    }
12
13    return -1; // Elemento non trovato
14 }
```

- La funzione può essere chiamata, per esempio, dal seguente main:

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int a[] = {4, 2, 7, 1, 5, 9, 3};
7     int n = 7;
8
9     cout << LinearSearch(n, a, 5) << endl; // stampa 4
10    cout << LinearSearch(n, a, 6) << endl; // stampa -1
11
12    return 0;
13 }
```

Esempio 4.2: esecuzione dell'implementazione in C++ della ricerca lineare

Consideriamo l'array dell'Esempio 4.1, memorizzato in C++ con indici da 0 a 6:

$a[j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$	$j = 6$
	4	2	7	1	5	9	3

- Ricerca di $m = 5$:

iterazione	i	confronto	azione
1	0	$a[0] = 4 \neq 5$	continua
2	1	$a[1] = 2 \neq 5$	continua
3	2	$a[2] = 7 \neq 5$	continua
4	3	$a[3] = 1 \neq 5$	continua
5	4	$a[4] = 5 = m$	Trovato! Restituisce 4

La funzione restituisce 4: l'elemento si trova all'indice 4 di C++, cioè all'indice 5 dello pseudocodice.

- Ricerca di $m = 6$:

iterazione	i	confronto	azione
1	0	$a[0] = 4 \neq 6$	continua
2	1	$a[1] = 2 \neq 6$	continua
3	2	$a[2] = 7 \neq 6$	continua
4	3	$a[3] = 1 \neq 6$	continua
5	4	$a[4] = 5 \neq 6$	continua
6	5	$a[5] = 9 \neq 6$	continua
7	6	$a[6] = 3 \neq 6$	continua

Il ciclo `for` termina con $i = n = 7$: l'elemento non è stato trovato e la funzione restituisce -1 .

4.3 Ricerca binaria

- Pensiamo a come si cerca una parola in un dizionario cartaceo: nessuno lo legge dalla prima pagina. Lo apriamo a metà e, poiché le parole sono in ordine alfabetico, sappiamo in quale metà proseguire. La ricerca binaria applica la stessa idea a un array ordinato.

Definizione 4.3: array ordinato

Un array $a[1..n]$ di n numeri naturali è **ordinato in modo non decrescente** se

$$a[1] \leq a[2] \leq \dots \leq a[n].$$

Dati un array $a[1..n]$ di n numeri naturali ordinato in modo non decrescente e un numero naturale m da cercare, consideriamo il problema di determinare se m è uno degli elementi dell'array.

- La ricerca binaria mantiene una porzione $a[\text{left}..\text{right}]$ dell'array, detta **intervallo di ricerca**, che è l'unica porzione che può ancora contenere m . All'inizio l'intervallo di ricerca è l'intero array.

- A ogni iterazione il valore cercato m viene confrontato con l'elemento $a[mid]$ al centro dell'intervallo di ricerca:
 - se $a[mid] < m$, allora, poiché l'array è ordinato, tutti gli elementi $a[left], \dots, a[mid]$ sono minori di m , e la ricerca prosegue nella metà destra $a[mid + 1..right]$;
 - se $a[mid] > m$, allora tutti gli elementi $a[mid], \dots, a[right]$ sono maggiori di m , e la ricerca prosegue nella metà sinistra $a[left..mid - 1]$.
- Questo è lo pseudocodice dell'algoritmo di ricerca binaria (Binary Search) che determina se un valore m è presente in un array $a[1..n]$ di n numeri naturali ordinato in modo non decrescente:

Algorithm: BinarySearch

Input : Un array $a[1..n]$ di n numeri naturali ordinato in modo non decrescente e un valore m da cercare

Output: Indice di m in a , oppure -1 se non trovato

```

1 function BinarySearch( $a, n, m$ )
2    $left := 1$                                 // Estremo sinistro
3    $right := n$                                 // Estremo destro
4   while  $left \leq right$  do
5      $mid := left + \lfloor (right - left) / 2 \rfloor$  // Indice centrale
6     if  $a[mid] = m$  then
7       return  $mid$                             // Elemento trovato
8     end
9     if  $a[mid] < m$  then
10       $left \leftarrow mid + 1$                   // Cerca nella metà destra
11    end
12    else
13       $right \leftarrow mid - 1$                 // Cerca nella metà sinistra
14    end
15  end
16  return  $-1$                                 // Elemento non trovato
17 end

```

- Spiegazione dell'algoritmo:
 - L'algoritmo mantiene due indici, $left$ e $right$, che definiscono l'intervallo di ricerca $a[left..right]$, il quale contiene $right - left + 1$ elementi.
 - A ogni iterazione calcola l'indice centrale mid (riga 5) e confronta $a[mid]$ con il valore cercato m .
 - Se $a[mid] = m$, l'elemento è trovato e ne viene restituito l'indice (riga 7).
 - Se $a[mid] < m$, la ricerca prosegue nella metà destra aggiornando $left \leftarrow mid + 1$ (riga 10).
 - Se $a[mid] > m$, la ricerca prosegue nella metà sinistra aggiornando $right \leftarrow mid - 1$ (riga 13).

- Se l'intervallo di ricerca diventa vuoto ($left > right$), l'elemento non è nell'array e viene restituito -1 (riga 16).
- Poiché $0 \leq \lfloor (right - left) / 2 \rfloor \leq right - left$, l'indice mid appartiene sempre all'intervallo di ricerca: $left \leq mid \leq right$.

Esempio 4.3: Ricerca binaria

Consideriamo l'array ordinato $a = [3, 9, 11, 12, 12, 16, 19]$ con $n = 7$.

- Ricerca di $m = 12$:

iterazione	$left$	$right$	mid	$a[mid]$	azione
1	1	7	4	12	$a[mid] = m \Rightarrow$ Trovato!

Quindi, l'elemento 12 si trova all'indice 4.

- Ricerca di $m = 10$:

iterazione	$left$	$right$	mid	$a[mid]$	azione
1	1	7	4	12	$a[mid] > m \Rightarrow right \leftarrow 3$
2	1	3	2	9	$a[mid] < m \Rightarrow left \leftarrow 3$
3	3	3	3	11	$a[mid] > m \Rightarrow right \leftarrow 2$
–	3	2	–	–	$left > right \Rightarrow$ Non trovato!

Quindi, l'elemento 10 non è presente nell'array (restituisce -1).

- Ricerca di $m = 3$:

iterazione	$left$	$right$	mid	$a[mid]$	azione
1	1	7	4	12	$a[mid] > m \Rightarrow right \leftarrow 3$
2	1	3	2	9	$a[mid] > m \Rightarrow right \leftarrow 1$
3	1	1	1	3	$a[mid] = m \Rightarrow$ Trovato!

Quindi, l'elemento 3 si trova all'indice 1.

- Ricerca di $m = 19$:

iterazione	$left$	$right$	mid	$a[mid]$	azione
1	1	7	4	12	$a[mid] < m \Rightarrow left \leftarrow 5$
2	5	7	6	16	$a[mid] < m \Rightarrow left \leftarrow 7$
3	7	7	7	19	$a[mid] = m \Rightarrow$ Trovato!

Quindi, l'elemento 19 si trova all'indice 7.

