



SAPIENZA
UNIVERSITÀ DI ROMA

Algoritmi

Volume 4 — Analisi degli algoritmi

L'analisi degli algoritmi: la dimensione dell'input, il tempo di esecuzione nel caso migliore, peggiore e medio, e la correttezza dimostrata con le invarianti di ciclo.

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

11	Dimensione dell'input	2
	Introduzione	2
	11.1 Dimensione dell'input	2
	11.2 Notazione asintotica: un richiamo	3
	11.3 Istanza di input costituita da un numero naturale	4
	11.4 Istanza di input costituita da due numeri naturali	6
	11.5 Istanza di input costituita da un array di numeri naturali	8
	11.6 Dimensione dell'input e numero di passi	13
12	Tempo di esecuzione	17
	12.1 Tempo di esecuzione di un algoritmo	17
	12.2 Ricerca del massimo	19
	12.3 Insertion sort	23
	12.4 Cambio di base	27
	12.5 Test di primalità	30
	12.6 Riepilogo	33
13	Correttezza	35
	13.1 Correttezza di un algoritmo	35
	13.2 Correttezza di Find Maximum	37
	13.3 Correttezza di Insertion Sort	39
	13.4 Correttezza di Binary Search	43
	13.5 Correttezza dell'algoritmo di Euclide	45
	13.6 Correttezza degli algoritmi ricorsivi	47



Dimensione dell'input

Introduzione

Per confrontare due algoritmi che risolvono lo stesso problema contiamo quante operazioni eseguono. Questo numero cresce con l'input: ordinare un array di un milione di componenti richiede più tempo che ordinare un array di dieci componenti, e verificare se un numero di venti cifre è primo richiede più tempo che verificarlo per un numero di due cifre. Per rendere preciso questo discorso serve una misura di quanto sia *grande* un input.

In questo capitolo fissiamo questa misura: la **dimensione dell'input** è il numero di bit necessari per scrivere l'input in binario, come nel capitolo «Rappresentazione dei numeri». Procediamo così:

- definiamo la dimensione dell'input e richiamiamo la notazione asintotica usata per descriverne l'ordine di crescita (capitolo 9 «Notazione asintotica»);
- calcoliamo la dimensione dell'input, e il suo ordine di crescita, quando l'input è un numero naturale, due numeri naturali, oppure un array di numeri naturali;
- mostriamo perché la distinzione è importante: lo stesso numero di passi può essere piccolo o enorme rispetto alla dimensione dell'input, a seconda che il parametro n sia il *valore* di un numero o il *numero di componenti* di un array.

11.1 Dimensione dell'input

Definizione 11.1: dimensione dell'input

La **dimensione dell'input** $|I|$ di un algoritmo è il **numero totale di bit necessari per rappresentare in binario una data istanza di input I** .

- La dimensione dell'input è una funzione che associa a ogni istanza di input un numero naturale, pari al numero totale di bit necessari per rappresentare in binario l'istanza di input.
- Consideriamo algoritmi le cui istanze di input sono costituite da uno o più numeri naturali, oppure da array di numeri naturali.

- Nel capitolo «Rappresentazione dei numeri» abbiamo visto che il numero minimo di cifre necessarie per scrivere un numero naturale $n \geq 1$ in base 2 è $\lfloor \log_2 n \rfloor + 1$. Equivalentemente, un numero naturale n si scrive con esattamente k bit se e solo se

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

Esempio 11.1: il numero di bit di alcune istanze di input

- Un numero naturale, $n = 37$. Poiché $2^5 = 32 \leq 37 < 64 = 2^6$, si ha $\lfloor \log_2 37 \rfloor = 5$, e

$$37 = (100101)_2, \quad |I| = \lfloor \log_2 37 \rfloor + 1 = 6.$$

- Due numeri naturali, $n = 37$ e $m = 200$. Poiché $2^7 = 128 \leq 200 < 256 = 2^8$, il numero $200 = (11001000)_2$ richiede 8 bit, e

$$|I| = 6 + 8 = 14.$$

- L'array $a = [6, 1, 3, 8]$. Le sue componenti sono $6 = (110)_2$, $1 = (1)_2$, $3 = (11)_2$ e $8 = (1000)_2$, e

$$|I| = 3 + 1 + 2 + 4 = 10.$$

La dimensione esatta dell'input dipende dai valori specifici dell'istanza. Nell'analisi degli algoritmi ci interessa come essa *cresce*, e per questo usiamo la notazione asintotica.