- La figura seguente mostra l'intervallo di ricerca a ogni iterazione della ricerca di $m = 10$: gli elementi fuori dall'intervallo di ricerca (grigi) sono scartati, e l'elemento $a[mid]$ confrontato con m è evidenziato in rosso.

indice	1	2	3	4	5	6	7	
iterazione 1	3	9	11	12	12	16	19	$a[4] = 12 > 10 \Rightarrow right \leftarrow 3$
iterazione 2	3	9	11	12	12	16	19	$a[2] = 9 < 10 \Rightarrow left \leftarrow 3$
iterazione 3	3	9	11	12	12	16	19	$a[3] = 11 > 10 \Rightarrow right \leftarrow 2$
fine del ciclo	3	9	11	12	12	16	19	$left > right \Rightarrow$ restituisce -1

Figura 4.1: Intervalli di ricerca di **BinarySearch** per $m = 10$ in $a = [3, 9, 11, 12, 12, 16, 19]$: le celle blu formano l'intervallo di ricerca $a[left..right]$, la cella rossa è $a[mid]$, le celle grigie sono state scartate.

- Se m compare più di una volta nell'array, la ricerca binaria restituisce l'indice di **una** delle sue occorrenze, non necessariamente la prima. Per esempio, per $a = [1, 2, 2, 2, 2, 2, 3]$ e $m = 2$, la prima iterazione calcola $mid = 1 + \lfloor 6/2 \rfloor = 4$ e, poiché $a[4] = 2 = m$, l'algoritmo restituisce 4, mentre **LinearSearch** restituisce 2.

4.3.1 Numero di iterazioni

- Ogni iterazione che non trova m scarta l'elemento $a[mid]$ e almeno metà dei restanti elementi dell'intervallo di ricerca. Per questo la ricerca binaria è veloce: la lunghezza dell'intervallo di ricerca si (almeno) dimezza a ogni iterazione.

Osservazione 4.4: numero di iterazioni della ricerca binaria

Dati un array $a[1..n]$ di $n \geq 1$ numeri naturali ordinato in modo non decrescente e un valore m da cercare, il ciclo while di **BinarySearch** esegue al più

$$\lfloor \log_2 n \rfloor + 1$$

iterazioni.

Dimostrazione. Sia $L = right - left + 1 \geq 1$ il numero di elementi dell'intervallo di ricerca all'inizio di un'iterazione che non restituisce. Poiché $mid = left + \lfloor (L-1)/2 \rfloor$, il nuovo intervallo di ricerca ha:

- $mid - left = \lfloor (L-1)/2 \rfloor$ elementi, se $right \leftarrow mid - 1$;
- $right - mid = (L-1) - \lfloor (L-1)/2 \rfloor = \lceil (L-1)/2 \rceil$ elementi, se $left \leftarrow mid + 1$.

In entrambi i casi il nuovo numero di elementi è al più $\lceil (L-1)/2 \rceil = \lfloor L/2 \rfloor \leq L/2$.

All'inizio $L = n$. Quindi, dopo k iterazioni che non restituiscono, l'intervallo di ricerca ha al più $n/2^k$ elementi. Un'iterazione viene eseguita solo se l'intervallo di ricerca non è vuoto, quindi l'iterazione $k+1$ viene eseguita solo se

$$1 \leq \frac{n}{2^k}, \quad \text{cioè} \quad k \leq \log_2 n.$$

Poiché k è un intero, $k \leq \lfloor \log_2 n \rfloor$, e il numero $k + 1$ di una qualsiasi iterazione eseguita è al più $\lfloor \log_2 n \rfloor + 1$. $\square$

- Nell'Esempio 4.3, $n = 7$ e $\lfloor \log_2 7 \rfloor + 1 = 3$: le ricerche di $m = 10$, $m = 3$ e $m = 19$ eseguono esattamente 3 iterazioni, quindi il limite viene raggiunto.
- La differenza con la ricerca lineare cresce molto rapidamente con n . La tabella seguente confronta il numero di confronti della ricerca lineare quando m non è nell'array (Osservazione 4.2) con il numero massimo di iterazioni della ricerca binaria:

n	ricerca lineare n confronti	ricerca binaria al più $\lfloor \log_2 n \rfloor + 1$ iterazioni
1 000	1 000	10
1 000 000	1 000 000	20
1 000 000 000	1 000 000 000	30

La ricerca binaria richiede che l'array sia ordinato in modo non decrescente. Su un array ordinato di n elementi esegue al più $\lfloor \log_2 n \rfloor + 1$ iterazioni, mentre la ricerca lineare può richiedere n confronti. Se l'array non è ordinato, la ricerca binaria può scartare erroneamente la metà che contiene m , e si deve usare la ricerca lineare.

4.3.2 Implementazione in C++

- Questa è un'implementazione in C++ dell'algoritmo di ricerca binaria. L'intervallo di ricerca iniziale è $a[0..n - 1]$.

```

1 // Restituisce l'indice di m nell'array a, oppure -1 se m non c'è
2 // L'array a deve essere ordinato in ordine non decrescente
3
4 int BinarySearch(int n, int *a, int m)
5 {
6     int left = 0;
7     int right = n - 1;
8
9     while (left <= right)
10    {
11        int mid = left + (right - left) / 2;
12
13        if (a[mid] == m)
14        {
15            return mid; // Elemento trovato all'indice mid
16        }
17
18        if (a[mid] < m)
19        {
20            left = mid + 1; // Cerca nella metà destra
21        }
22        else
23        {
24            right = mid - 1; // Cerca nella metà sinistra

```

```

25     }
26 }
27
28 return -1; // Elemento non trovato
29 }

```

- La divisione fra due valori `int` con $right - left \geq 0$ scarta la parte frazionaria, quindi $(right - left)/2$ è esattamente $\lfloor (right - left)/2 \rfloor$.
- L'indice centrale è calcolato come $left + (right - left)/2$ invece che $(left + right)/2$: le due espressioni hanno lo stesso valore, ma la somma $left + right$ può superare il massimo valore di un `int` quando l'array è molto lungo, mentre $left + (right - left)/2$ non supera mai $right$.
- La funzione può essere chiamata, per esempio, dal seguente `main`:

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int a[] = {3, 9, 11, 12, 12, 16, 19};
7     int n = 7;
8
9     cout << BinarySearch(n, a, 12) << endl; // stampa 3
10    cout << BinarySearch(n, a, 10) << endl; // stampa -1
11    cout << BinarySearch(n, a, 3) << endl; // stampa 0
12    cout << BinarySearch(n, a, 19) << endl; // stampa 6
13
14    return 0;
15 }

```

Esempio 4.4: esecuzione dell'implementazione in C++ della ricerca binaria

Consideriamo l'array ordinato dell'Esempio 4.3, memorizzato in C++ con indici da 0 a 6:

$a[j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$	$j = 6$
	3	9	11	12	12	16	19

All'inizio $left = 0$ e $right = 6$.

- Ricerca di $m = 12$:

iterazione	$left$	$right$	mid	confronto e aggiornamento
1	0	6	$0 + (6 - 0)/2 = 3$	$a[3] = 12 = m \Rightarrow$ Trovato! Restituisce 3

- Ricerca di $m = 10$:

iterazione	<i>left</i>	<i>right</i>	<i>mid</i>	confronto e aggiornamento
1	0	6	$0 + (6 - 0)/2 = 3$	$a[3] = 12 > 10 \Rightarrow right = 3 - 1 = 2$
2	0	2	$0 + (2 - 0)/2 = 1$	$a[1] = 9 < 10 \Rightarrow left = 1 + 1 = 2$
3	2	2	$2 + (2 - 2)/2 = 2$	$a[2] = 11 > 10 \Rightarrow right = 2 - 1 = 1$
–	2	1	–	$left > right \Rightarrow$ Restituisce -1

- Ricerca di $m = 3$:

iterazione	<i>left</i>	<i>right</i>	<i>mid</i>	confronto e aggiornamento
1	0	6	$0 + (6 - 0)/2 = 3$	$a[3] = 12 > 3 \Rightarrow right = 3 - 1 = 2$
2	0	2	$0 + (2 - 0)/2 = 1$	$a[1] = 9 > 3 \Rightarrow right = 1 - 1 = 0$
3	0	0	$0 + (0 - 0)/2 = 0$	$a[0] = 3 = m \Rightarrow$ Trovato! Restituisce 0

- Ricerca di $m = 19$:

iterazione	<i>left</i>	<i>right</i>	<i>mid</i>	confronto e aggiornamento
1	0	6	$0 + (6 - 0)/2 = 3$	$a[3] = 12 < 19 \Rightarrow left = 3 + 1 = 4$
2	4	6	$4 + (6 - 4)/2 = 5$	$a[5] = 16 < 19 \Rightarrow left = 5 + 1 = 6$
3	6	6	$6 + (6 - 6)/2 = 6$	$a[6] = 19 = m \Rightarrow$ Trovato! Restituisce 6

Ogni indice C++ è il corrispondente indice dell'Esempio 4.3 meno 1.

Punti chiave del capitolo

- Il problema della ricerca richiede un indice k tale che $a[k] = m$, oppure -1 se tale indice non esiste.
- La ricerca lineare funziona su ogni array, restituisce la prima occorrenza di m ed esegue fino a n confronti.
- La ricerca binaria funziona solo su array ordinati in modo non decrescente; dimezza l'intervallo di ricerca $a[left..right]$ a ogni iterazione ed esegue al più $\lfloor \log_2 n \rfloor + 1$ iterazioni.
- Nello pseudocodice gli indici vanno da 1 a n , in C++ da 0 a $n - 1$: gli indici restituiti differiscono di 1.

Algoritmi numerici

Introduzione

In questo capitolo studiamo quattro algoritmi classici che operano sui numeri naturali:

- il **cambio di base**, che scrive un numero naturale dato in base dieci come sequenza di cifre in un'altra base b ;
- il **test di primalità**, che decide se un numero naturale è primo;
- il **crivello di Eratostene**, che trova tutti i numeri primi fino a un dato numero naturale;
- l'**algoritmo di Euclide**, che calcola il massimo comun divisore di due numeri naturali.

Tutti e quattro gli algoritmi si basano su una sola operazione elementare, la **divisione con resto**: il cambio di base legge le cifre come resti, il test di primalità e il crivello cercano resti uguali a zero, e l'algoritmo di Euclide sostituisce una coppia di numeri con una coppia più piccola per mezzo di un resto. Per ogni algoritmo presentiamo lo pseudocodice (con array indicizzati a partire da 1), un'esecuzione passo per passo e un'implementazione in C++ (con array indicizzati a partire da 0), insieme alla sua esecuzione sugli stessi input.

5.1 Cambio di base

Nel capitolo «Rappresentazione dei numeri» abbiamo visto che una sequenza di k cifre $(d_{k-1} \dots d_1 d_0)_b$ in base b rappresenta il numero naturale $\sum_{i=0}^{k-1} d_i b^i$, e che servono $\lfloor \log_b n \rfloor + 1$ cifre per scrivere un numero naturale $n \geq 1$ in base b . In questa sezione risolviamo il problema inverso: dato n in base dieci, calcoliamo le sue cifre in base b .

5.1.1 Divisione con resto

Definizione 5.1: divisione con resto

Dati un numero naturale n e un numero naturale $b \geq 1$, la **divisione con resto** (o divisione euclidea) di n per b determina i due numeri naturali q e r tali che

$$n = q b + r \quad \text{con} \quad 0 \leq r < b.$$

Il numero n si chiama **dividendo**, b **divisore**, q **quoziente** e r **resto**. I due numeri q e r sono unici, e scriviamo

$$q = \lfloor n/b \rfloor \quad \text{e} \quad r = n \bmod b.$$

Il fatto chiave alla base del cambio di base è che una divisione per b estrae la cifra meno significativa di n in base b e lascia come quoziente il numero formato dalle cifre rimanenti.

Osservazione 5.2

Sia $n \geq 1$ un numero naturale la cui rappresentazione in base $b \geq 2$ è $(d_{k-1} \dots d_1 d_0)_b$. Allora

$$n \bmod b = d_0 \quad \text{e} \quad \lfloor n/b \rfloor = \sum_{i=1}^{k-1} d_i b^{i-1} = (d_{k-1} \dots d_1)_b,$$

dove il quoziente è 0 se $k = 1$.

Dimostrazione. Raccogliendo il fattore b da tutti i termini con $i \geq 1$, otteniamo

$$n = \sum_{i=0}^{k-1} d_i b^i = \left(\sum_{i=1}^{k-1} d_i b^{i-1} \right) b + d_0.$$

Poiché d_0 è una cifra in base b , si ha $0 \leq d_0 < b$. Per l'unicità del quoziente e del resto della divisione con resto, il quoziente di n diviso b è $\sum_{i=1}^{k-1} d_i b^{i-1}$ e il resto è d_0 . $\square$

Dividendo n per b , il resto è la cifra meno significativa d_0 , e il quoziente è il numero scritto con le altre cifre $(d_{k-1} \dots d_1)_b$. Dividendo di nuovo il quoziente per b si ottiene d_1 , e così via: ripetendo la divisione fino a ottenere quoziente 0 si producono tutte le cifre di n in base b , dalla meno significativa alla più significativa, in esattamente $\lfloor \log_b n \rfloor + 1$ divisioni.

5.1.2 L'algoritmo

Dato un numero naturale $n > 0$ rappresentato in base 10 (decimale) e una base di arrivo $b \geq 2$, consideriamo il problema di convertire n in base b .

- Pseudocodice dell'algoritmo per convertire sequenze di cifre dalla base dieci alla base b :

Algorithm: ChangeOfBase

Input : Un numero naturale $n \geq 1$ (in base 10) e una base $b \geq 2$

Output: Un array $d[1..nd]$ di cifre in base b e il numero di cifre nd