11.2 Notazione asintotica: un richiamo

La notazione asintotica è definita e studiata nel capitolo 9 «Notazione asintotica». Qui richiamiamo soltanto le definizioni e i due fatti che si usano in questo capitolo. Come là, f e g sono funzioni definite sui numeri naturali $n \geq 1$ e a valori reali non negativi, e la notazione confronta i loro ordini di crescita per tutti gli n sufficientemente grandi.

- $f(n) = O(g(n))$ (**limite superiore**) se esistono $c \in \mathbb{R}_{>0}$ e $n_1 \in \mathbb{N}$ tali che

$$0 \leq f(n) \leq c g(n), \quad \forall n \geq n_1;$$

- $f(n) = \Omega(g(n))$ (**limite inferiore**) se esistono $c \in \mathbb{R}_{>0}$ e $n_1 \in \mathbb{N}$ tali che

$$0 \leq c g(n) \leq f(n), \quad \forall n \geq n_1;$$

- $f(n) = \Theta(g(n))$ (**entrambi**) se esistono $c_1, c_2 \in \mathbb{R}_{>0}$ e $n_1 \in \mathbb{N}$ tali che

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \quad \forall n \geq n_1.$$

- Per dimostrare una di queste affermazioni **si determinano** le costanti (c , oppure c_1 e c_2) e la soglia n_1 .
- Per funzioni di due parametri, come $f(n, m)$, le definizioni sono le stesse, con le disuguaglianze richieste per ogni $n \geq n_1$ e ogni $m \geq m_1$.

- I due fatti seguenti sono dimostrati nel capitolo 9 «Notazione asintotica».

Osservazione 11.2: Θ equivale a O e Ω

Si ha $f(n) = \Theta(g(n))$ se e solo se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.

Proposizione 11.3: criterio del limite

Sia $g(n) > 0$ per ogni $n \geq 1$. Se

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = L \in \mathbb{R}_{>0},$$

allora $f(n) = \Theta(g(n))$.

11.3 Istanza di input costituita da un numero naturale

Partiamo dall'input più semplice: un solo numero naturale.

- Se l'input è costituito da un solo numero naturale $n \geq 1$, la dimensione dell'input è

$$|I| = \lfloor \log_2 n \rfloor + 1,$$

che è il numero minimo di bit necessari per rappresentare n in binario.

Osservazione 11.4: dimensione dell'input di un numero naturale

Si ha

$$\lfloor \log_2 n \rfloor + 1 = \Theta(\log_2 n). \quad (11.1)$$

Dimostrazione. Per dimostrare il risultato, determiniamo $c_1, c_2 \in \mathbb{R}_{>0}$ e $n_1 \in \mathbb{N}$ tali che

$$0 \leq c_1 \log_2 n \leq \lfloor \log_2 n \rfloor + 1 \leq c_2 \log_2 n, \quad \forall n \geq n_1.$$

Poiché $x < \lfloor x \rfloor + 1$ per ogni numero reale x , si ha

$$\log_2 n \leq \lfloor \log_2 n \rfloor + 1, \quad \forall n \geq 1.$$

Inoltre, poiché $\lfloor x \rfloor \leq x$ per ogni numero reale x , e $\log_2 n \geq 1$ per ogni $n \geq 2$, si ha

$$\lfloor \log_2 n \rfloor + 1 \leq \log_2 n + 1 \leq 2 \log_2 n, \quad \forall n \geq 2.$$

Quindi, scegliendo $c_1 = 1$, $c_2 = 2$ e $n_1 = 2$, otteniamo

$$0 \leq \log_2 n \leq \lfloor \log_2 n \rfloor + 1 \leq 2 \log_2 n, \quad \forall n \geq 2.$$

□

Se l'input di un algoritmo è costituito da un solo numero naturale $n \geq 1$, la dimensione dell'input soddisfa:

$$|I| = \Theta(\log_2 n).$$

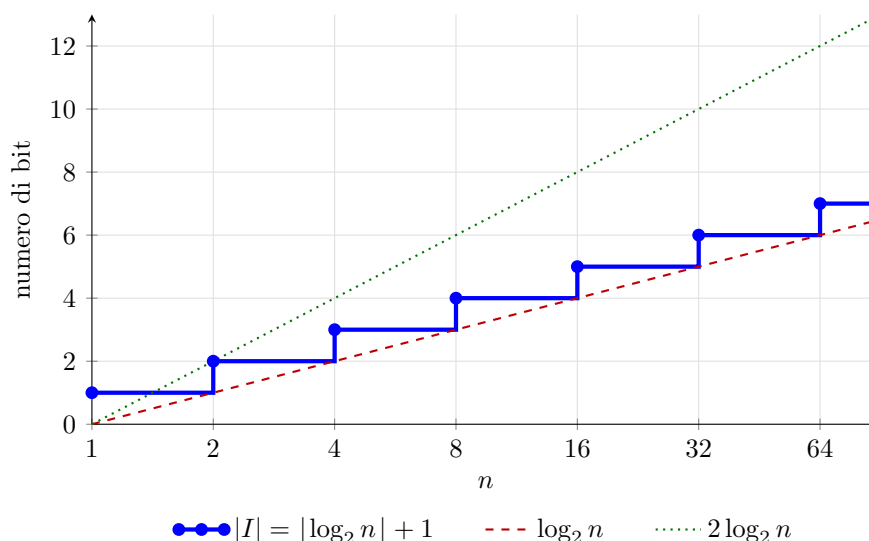


Figura 11.1: La dimensione dell'input $|I| = \lfloor \log_2 n \rfloor + 1$ di un numero naturale n (asse orizzontale in scala logaritmica): aumenta di un bit a ogni potenza di 2, e per $n \geq 2$ è compresa fra $\log_2 n$ e $2 \log_2 n$, come afferma l'Osservazione 11.4.

Esempio 11.2: i limiti per un numero naturale

Per $n = 37$ si ha $|I| = 6$ (Esempio 11.1) e $\log_2 37 \approx 5.21$. Le disuguaglianze della dimostrazione dell'Osservazione 11.4 diventano

$$\log_2 37 \approx 5.21 \leq |I| = 6 \leq 2 \log_2 37 \approx 10.42.$$

Per $n = 1$ si ha $|I| = 1$ ma $\log_2 1 = 0$: per questo la soglia della dimostrazione è $n_1 = 2$.

11.4 Istanza di input costituita da due numeri naturali

Quando l'input è costituito da due numeri, vanno scritti i bit di entrambi.

- Se l'input è costituito da due numeri naturali $n, m \geq 1$, la dimensione dell'input è

$$|I| = (\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1),$$

che è il numero minimo di bit necessari per rappresentare in binario sia n sia m .

Osservazione 11.5: dimensione dell'input di due numeri naturali

Si ha

$$(\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1) = \Theta(\log_2 n + \log_2 m). \quad (11.2)$$

Dimostrazione. Per dimostrare il risultato, determiniamo $c_1, c_2 \in \mathbb{R}_{>0}$ e $n_1, m_1 \in \mathbb{N}$ tali che

$$0 \leq c_1(\log_2 n + \log_2 m) \leq (\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1) \leq c_2(\log_2 n + \log_2 m)$$

per ogni $n \geq n_1$ e $m \geq m_1$.

Abbiamo

$$\log_2 n \leq \lfloor \log_2 n \rfloor + 1, \quad \log_2 m \leq \lfloor \log_2 m \rfloor + 1, \quad \forall n, m \geq 1.$$

Quindi

$$\log_2 n + \log_2 m \leq (\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1), \quad \forall n, m \geq 1.$$

Inoltre

$$\lfloor \log_2 n \rfloor + 1 \leq \log_2 n + 1, \quad \lfloor \log_2 m \rfloor + 1 \leq \log_2 m + 1.$$

Dunque

$$(\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1) \leq (\log_2 n + 1) + (\log_2 m + 1).$$

Se $n, m \geq 2$, allora $\log_2 n \geq 1$ e $\log_2 m \geq 1$, quindi

$$\log_2 n + 1 \leq 2 \log_2 n, \quad \log_2 m + 1 \leq 2 \log_2 m.$$

Pertanto

$$(\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1) \leq 2(\log_2 n + \log_2 m), \quad \forall n, m \geq 2.$$