```

1 function ChangeOfBase( $n, b$ )
2    $nd := \lfloor \log_b n \rfloor + 1$                                 // Numero di cifre necessarie
3    $d := \text{new array}[1..nd]$ 
4    $k := 1$ 
5   while  $n > 0$  do
6      $q := \lfloor n/b \rfloor$                                     // Quoziente
7      $r := n \bmod b$                                        // Resto (cifra in base  $b$ )
8      $d[k] \leftarrow r$ 
9      $n \leftarrow q$ 
10     $k \leftarrow k + 1$ 
11  end
12  return  $d, nd$ 
13 end
  
```

- **Spiegazione:**

- L'algoritmo calcola il numero di cifre nd necessarie per rappresentare n in base b con la formula $\lfloor \log_b n \rfloor + 1$.
- Inizializza un array $d[1..nd]$ per memorizzare le cifre e un contatore k che tiene traccia della posizione corrente.
- Il ciclo principale divide ripetutamente n per b per estrarre le cifre da destra a sinistra (dalla meno significativa alla più significativa).
- A ogni iterazione il resto $r = n \bmod b$ fornisce la cifra successiva, che viene memorizzata in $d[k]$.
- Il quoziente $q = \lfloor n/b \rfloor$ diventa il nuovo valore di n per l'iterazione successiva.
- L'algoritmo termina quando $n = 0$, restituendo l'array d e il numero totale di cifre nd .
- Le cifre in d sono memorizzate in ordine inverso: l'elemento $d[k]$ è la cifra d_{k-1} di peso b^{k-1} , quindi $d[1] = d_0$ è la cifra meno significativa (la più a destra) e $d[nd] = d_{nd-1}$ è la cifra più significativa (la più a sinistra).

Esempio 5.1: conversione dalla base dieci alla base b

- Conversione di 7 in base $b = 2$:

iterazione k	n	n/b	q	r	cifra
1	7	$7/2$	3	1	$d[1] = d_0$ (meno significativa)
2	3	$3/2$	1	1	$d[2] = d_1$
3	1	$1/2$	0	1	$d[3] = d_2$ (più significativa)

Quindi 7 in base 2 è $(111)_2$, con $nd = \lfloor \log_2 7 \rfloor + 1 = 3$ cifre.

- Conversione di 8 in base $b = 2$:

iterazione k	n	n/b	q	r	cifra
1	8	$8/2$	4	0	$d[1] = d_0$ (meno significativa)
2	4	$4/2$	2	0	$d[2] = d_1$
3	2	$2/2$	1	0	$d[3] = d_2$
4	1	$1/2$	0	1	$d[4] = d_3$ (più significativa)

Quindi 8 in base 2 è $(1000)_2$, con $nd = \lfloor \log_2 8 \rfloor + 1 = 4$ cifre.

- Conversione di 25 in base $b = 3$:

iterazione k	n	n/b	q	r	cifra
1	25	$25/3$	8	1	$d[1] = d_0$ (meno significativa)
2	8	$8/3$	2	2	$d[2] = d_1$
3	2	$2/3$	0	2	$d[3] = d_2$ (più significativa)

Quindi 25 in base 3 è $(221)_3$, con $nd = \lfloor \log_3 25 \rfloor + 1 = 3$ cifre.

- Conversione di 100 in base $b = 8$:

iterazione k	n	n/b	q	r	cifra
1	100	$100/8$	12	4	$d[1] = d_0$ (meno significativa)
2	12	$12/8$	1	4	$d[2] = d_1$
3	1	$1/8$	0	1	$d[3] = d_2$ (più significativa)

Quindi 100 in base 8 è $(144)_8$, con $nd = \lfloor \log_8 100 \rfloor + 1 = 3$ cifre.

5.1.3 Implementazione in C++

- Questa è un'implementazione in C++ dell'algoritmo di conversione dalla base dieci alla base b :

```

1 // Converta un numero decimale n in base b
2 // La funzione restituisce il numero di cifre nd del risultato
3 // digits[i] contiene la cifra in posizione i (da destra a sinistra, a partire da i=0)
4
5 int ChangeToBase(int n, int b, int *digits)
6 {
7     int nd = 0; // contatore del numero di cifre
8
9     while (n > 0)
10    {
11        int q = n / b; // quoziente
12        int r = n % b; // resto
13
14        digits[nd] = r; // memorizza la cifra in posizione nd
15        n = q;          // n diventa il quoziente
16        nd++;          // incrementa il contatore
17    }
18
19    return nd; // restituisce il numero di cifre
20 }

```

- Per due numeri naturali n e b con $b \geq 1$, gli operatori n / b e $n \% b$ del C++ calcolano esattamente il quoziente $\lfloor n/b \rfloor$ e il resto $n \bmod b$ della divisione con resto.
- L'array `digits` è indicizzato a partire da 0: l'elemento `digits[i]` è la cifra d_i di peso b^i . La cifra memorizzata nello pseudocodice in $d[k]$ è memorizzata in C++ in `digits[k-1]`.
- La funzione non calcola $\lfloor \log_b n \rfloor + 1$: conta le cifre nella variabile `nd` man mano che le estrae. Ciò evita anche un problema sottile dell'aritmetica in virgola mobile: in doppia precisione $\log(1000.0)/\log(10.0)$ vale 2.9999999999999996, quindi la formula darebbe 3 cifre invece delle 4 cifre di $(1000)_{10}$.
- L'array `digits` deve essere dichiarato dal chiamante con un numero sufficiente di elementi; per esempio, 31 elementi bastano per ogni `int` positivo (al più $2^{31} - 1$) e per ogni base $b \geq 2$.

Esempio 5.2: esecuzione della funzione C++

Ogni riga mostra i valori calcolati in un'iterazione del ciclo `while` e il contenuto dell'array `digits` dopo l'iterazione (l'elemento appena scritto è in blu; “—” indica un elemento non ancora scritto).

- Conversione di $n = 7$ in base $b = 2$:

iterazione	q	r	digits[3]	digits[2]	digits[1]	digits[0]	n	nd
inizio	–	–	–	–	–	–	7	0
1	3	1	–	–	–	1	3	1
2	1	1	–	–	1	1	1	2
3	0	1	–	1	1	1	0	3

La funzione restituisce $nd = 3$; leggendo **digits** dalla posizione 2 alla posizione 0 si ottiene $(111)_2$.

- Conversione di $n = 8$ in base $b = 2$:

iterazione	q	r	digits[4]	digits[3]	digits[2]	digits[1]	digits[0]	n	nd
inizio	–	–	–	–	–	–	–	8	0
1	4	0	–	–	–	–	0	4	1
2	2	0	–	–	–	0	0	2	2
3	1	0	–	–	0	0	0	1	3
4	0	1	–	1	0	0	0	0	4

La funzione restituisce $nd = 4$; leggendo **digits** dalla posizione 3 alla posizione 0 si ottiene $(1000)_2$.

- Conversione di $n = 25$ in base $b = 3$:

iterazione	q	r	digits[3]	digits[2]	digits[1]	digits[0]	n	nd
inizio	–	–	–	–	–	–	25	0
1	8	1	–	–	–	1	8	1
2	2	2	–	–	2	1	2	2
3	0	2	–	2	2	1	0	3

La funzione restituisce $nd = 3$; leggendo **digits** dalla posizione 2 alla posizione 0 si ottiene $(221)_3$.

- Conversione di $n = 100$ in base $b = 8$:

iterazione	q	r	digits[3]	digits[2]	digits[1]	digits[0]	n	nd
inizio	–	–	–	–	–	–	100	0
1	12	4	–	–	–	4	12	1
2	1	4	–	–	4	4	1	2
3	0	1	–	1	4	4	0	3

La funzione restituisce $nd = 3$; leggendo **digits** dalla posizione 2 alla posizione 0 si ottiene $(144)_8$.

Punti chiave.

- Il resto della divisione di n per b è la cifra meno significativa di n in base b ; il quoziente

è il numero scritto con le altre cifre.

- L'algoritmo esegue una divisione per ogni cifra: $\lfloor \log_b n \rfloor + 1$ iterazioni.
- Le cifre vengono prodotte dalla meno significativa alla più significativa, quindi il risultato si legge al contrario: da $d[nd]$ a $d[1]$ nello pseudocodice, da `digits[nd-1]` a `digits[0]` in C++.

5.2 Test di primalità

Il cambio di base usa il resto $n \bmod b$ per leggere una cifra. Un resto uguale a zero ha un significato proprio: dice che la divisione è esatta, cioè che b è un divisore di n . Contare i divisori porta ai numeri primi.

Definizione 5.3: divisore

Un numero naturale $d \geq 1$ è un **divisore** di un numero naturale n se esiste un numero naturale k tale che $n = k d$:

$$d \text{ è un divisore di } n \iff \exists k \in \mathbb{N}, n = k d$$

o, equivalentemente, se $n \bmod d = 0$.

Definizione 5.4: numero primo

Un numero naturale $n \geq 2$ è **primo** se i suoi unici divisori sono 1 e n stesso. Un numero naturale $n \geq 2$ che non è primo si dice **composto**: ha un divisore d con $2 \leq d \leq n - 1$. I numeri 0 e 1 non sono primi.

Dato un numero naturale n , consideriamo il problema di stabilire se n è un numero primo.

5.2.1 Un primo algoritmo

La definizione suggerisce un algoritmo diretto: provare, uno per uno, i candidati divisori di n diversi da 1 e da n . L'osservazione seguente mostra che i candidati maggiori di $n/2$ si possono saltare.

Osservazione 5.5

Per ogni numero naturale $n \geq 2$, se d è un divisore di n con $d < n$, allora $d \leq n/2$.

Dimostrazione. Poiché d è un divisore di n , esiste un numero naturale k tale che $n = k d$. Poiché $d < n$, si ha $k \geq 2$, quindi

$$d = \frac{n}{k} \leq \frac{n}{2}.$$

□

- Ne segue che basta controllare se qualche numero naturale d in $\{2, 3, \dots, \lfloor n/2 \rfloor\}$ è un divisore di n . Se non esiste alcun divisore di questo tipo, n è primo.
- Pseudocodice dell'algoritmo Basic Primality Test (test di primalità di base):

Algorithm: BasicPrimalityTest**Input** : Un numero naturale n **Output**: **true** se n è primo, **false** altrimenti