Quindi, scegliendo $c_1 = 1$, $c_2 = 2$, $n_1 = 2$ e $m_1 = 2$, otteniamo

$$0 \leq \log_2 n + \log_2 m \leq (\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1) \leq 2(\log_2 n + \log_2 m), \quad \forall n, m \geq 2.$$

□

Se l'input di un algoritmo è costituito da due numeri naturali $n, m \geq 1$, la dimensione dell'input soddisfa:

$$|I| = \Theta(\underbrace{\log_2 n + \log_2 m}_{\log_2(nm)}).$$

La somma dei due logaritmi può essere sostituita dal logaritmo del maggiore dei due numeri: il numero maggiore richiede almeno metà dei bit dell'istanza.

Osservazione 11.6: dimensione dell'input di due numeri naturali e loro massimo

Si ha

$$(\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1) = \Theta(\log_2 M), \quad (11.3)$$

dove

$$M = \max\{n, m\}.$$

Dimostrazione. Per dimostrare il risultato, determiniamo $c_1, c_2 \in \mathbb{R}_{>0}$ e $n_1, m_1 \in \mathbb{N}$ tali che

$$0 \leq c_1 \log_2 M \leq (\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1) \leq c_2 \log_2 M, \quad \forall n \geq n_1, m \geq m_1.$$

Poiché uno dei due numeri è uguale a M , si ha

$$\lfloor \log_2 M \rfloor + 1 \leq (\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1).$$

Inoltre,

$$\log_2 M \leq \lfloor \log_2 M \rfloor + 1, \quad \forall M \geq 1.$$

Quindi,

$$\log_2 M \leq (\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1), \quad \forall n, m \geq 1.$$

D'altra parte, poiché $n \leq M$ e $m \leq M$, si ha

$$\lfloor \log_2 n \rfloor + 1 \leq \lfloor \log_2 M \rfloor + 1, \quad \lfloor \log_2 m \rfloor + 1 \leq \lfloor \log_2 M \rfloor + 1.$$

Dunque

$$(\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1) \leq 2(\lfloor \log_2 M \rfloor + 1).$$

Se $M \geq 2$, allora $\log_2 M \geq 1$, quindi

$$\lfloor \log_2 M \rfloor + 1 \leq \log_2 M + 1 \leq 2 \log_2 M.$$

Pertanto,

$$(\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1) \leq 4 \log_2 M, \quad \forall M \geq 2.$$

Quindi, scegliendo $c_1 = 1$, $c_2 = 4$, $n_1 = 2$ e $m_1 = 2$ (che danno $M \geq 2$), otteniamo

$$0 \leq \log_2 M \leq (\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1) \leq 4 \log_2 M, \quad \forall n, m \geq 2.$$

□

Se l'input di un algoritmo è costituito da due numeri naturali $n, m \geq 1$, la dimensione dell'input soddisfa:

$$|I| = \Theta(\log_2 M),$$

dove

$$M = \max\{n, m\}.$$

Esempio 11.3: i limiti per due numeri naturali

Per $n = 37$ e $m = 200$ si ha $|I| = 14$ (Esempio 11.1), $M = 200$, e

$$\log_2 37 + \log_2 200 \approx 12.85, \quad \log_2 200 \approx 7.64.$$

Le disuguaglianze delle dimostrazioni delle Osservazioni 11.5 e 11.6 diventano

$$12.85 \leq 14 \leq 2 \cdot 12.85 = 25.71 \quad \text{e} \quad 7.64 \leq 14 \leq 4 \cdot 7.64 = 30.58,$$

dove tutti i valori sono arrotondati a due cifre decimali.

11.5 Istanza di input costituita da un array di numeri naturali

La maggior parte degli algoritmi del corso riceve un array. La sua dimensione dell'input dipende sia dal numero n di componenti sia dai loro valori.

- Se l'istanza di input è costituita da un array $a[1..n]$ di n numeri naturali, con $a[i] \geq 1$ per ogni $i \in \{1, 2, \dots, n\}$, la dimensione dell'input è

$$|I| = \sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1),$$

che è il numero minimo di bit necessari per rappresentare in binario tutte le componenti dell'array.

11.5.1 Un limite superiore

Osservazione 11.7: limite superiore sulla dimensione dell'input di un array

Dato un array $a[1..n]$ di n numeri naturali, con $a[i] \geq 1$ per ogni $i \in \{1, 2, \dots, n\}$, si ha

$$\sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1) = O(n(1 + \log_2 M)), \quad (11.4)$$

dove

$$M = \max \{a[1], a[2], \dots, a[n]\}.$$

Dimostrazione. Per dimostrare il risultato, determiniamo $c \in \mathbb{R}_{>0}$ e $n_1 \in \mathbb{N}$ tali che

$$0 \leq \sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1) \leq cn(1 + \log_2 M), \quad \forall n \geq n_1.$$

Poiché $a[i] \leq M$ per ogni $i \in \{1, 2, \dots, n\}$, si ha

$$\lfloor \log_2 a[i] \rfloor + 1 \leq \lfloor \log_2 M \rfloor + 1, \quad \forall i \in \{1, 2, \dots, n\}.$$

Quindi,

$$\sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1) \leq \sum_{i=1}^n (\lfloor \log_2 M \rfloor + 1) = n(\lfloor \log_2 M \rfloor + 1).$$

Inoltre, si ha

$$\lfloor \log_2 M \rfloor + 1 \leq \log_2 M + 1.$$

Dunque,

$$\sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1) \leq n(1 + \log_2 M), \quad \forall n \geq 1.$$

Quindi, scegliendo $c = 1$ e $n_1 = 1$, otteniamo

$$0 \leq \sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1) \leq n(1 + \log_2 M), \quad \forall n \geq 1.$$

□

Se l'input di un algoritmo è costituito da un array $a[1..n]$ di n numeri naturali, con $a[i] \geq 1$ per ogni $i \in \{1, 2, \dots, n\}$, allora la dimensione dell'input soddisfa

$$|I| = O(n(1 + \log_2 M)),$$

dove

$$M = \max \{a[1], a[2], \dots, a[n]\}.$$

Il limite superiore $n(1 + \log_2 M)$ conta ogni componente come se fosse grande quanto il massimo. Quando solo poche componenti sono grandi, è una forte sovrastima, e il limite non può essere sostituito da Θ .

Esempio 11.4: il limite superiore non è stretto

Dato un array $a[1..n]$ di n numeri naturali, con $a[i] \geq 1$ per ogni $i \in \{1, 2, \dots, n\}$, per dimostrare che, in generale,

$$\sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1) \neq \Theta(n(1 + \log_2 M)), \quad M = \max\{a[1], a[2], \dots, a[n]\},$$

è sufficiente fornire un controesempio.

- Consideriamo la famiglia di array definita da

$$a[1] = M, \quad a[2] = a[3] = \cdots = a[n] = 1,$$

con $M \geq 2$.

- Per questi array, la dimensione dell'input è

$$|I| = \sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1) = (\lfloor \log_2 M \rfloor + 1) + (n-1)(\lfloor \log_2 1 \rfloor + 1).$$

Poiché $\lfloor \log_2 1 \rfloor + 1 = 1$, otteniamo

$$|I| = (\lfloor \log_2 M \rfloor + 1) + (n-1) = n + \lfloor \log_2 M \rfloor.$$

- Scegliamo ora $M = 2^n$, in modo che $\log_2 M = n$. Allora

$$|I| = n + n = 2n, \quad \text{mentre} \quad n(1 + \log_2 M) = n(1 + n) = n^2 + n.$$

- Se avessimo $2n = \Theta(n^2 + n)$, esisterebbero $c_1 \in \mathbb{R}_{>0}$ e $n_1 \in \mathbb{N}$ tali che $c_1(n^2 + n) \leq 2n$ per ogni $n \geq n_1$, cioè $c_1(n+1) \leq 2$ per ogni $n \geq n_1$, il che è falso per n abbastanza grande.

Quindi, per questa famiglia di array, $|I| \neq \Theta(n(1 + \log_2 M))$.

- Se $M \geq 2$, allora $\log_2 M \geq 1$, e quindi $1 + \log_2 M \leq 2 \log_2 M$. Dunque

$$n(1 + \log_2 M) \leq 2n \log_2 M,$$

e la dimensione dell'input soddisfa

$$|I| = O(n \log_2 M).$$

- Se $M = 1$, allora tutte le componenti dell'array sono uguali a 1, e quindi

$$|I| = \sum_{i=1}^n (\lfloor \log_2 1 \rfloor + 1) = n.$$

11.5.2 Un limite inferiore

Ogni componente dell'array richiede almeno un bit. Questa semplice osservazione dà un limite inferiore che vale per ogni array.