```

1 function BasicPrimalityTest( $n$ )
2   if  $n < 2$  then
3     return false                                // 0 e 1 non sono primi
4   end
5   for  $d := 2$  to  $\lfloor n/2 \rfloor$  do
6     if  $n \bmod d = 0$  then
7       return false                                //  $d$  divide  $n$ : non primo
8     end
9   end
10  return true                                    // Nessun divisore trovato:  $n$  è primo
11 end

```

- Spiegazione dell'algoritmo:
 - I numeri minori di 2 non sono primi per definizione.
 - Il ciclo prova ogni candidato divisore d da 2 fino a $\lfloor n/2 \rfloor$.
 - Se d è un divisore di n (cioè $n \bmod d = 0$), la funzione restituisce subito **false**.
 - Se il ciclo termina senza trovare alcun divisore, n è primo e la funzione restituisce **true**.
- Per un numero primo il ciclo viene eseguito per intero: per $n = 29$ l'algoritmo prova i 13 candidati $d = 2, 3, \dots, 14$. Mostriamo ora che bastano molti meno candidati.

5.2.2 Provare i divisori fino a $\sqrt{n}$

I divisori di un numero naturale si presentano a coppie: d e n/d . In ogni coppia, uno dei due numeri è al più $\sqrt{n}$.

Osservazione 5.6

Per ogni numero naturale $n \geq 2$, se d è un divisore di n allora n/d è un divisore di n .

Dimostrazione. Se d è un divisore di n , allora esiste un numero naturale k tale che

$$n = k d.$$

Poiché $k = n/d$, ne segue che n/d è un numero naturale. Quindi

$$n = d \left(\frac{n}{d} \right),$$

quindi anche n/d è un divisore di n . □

Osservazione 5.7

Per ogni numero naturale $n \geq 2$, se $d > \sqrt{n}$ allora $n/d < \sqrt{n}$.

Dimostrazione. Se $d > \sqrt{n}$, dividendo n per entrambi i membri di questa disuguaglianza otteniamo

$$\frac{n}{d} < \frac{n}{\sqrt{n}} = \sqrt{n}.$$

Quindi n/d è minore di $\sqrt{n}$. □

Corollario 5.8

Ogni numero naturale composto $n \geq 2$ ha un divisore d con $2 \leq d \leq \lfloor \sqrt{n} \rfloor$. Equivalentemente, un numero naturale $n \geq 2$ è primo se e solo se nessun numero naturale in $\{2, 3, \dots, \lfloor \sqrt{n} \rfloor\}$ è un divisore di n .

Dimostrazione. Poiché n è composto, ha un divisore d con $2 \leq d \leq n-1$. Se $d \leq \sqrt{n}$, allora $d \leq \lfloor \sqrt{n} \rfloor$, perché d è un numero naturale, e abbiamo finito. Altrimenti $d > \sqrt{n}$: per le due osservazioni precedenti, n/d è un divisore di n con $n/d < \sqrt{n}$, e $n/d \geq 2$ perché $d < n$. Quindi n/d è un divisore di n con $2 \leq n/d \leq \lfloor \sqrt{n} \rfloor$. □

Dalle osservazioni precedenti sappiamo che, per ogni numero naturale $n \geq 2$, se $d > \sqrt{n}$ è un divisore di n , allora anche n/d è un divisore di n e $n/d < \sqrt{n}$. Ne segue che ogni divisore di n maggiore di $\sqrt{n}$ è accoppiato a un divisore di n minore di $\sqrt{n}$. Quindi, se n ha un qualche divisore (diverso da 1 e da n), almeno uno di essi deve essere $\leq \sqrt{n}$.

Esempio 5.3: illustrazione della proprietà

Consideriamo $n = 36$: $\sqrt{36} = 6$. I divisori di 36 diversi da 1 e da 36 sono 2, 3, 4, 6, 9, 12, 18.

divisore $d > \sqrt{36}$	divisore accoppiato $36/d < \sqrt{36}$
9	$36/9 = 4$
12	$36/12 = 3$
18	$36/18 = 2$

Ogni divisore > 6 è accoppiato a un divisore < 6 ; il divisore $6 = \sqrt{36}$ è accoppiato con se stesso.

- Per il corollario, il ciclo dell'algoritmo di base può fermarsi a $\lfloor \sqrt{n} \rfloor$ invece che a $\lfloor n/2 \rfloor$.

- Pseudocodice dell'algoritmo Primality Test (test di primalità):

Algorithm: PrimalityTest**Input** : Un numero naturale n **Output**: true se n è primo, false altrimenti

```

1 function PrimalityTest( $n$ )
2   if  $n < 2$  then
3     return false                                // 0 e 1 non sono primi
4   end
5    $d := 2$ 
6   while  $d \leq \lfloor \sqrt{n} \rfloor$  do
7     if  $n \bmod d = 0$  then
8       return false                                //  $d$  divide  $n$ : non primo
9     end
10     $d \leftarrow d + 1$ 
11  end
12  return true                                // Nessun divisore trovato:  $n$  è primo
13 end

```

- Spiegazione dell'algoritmo:

- I numeri minori di 2 non sono primi per definizione.
- Il ciclo prova ogni candidato divisore d da 2 fino a $\lfloor \sqrt{n} \rfloor$.
- Se d divide n (cioè $n \bmod d = 0$), la funzione restituisce subito **false**.
- Se il ciclo termina senza trovare alcun divisore, allora, per il corollario, n è primo e la funzione restituisce **true**.
- Per $n = 2$ e $n = 3$ si ha $\lfloor \sqrt{n} \rfloor = 1 < 2$: il ciclo non viene eseguito e la funzione restituisce **true**.

Esempio 5.4: test di primalità

- Verifica $n = 36$: $\lfloor \sqrt{36} \rfloor = 6$

d	$36 \bmod d$	azione
2	0	restituisce false
36 non è primo ($36 = 2 \cdot 18$).		

- Verifica $n = 29$: $\lfloor \sqrt{29} \rfloor = 5$

d	$29 \bmod d$	azione
2	$1 \neq 0$	continua
3	$2 \neq 0$	continua
4	$1 \neq 0$	continua
5	$4 \neq 0$	continua
Il ciclo termina. Restituisce true : 29 è primo .		

- Verifica $n = 25$: $\lfloor \sqrt{25} \rfloor = 5$

d	$25 \bmod d$	azione
2	$1 \neq 0$	continua
3	$1 \neq 0$	continua
4	$1 \neq 0$	continua
5	0	restituisce false
25 non è primo ($25 = 5 \cdot 5$).		

Per il numero primo 29 l'algoritmo prova 4 candidati, mentre l'algoritmo di base ne prova 13.

5.2.3 Implementazione in C++

- Questa è un'implementazione in C++ dell'algoritmo Primality Test:

```

1 // Restituisce true se n è primo, false altrimenti
2
3 bool PrimalityTest(int n)
4 {
5     if (n < 2)
6     {
7         return false; // 0 e 1 non sono primi
8     }
9
10    for (int d = 2; d <= (int)sqrt((double)n); d++)
11    {
12        if (n % d == 0)
13        {
14            return false; // d divide n: non è primo
15        }
16    }
17
18    return true; // Nessun divisore trovato: n è primo
19 }
```

- La funzione `sqrt` dell'header `cmath` opera su numeri in virgola mobile: il cast `(double)` converte `n` in un numero in virgola mobile, e il cast `(int)` tronca la radice quadrata, quindi il test confronta `d` con $\lfloor \sqrt{n} \rfloor$.

- Il ciclo while dello pseudocodice diventa un ciclo for: l'inizializzazione $d = 2$, il test su d e l'incremento $d++$ sono scritti nella sua intestazione.

Esempio 5.5: esecuzione della funzione C++

Sui tre numeri dell'esempio precedente la funzione esegue gli stessi test:

n	<code>(int)sqrt((double)n)</code>	valori di d provati	valore restituito
36	6	2 (con $36 \% 2 == 0$)	false: 36 non è primo
29	5	2, 3, 4, 5 (nessun resto è 0)	true: 29 è primo
25	5	2, 3, 4, 5 (con $25 \% 5 == 0$)	false: 25 non è primo

Per $n = 29$, dopo l'iterazione con $d = 5$ l'incremento dà $d = 6$, il test $6 \leq 5$ è falso e il ciclo termina.

Punti chiave.

- I divisori di n si presentano a coppie d e n/d , e in ogni coppia uno dei due numeri è al più $\sqrt{n}$.
- Quindi $n \geq 2$ è primo se e solo se nessun numero naturale in $\{2, 3, \dots, \lfloor \sqrt{n} \rfloor\}$ lo divide.
- Il Primality Test esegue al più $\lfloor \sqrt{n} \rfloor - 1$ iterazioni, invece delle $\lfloor n/2 \rfloor - 1$ iterazioni dell'algoritmo di base.

5.3 Crivello di Eratostene

Il Primality Test risponde alla domanda per un solo numero. Quando servono tutti i numeri primi fino a n , chiamarlo su $2, 3, \dots, n$ ripete molte divisioni; il crivello lavora al contrario: invece di cercare i divisori di ciascun numero, cancella i multipli di ciascun primo.

Dato un numero naturale n , consideriamo il problema di determinare tutti i numeri primi minori o uguali a n .

- Il **crivello di Eratostene** è un algoritmo antico (attribuito al matematico greco Eratostene di Cirene, ca. 276–194 a.C.) che calcola tutti i numeri primi fino a un numero naturale n .
- L'idea chiave è partire con tutti i numeri naturali da 2 a n marcati come potenzialmente primi, e poi marcare iterativamente come composti i multipli di ciascun primo.

Definizione 5.9: multiplo

Un numero naturale m è un **multiplo** di un numero naturale $p \geq 1$ se esiste un numero naturale k tale che $m = k p$:

$$m \text{ è un multiplo di } p \iff \exists k \in \mathbb{N}, m = k p$$

o, equivalentemente, se $m \bmod p = 0$.

La prima osservazione dice da dove può partire la marcatura dei multipli di p .

Osservazione 5.10

Per ogni numero naturale $p \geq 2$, se un numero naturale m è un multiplo di p con $p < m < p^2$, allora m ha un divisore d con $2 \leq d < p$.

Dimostrazione. Poiché m è un multiplo di p , esiste un numero naturale d tale che $m = d p$. Allora d è un divisore di m .

Inoltre, poiché $m > p$ e $m = d p$, si ha $d \geq 2$. Poiché $m < p^2$ e $m = d p$, si ha $d < p$. Quindi $2 \leq d < p$. $\square$

Dall'osservazione precedente, per ogni primo p e ogni multiplo m di p con $p < m < p^2$, il numero m ha un divisore d con $2 \leq d < p$. Quindi m ha anche un divisore primo q con $2 \leq q < p$, ed era perciò già stato marcato come composto quando è stato elaborato q . Dunque, nel marcare i multipli di p , si può partire da p^2 .

Esempio 5.6: illustrazione della proprietà

Sia $p = 5$. I suoi multipli compresi tra p e $p^2 = 25$ sono $m = 10, 15, 20$.

multiplo m	$m = d \cdot p$	divisore $d < p$	già marcato quando
10	$10 = 2 \cdot 5$	$d = 2$	$p = 2$ (da $j = 4$, passo 2)
15	$15 = 3 \cdot 5$	$d = 3$	$p = 3$ (da $j = 9$, passo 3)
20	$20 = 4 \cdot 5$	$d = 4$	$p = 2$ (da $j = 4$, passo 2)

Il numero 20 ha il divisore $d = 4$, che non è primo, e anche il divisore primo 2, che lo ha marcato. Il multiplo successivo è $25 = 5 \cdot 5 = p^2$, che non è stato marcato da nessun primo precedente. Quindi, per $p = 5$, si può iniziare a marcare i numeri da 25.

La seconda osservazione dice dove può fermarsi il ciclo esterno su p .

Osservazione 5.11

Per ogni numero naturale $n \geq 2$, ogni numero naturale composto $m \leq n$ ha un divisore primo q con $q \leq \lfloor \sqrt{n} \rfloor$.

Dimostrazione. Per il corollario della sezione «Test di primalità», m ha un divisore d con $2 \leq d \leq \lfloor \sqrt{m} \rfloor \leq \lfloor \sqrt{n} \rfloor$. Sia q il più piccolo divisore di m con $q \geq 2$; allora $q \leq d \leq \lfloor \sqrt{n} \rfloor$. Il numero q è primo: se q avesse un divisore c con $2 \leq c < q$, allora c sarebbe anche un divisore di m (da $q = k_1 c$ e $m = k_2 q$ si ottiene $m = k_2 k_1 c$), contro la scelta di q come il più piccolo. $\square$

- Quindi, dopo che sono stati marcati i multipli di tutti i primi $p \leq \lfloor \sqrt{n} \rfloor$, ogni numero composto fino a n è marcato, e il ciclo esterno può fermarsi.
- Pseudocodice del crivello di Eratostene:

Algorithm: SieveOfEratosthenes

Input : Un numero naturale $n \geq 1$
Output: Un array booleano $prime[1..n]$ dove $prime[i] = 1$ se e solo se i è primo