Osservazione 11.8: limite inferiore sulla dimensione dell'input di un array

Dato un array $a[1..n]$ di n numeri naturali, con $a[i] \geq 1$ per ogni $i \in \{1, 2, \dots, n\}$, si ha

$$\sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1) = \Omega(n). \quad (11.5)$$

Dimostrazione. Per dimostrare il risultato, determiniamo $c \in \mathbb{R}_{>0}$ e $n_1 \in \mathbb{N}$ tali che

$$0 \leq cn \leq \sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1), \quad \forall n \geq n_1.$$

Poiché $a[i] \geq 1$ per ogni $i \in \{1, 2, \dots, n\}$, si ha

$$\lfloor \log_2 a[i] \rfloor + 1 \geq \lfloor \log_2 1 \rfloor + 1 = 1, \quad \forall i \in \{1, 2, \dots, n\}.$$

Quindi,

$$\sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1) \geq \sum_{i=1}^n 1 = n.$$

Quindi, scegliendo $c = 1$ e $n_1 = 1$, otteniamo

$$0 \leq n \leq \sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1), \quad \forall n \geq 1.$$

□

Se l'input di un algoritmo è costituito da un array $a[1..n]$ di n numeri naturali, con $a[i] \geq 1$ per ogni $i \in \{1, 2, \dots, n\}$, allora la dimensione dell'input soddisfa

$$|I| = \Omega(n).$$

Esempio 11.5: i limiti per un array

Per l'array $a = [6, 1, 3, 8]$ dell'Esempio 11.1 si ha $n = 4$, $|I| = 10$ e $M = 8$. Il limite inferiore dell'Osservazione 11.8 e il limite superiore dell'Osservazione 11.7 danno

$$n = 4 \leq |I| = 10 \leq n(1 + \log_2 M) = 4(1 + 3) = 16.$$

11.5.3 Componenti di dimensione costante

In molte applicazioni le componenti dell'array sono memorizzate con un numero fisso di bit, indipendente da n . In questo caso i due limiti coincidono.

Corollario 11.9: componenti di dimensione costante

Dato un array $a[1..n]$ di n numeri naturali, con $a[i] \geq 1$ per ogni $i \in \{1, 2, \dots, n\}$, se tutte le componenti dell'array possono essere rappresentate usando al più un numero costante fissato di bit, allora

$$|I| = \Theta(n).$$

Dimostrazione. Per ipotesi esiste una costante $c \in \mathbb{N}$ tale che

$$\lfloor \log_2 a[i] \rfloor + 1 \leq c, \quad \forall i \in \{1, 2, \dots, n\}.$$

Quindi,

$$|I| = \sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1) \leq \sum_{i=1}^n c = cn, \quad \forall n \geq 1,$$

e dunque $|I| = O(n)$. Per l'Osservazione 11.8 si ha anche $|I| = \Omega(n)$, e quindi, per l'Osservazione 11.2,

$$|I| = \Theta(n).$$

□

11.5.4 Due varianti dell'input array

- Se l'input è costituito da un numero naturale $n \geq 1$ e da un array $a[1..n]$ di n numeri naturali, con $a[i] \geq 1$ per ogni $i \in \{1, 2, \dots, n\}$, la dimensione dell'input è

$$|I| = (\lfloor \log_2 n \rfloor + 1) + \sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1).$$

Sia

$$M = \max\{n, a[1], a[2], \dots, a[n]\}.$$

Allora si ha

$$\lfloor \log_2 n \rfloor + 1 \leq 1 + \log_2 M,$$

e, per ogni $i \in \{1, 2, \dots, n\}$,

$$\lfloor \log_2 a[i] \rfloor + 1 \leq 1 + \log_2 M.$$

Quindi,

$$|I| \leq (1 + \log_2 M) + n(1 + \log_2 M) = (n + 1)(1 + \log_2 M).$$

Dunque

$$|I| = O((n + 1)(1 + \log_2 M)).$$

Poiché $n + 1 \leq 2n$ per ogni $n \geq 1$, otteniamo anche

$$|I| = O(n(1 + \log_2 M)).$$

Inoltre, se $M \geq 2$, allora

$$1 + \log_2 M \leq 2 \log_2 M,$$

e quindi

$$|I| = O(n \log_2 M).$$

Il limite inferiore $|I| = \Omega(n)$ continua a valere, poiché i soli bit dell'array sono almeno n .

- Se le componenti dell'array possono essere 0, la dimensione dell'input è

$$|I| = \sum_{i=1}^n (\lfloor \log_2(\max\{1, a[i]\}) \rfloor + 1),$$

così che ogni componente uguale a 0 contribuisce esattamente con 1 bit. In questo caso, se poniamo

$$M = \max\{1, a[1], a[2], \dots, a[n]\},$$

le formule precedenti diventano

$$|I| = O(n(1 + \log_2 M)),$$

e, se $M \geq 2$,

$$|I| = O(n \log_2 M).$$

Inoltre, continua a valere

$$|I| = \Omega(n),$$

poiché ogni componente contribuisce con almeno 1 bit.

In particolare, se tutte le componenti dell'array possono essere rappresentate usando al più un numero costante fissato di bit, allora

$$|I| = \Theta(n).$$

11.6 Dimensione dell'input e numero di passi

Possiamo ora spiegare perché la dimensione dell'input si misura in bit. Il tempo richiesto da un algoritmo si stima contando i suoi **passi**: nello pseudocodice, un passo è l'esecuzione di una riga, e la condizione di un ciclo **while** conta come un passo ogni volta che viene valutata. Il numero di passi viene calcolato prima come funzione $S(n)$ di un parametro naturale n dell'input, e poi espresso come funzione della dimensione dell'input $|I|$.

Definizione 11.10: tempo di esecuzione nel caso peggiore

Il **tempo di esecuzione nel caso peggiore** $T(|I|)$ di un algoritmo è il massimo numero di passi eseguiti dall'algoritmo su un qualsiasi input I di una data dimensione $|I|$.

Definizione 11.11: algoritmi polinomiali ed esponenziali

Un algoritmo è **polinomiale** se il suo tempo di esecuzione nel caso peggiore soddisfa $T(|I|) = O(|I|^k)$ per una qualche costante $k \in \mathbb{N}$.

Un algoritmo è **esponenziale** se il suo tempo di esecuzione nel caso peggiore soddisfa $T(|I|) = 2^{\Theta(|I|)}$, cioè se esistono $c_1, c_2 \in \mathbb{R}_{>0}$ e $k_1 \in \mathbb{N}$ tali che

$$2^{c_1|I|} \leq T(|I|) \leq 2^{c_2|I|}, \quad \forall |I| \geq k_1.$$

Il parametro n può avere due ruoli molto diversi. Può essere il **valore** di un numero dato in input, oppure il **numero di componenti** di un array dato in input. L'osservazione seguente mostra come n è legato a $|I|$ nei due casi.

Osservazione 11.12: valore di un numero e numero di componenti di un array

1. Se l'input è costituito da un solo numero naturale $n \geq 1$, allora

$$2^{|I|-1} \leq n \leq 2^{|I|} - 1 :$$

il valore n è esponenziale nella dimensione dell'input.

2. Se l'input è costituito da un array $a[1..n]$ di n numeri naturali (ed eventualmente dal numero n), allora

$$n \leq |I| :$$

il numero di componenti n è al più lineare nella dimensione dell'input.

Dimostrazione. 1. Poiché $|I| = \lfloor \log_2 n \rfloor + 1$, si ha $\lfloor \log_2 n \rfloor = |I| - 1$, cioè

$$|I| - 1 \leq \log_2 n < |I|.$$

La funzione $x \mapsto 2^x$ è strettamente crescente, quindi $2^{|I|-1} \leq n < 2^{|I|}$. Poiché n è un intero, $n \leq 2^{|I|} - 1$.

2. Ogni componente dell'array contribuisce con almeno un bit alla dimensione dell'input (Osservazione 11.8, e lo stesso vale per le componenti uguali a 0), quindi $|I| \geq n$.

□

Quando un numero di passi è limitato da esponenziali con fattori costanti, i fattori costanti possono essere assorbiti nell'esponente.