```

1 function SieveOfEratosthenes( $n$ )
2    $prime := \text{new array}[1..n]$  // dichiara l'array booleano
3    $prime \leftarrow 1$  // inizializza tutti gli elementi a 1
4    $prime[1] \leftarrow 0$  // 1 non è primo
5    $p := 2$ 
6   while  $p \leq \lfloor \sqrt{n} \rfloor$  do
7     if  $prime[p] = 1$  then
8        $j := p^2$ 
9       while  $j \leq n$  do
10         $prime[j] \leftarrow 0$  // j non è primo
11         $j \leftarrow j + p$ 
12      end
13    end
14     $p \leftarrow p + 1$ 
15  end
16  return  $prime$ 
17 end
```

- Spiegazione dell'algoritmo:
 - L'array $prime$ viene dichiarato e tutti i suoi elementi sono inizializzati a 1; l'elemento $prime[1]$ viene subito posto a 0, poiché 1 non è primo.
 - Il ciclo esterno scorre ogni candidato primo p da 2 fino a $\lfloor \sqrt{n} \rfloor$.
 - Se $prime[p] = 1$, allora p è primo (un p composto sarebbe stato marcato dal suo più piccolo divisore primo), e tutti i suoi multipli a partire da p^2 sono posti a 0.
 - Il ciclo interno parte da p^2 perché tutti i multipli più piccoli $2p, 3p, \dots, (p-1)p$ sono già stati posti a 0 nell'elaborare i primi più piccoli 2, 3, 5, 7, $\dots$
 - Dopo l'esecuzione dell'algoritmo, $prime[i] = 1$ se e solo se i è primo.

Esempio 5.7: crivello di Eratostene ($n = 10$)

Con $n = 10$ ($\lfloor \sqrt{10} \rfloor = 3$), ogni riga mostra lo stato dell'array $prime[1..10]$ dopo l'assegnamento $prime[j] \leftarrow 0$. Le celle blu indicano gli elementi appena posti a 0; le celle rosse indicano il valore corrente di p .

p	j	$prime[1..10]$										azione
		1	2	3	4	5	6	7	8	9	10	
—	—	0	1	1	1	1	1	1	1	1	1	inizializzazione
2	4	0	1	1	0	1	1	1	1	1	1	$prime[4] \leftarrow 0$
	6	0	1	1	0	1	0	1	1	1	1	$prime[6] \leftarrow 0$
	8	0	1	1	0	1	0	1	0	1	1	$prime[8] \leftarrow 0$
	10	0	1	1	0	1	0	1	0	1	0	$prime[10] \leftarrow 0$
	12											$j = 12 > 10$: arresto
3	9	0	1	1	0	1	0	1	0	0	0	$prime[9] \leftarrow 0$
	12											$j = 12 > 10$: arresto

Il ciclo esterno termina: $p = 4 > \lfloor \sqrt{10} \rfloor = 3$.

Risultato: i numeri primi ≤ 10 sono 2, 3, 5, 7.

Esempio 5.8: crivello di Eratostene ($n = 20$)

Con $n = 20$ ($\lfloor \sqrt{20} \rfloor = 4$), ogni riga mostra lo stato dell'array $prime[1..20]$ dopo l'assegnamento $prime[j] \leftarrow 0$. Celle blu: elemento appena posto a 0; celle rosse: valore corrente di p . Quando l'elemento è già 0, l'assegnamento non modifica l'array.

		prime[1..20]																					
p	j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	azione	
—	—	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	inizializzazione	
2	4	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	prime[4] ← 0	
	6	0	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	prime[6] ← 0	
	8	0	1	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	prime[8] ← 0	
	10	0	1	1	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	prime[10] ← 0	
	12	0	1	1	0	1	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	prime[12] ← 0	
	14	0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	1	1	1	1	1	prime[14] ← 0	
	16	0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1	1	1	prime[16] ← 0	
	18	0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1	prime[18] ← 0	
	20	0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	prime[20] ← 0	
	22																					j = 22 > 20: arresto	
3	9	0	1	1	0	1	0	1	0	0	0	1	0	1	0	1	0	1	0	1	0	prime[9] ← 0	
	12	0	1	1	0	1	0	1	0	0	0	1	0	1	0	1	0	1	0	1	0	prime[12] ← 0 (già 0)	
	15	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	prime[15] ← 0	
	18	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	prime[18] ← 0 (già 0)	
	21																					j = 21 > 20: arresto	
4																						prime[4] = 0: salta	

Il ciclo esterno termina: $p = 5 > \lfloor \sqrt{20} \rfloor = 4$.

Risultato: i numeri primi ≤ 20 sono 2, 3, 5, 7, 11, 13, 17, 19.

5.3.1 Implementazione in C++

- Questa è un'implementazione in C++ del crivello di Eratostene:

```

1 // l'array in input è già dichiarato con n+1 posizioni
2 // e inizializzato a 1, e n vale almeno 2
3 int SieveOfEratosthenes(int n, int *prime)
4 {
5     prime[0] = 0;
6     prime[1] = 0;
7
8     for (int p = 2; p <= (int)sqrt((double)n); p++)
9     {
10         if (prime[p] == 1)
11         {
12             // Marca i multipli di p a partire da p*p
13             for (int j = p * p; j <= n; j += p)
14             {
15                 prime[j] = 0; // j non è primo
16             }
17         }
18     }
19
20     return 1;
21 }

```

- In C++ l'array è indicizzato a partire da 0. Per mantenere l'elemento `prime[i]` associato al numero i , l'array ha $n+1$ elementi, `prime[0..n]`; l'elemento in più `prime[0]` si riferisce al numero 0, che non è primo, ed è posto a 0.
- I due cicli while dello pseudocodice diventano due cicli for; quello interno avanza con `j += p`.

Esempio 5.9: esecuzione della funzione C++

Ogni riga mostra l'array `prime[0..n]` al termine di un'iterazione del ciclo esterno, con i valori di `j` del suo ciclo interno; gli elementi modificati in quell'iterazione sono in blu.

- Esecuzione per $n = 10$ (`(int)sqrt((double)10) = 3`):

p	prime[0..10]										
	0	1	2	3	4	5	6	7	8	9	10
inizio	0	0	1	1	1	1	1	1	1	1	1
2	0	0	1	1	0	1	0	1	0	1	0
3	0	0	1	1	0	1	0	1	0	0	0

Valori di `j` nel ciclo interno: per `p = 2`: `j = 4, 6, 8, 10`; per `p = 3`: `j = 9`.

- Esecuzione per $n = 20$ (`(int)sqrt((double)20) = 4`):

	prime[0..20]																				
p	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
inizio	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0	0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
3	0	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0
4	0	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0

Valori di j nel ciclo interno: per $p = 2$: $j = 4, 6, 8, 10, 12, 14, 16, 18, 20$; per $p = 3$: $j = 9, 12, 15, 18$; per $p = 4$: nessun ciclo interno, poiché $\text{prime}[4] = 0$.

Alla fine gli elementi uguali a 1 sono quelli dei numeri primi: 2, 3, 5, 7 per $n = 10$, e 2, 3, 5, 7, 11, 13, 17, 19 per $n = 20$.

Punti chiave.

- Il crivello non divide: marca come composti i multipli di ciascun primo p .
- La marcatura dei multipli di p parte da p^2 : i multipli più piccoli hanno un divisore primo più piccolo e sono già marcati.
- Il ciclo esterno si ferma a $\lfloor \sqrt{n} \rfloor$: ogni numero composto fino a n ha un divisore primo non maggiore di $\lfloor \sqrt{n} \rfloor$.

5.4 Massimo comun divisore

L'ultimo algoritmo del capitolo tratta due numeri alla volta: cerchiamo i divisori che hanno in comune.

Definizione 5.12: divisore comune e massimo comun divisore

Siano n, m due numeri naturali con $n, m \geq 1$.

- Un numero naturale $d \geq 1$ è un **divisore comune** di n e m se d è un divisore sia di n sia di m , cioè se esistono numeri naturali k_1, k_2 tali che

$$n = k_1 d \quad \text{e} \quad m = k_2 d.$$

Equivalentemente, $n \bmod d = 0$ e $m \bmod d = 0$.

- Il **massimo comun divisore** di n e m , indicato con $\text{gcd}(n, m)$, è il più grande divisore comune di n e m .

- Il massimo comun divisore esiste sempre: 1 è un divisore comune, e ogni divisore comune è al più $\min\{n, m\}$.
- Per esempio, i divisori comuni di 48 e 18 sono 1, 2, 3, 6, quindi $\text{gcd}(48, 18) = 6$.

Dati due numeri naturali n, m con $n, m \geq 1$, consideriamo il problema di calcolare il loro massimo comun divisore.

L'osservazione seguente sostituisce la coppia (n, m) con una coppia di numeri più piccoli che ha lo stesso massimo comun divisore.

Osservazione 5.13

Per ogni coppia di numeri naturali $n \geq 1$ e $m \geq 1$,

$$\gcd(n, m) = \gcd(m, n \bmod m).$$

Dimostrazione. Sia $q = \lfloor n/m \rfloor$, in modo che $n = qm + (n \bmod m)$.

Sia d_1 un divisore comune di n e m . Allora esistono numeri naturali k_1, k_2 tali che

$$n = k_1 d_1 \quad \text{e} \quad m = k_2 d_1.$$

Poiché $n \bmod m = n - qm$, otteniamo

$$n \bmod m = (k_1 - qk_2) d_1,$$

dove $k_1 - qk_2 \geq 0$ perché $n \bmod m \geq 0$. Quindi d_1 è anche un divisore di $n \bmod m$. Perciò d_1 è un divisore comune di m e $n \bmod m$.

Viceversa, sia d_2 un divisore comune di m e $n \bmod m$. Allora esistono numeri naturali k_1, k_2 tali che

$$m = k_1 d_2 \quad \text{e} \quad n \bmod m = k_2 d_2.$$

Poiché $n = qm + (n \bmod m)$, otteniamo

$$n = (qk_1 + k_2) d_2,$$

quindi d_2 è anche un divisore di n . Perciò d_2 è un divisore comune di n e m .

Dunque n e m hanno esattamente gli stessi divisori comuni di m e $n \bmod m$, e quindi

$$\gcd(n, m) = \gcd(m, n \bmod m).$$

Nel membro di destra, $\gcd(m, 0)$ è inteso come il più grande divisore di m (ogni $d \geq 1$ divide $0 = 0 \cdot d$), cioè m stesso. $\square$

5.4.1 Algoritmo di Euclide

- L'**algoritmo di Euclide** (attribuito all'antico matematico greco Euclide, ca. 300 a.C.) calcola $\gcd(n, m)$ in modo efficiente sostituendo ripetutamente la coppia (n, m) con $(m, n \bmod m)$.
- L'algoritmo non richiede $n \geq m$: se $n < m$, allora $n \bmod m = n$, e la prima sostituzione scambia semplicemente i due numeri, trasformando (n, m) in (m, n) .

- Per l'osservazione precedente, sostituire (n, m) con $(m, n \bmod m)$ non cambia il massimo comun divisore.
- Poiché $n \bmod m < m$, il secondo argomento decresce strettamente a ogni sostituzione.
- Dunque l'algoritmo termina quando il resto diventa 0; a quel punto il massimo comun divisore è l'altro argomento:

$$\gcd(n, 0) = n.$$

- Pseudocodice dell'algoritmo di Euclide:

Algorithm: EuclideanAlgorithm

Input : Due numeri naturali $n, m \geq 1$

Output: $\gcd(n, m)$

```

1 function EuclideanAlgorithm( $n, m$ )
2   while  $m \neq 0$  do
3      $r := n \bmod m$ 
4      $n \leftarrow m$ 
5      $m \leftarrow r$ 
6   end
7   return  $n$                                      //  $m = 0$ :  $n$  è il MCD
8 end
  
```

- Spiegazione dell'algoritmo:
 - A ogni passo, (n, m) viene sostituita da $(m, n \bmod m)$.
 - Per l'osservazione precedente, questa sostituzione conserva il valore di $\gcd(n, m)$: indicando con n_0 e m_0 i valori in input, a ogni iterazione $\gcd(n, m) = \gcd(n_0, m_0)$.
 - Poiché $n \bmod m < m$, il valore di m decresce strettamente a ogni iterazione, quindi l'algoritmo termina.
 - Quando $m = 0$, l'uguaglianza $\gcd(n, m) = \gcd(n_0, m_0)$ insieme a $\gcd(n, 0) = n$ dà il risultato.

Esempio 5.10: calcolo del MCD

- Calcoliamo $\gcd(48, 18)$:

passo	n	m	$r = n \bmod m$
1	48	18	$48 \bmod 18 = 12$
2	18	12	$18 \bmod 12 = 6$
3	12	6	$12 \bmod 6 = 0$
4	6	0	— (il ciclo termina)

Risultato: $\gcd(48, 18) = 6$.

- Calcoliamo $\gcd(252, 105)$:

passo	n	m	$r = n \bmod m$
1	252	105	$252 \bmod 105 = 42$
2	105	42	$105 \bmod 42 = 21$
3	42	21	$42 \bmod 21 = 0$
4	21	0	— (il ciclo termina)

Risultato: $\gcd(252, 105) = 21$.

5.4.2 Coprimalità**Definizione 5.14: numeri coprimi**

Due numeri naturali $n, m \geq 1$ si dicono **coprimi** (o **primi tra loro**) se non hanno alcun divisore comune diverso da 1:

$$n \text{ e } m \text{ sono coprimi} \iff \gcd(n, m) = 1.$$

- Poiché l'algoritmo di Euclide calcola $\gcd(n, m)$, per verificare la coprimalità basta chiamare l'algoritmo e controllare se il risultato è uguale a 1.

Esempio 5.11: test di coprimalità

- Calcoliamo $\gcd(35, 12)$: l'algoritmo di Euclide restituisce 1. Quindi 35 e 12 sono **coprimi**.
- Calcoliamo $\gcd(48, 18)$: l'algoritmo di Euclide restituisce 6. Quindi 48 e 18 **non sono coprimi**.

5.4.3 Implementazione in C++

- Questa è un'implementazione in C++ dell'algoritmo di Euclide:

```

1 int EuclideanAlgorithm(int n, int m)
2 {
3     while (m != 0)
4     {
5         int r = n % m;
6         n = m;
7         m = r;
8     }
9     return n; // m == 0: n è il MCD
10 }

```

- I parametri `n` e `m` sono passati per valore: la funzione modifica le proprie copie, e le variabili del chiamante conservano i loro valori.

Esempio 5.12: esecuzione della funzione C++

Ogni riga mostra il test del ciclo `while`, il resto calcolato e i valori di `n` e `m` al termine dell'iterazione (in blu).

- Esecuzione per `n = 48, m = 18`:

iterazione	test <code>m != 0</code>	<code>r = n % m</code>	<code>n</code>	<code>m</code>
inizio			48	18
1	$18 \neq 0$	$48 \bmod 18 = 12$	18	12
2	$12 \neq 0$	$18 \bmod 12 = 6$	12	6
3	$6 \neq 0$	$12 \bmod 6 = 0$	6	0
Test <code>m != 0</code> falso: il ciclo termina e la funzione restituisce <code>n = 6</code> .				

- Esecuzione per `n = 35, m = 12`:

iterazione	test <code>m != 0</code>	<code>r = n % m</code>	<code>n</code>	<code>m</code>
inizio			35	12
1	$12 \neq 0$	$35 \bmod 12 = 11$	12	11
2	$11 \neq 0$	$12 \bmod 11 = 1$	11	1
3	$1 \neq 0$	$11 \bmod 1 = 0$	1	0
Test <code>m != 0</code> falso: il ciclo termina e la funzione restituisce <code>n = 1</code> .				

Quindi $\gcd(48, 18) = 6$ e $\gcd(35, 12) = 1$.

Punti chiave.

- Le coppie (n, m) e $(m, n \bmod m)$ hanno gli stessi divisori comuni, quindi lo stesso massimo comun divisore.
- L'algoritmo di Euclide ripete la sostituzione finché il secondo numero è 0, e restituisce il primo, poiché $\gcd(n, 0) = n$.
- Due numeri sono coprimi se e solo se l'algoritmo di Euclide restituisce 1.