Osservazione 11.13: fattori costanti nell'esponente

Siano $a_1, a_2, b_1, b_2 \in \mathbb{R}_{>0}$. Se una funzione T soddisfa

$$a_1 2^{b_1 x} \leq T(x) \leq a_2 2^{b_2 x}$$

per ogni numero naturale x sufficientemente grande, allora $T(x) = 2^{\Theta(x)}$.

Dimostrazione. Scriviamo i fattori costanti come potenze di 2: $a_1 = 2^{\log_2 a_1}$ e $a_2 = 2^{\log_2 a_2}$. Allora

$$a_2 2^{b_2 x} = 2^{b_2 x + \log_2 a_2} \leq 2^{2b_2 x} \quad \text{ogni volta che} \quad b_2 x \geq \log_2 a_2,$$

e

$$a_1 2^{b_1 x} = 2^{b_1 x + \log_2 a_1} \geq 2^{\frac{b_1}{2} x} \quad \text{ogni volta che} \quad \frac{b_1}{2} x \geq -\log_2 a_1.$$

Entrambe le condizioni valgono per ogni x sufficientemente grande. Quindi, con $c_1 = \frac{b_1}{2}$ e $c_2 = 2b_2$, otteniamo $2^{c_1 x} \leq T(x) \leq 2^{c_2 x}$ per ogni x sufficientemente grande, cioè $T(x) = 2^{\Theta(x)}$. $\square$

Esempio 11.6: la somma delle componenti di un array

Consideriamo il seguente algoritmo, che calcola la somma delle componenti di un array.

Algorithm: SumArray

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

Output: $a[1] + a[2] + \dots + a[n]$

```

1 function SumArray(a, n)
2   s := 0
3   i := 1
4   while i ≤ n do
5     s ← s + a[i]
6     i ← i + 1
7   end
8   return s
9 end

```

- **Passo 1: numero di passi.** Le righe 2, 3 e 8 sono eseguite 1 volta ciascuna; la condizione alla riga 4 è valutata $n + 1$ volte (l'ultima volta è falsa); le righe 5 e 6 sono eseguite n volte ciascuna. Quindi

$$S(n) = 3 + (n + 1) + 2n = 3n + 4,$$

per ogni array di n componenti, qualunque siano i suoi valori.

- **Passo 2: da n a $|I|$.** Per l'Osservazione 11.12, ogni input di dimensione $|I|$ ha $n \leq |I|$ componenti, quindi richiede al più $3|I| + 4$ passi. L'array di $|I|$ componenti tutte uguali a 1 ha dimensione dell'input $|I|$ e richiede esattamente $3|I| + 4$ passi. Dunque

$$T(|I|) = 3|I| + 4 = O(|I|).$$

- **Passo 3: conclusione.** SumArray è polinomiale (lineare) nella dimensione dell'input.

Se invece lo stesso ciclo andasse da 1 al *valore* di un numero n dato in input, i $\Theta(n)$ passi sarebbero esponenziali nella dimensione dell'input, perché n può arrivare a $2^{|I|} - 1$ (Osservazione 11.12).

Punti chiave del capitolo

- La dimensione dell'input $|I|$ è il numero di bit necessari per scrivere in binario l'istanza di input.
- Un numero naturale n : $|I| = \lfloor \log_2 n \rfloor + 1 = \Theta(\log_2 n)$. Due numeri naturali n, m : $|I| = \Theta(\log_2 n + \log_2 m) = \Theta(\log_2 \max\{n, m\})$.
- Un array di n numeri naturali: $|I| = \Omega(n)$ e $|I| = O(n(1 + \log_2 M))$, con M la componente massima; il limite superiore in generale non è stretto. Se le componenti hanno un numero costante di bit, $|I| = \Theta(n)$.
- Lo stesso numero di passi $S(n)$ può essere polinomiale o esponenziale in $|I|$: il valore di un numero è esponenziale nella sua dimensione dell'input, mentre il numero di componenti di un array è al più la sua dimensione dell'input.

Tempo di esecuzione

Nel capitolo «Dimensione dell'input» abbiamo imparato a misurare la dimensione $|I|$ di un'istanza di input I , cioè il numero di bit necessari per rappresentarla. In questo capitolo usiamo tale misura per rispondere alla domanda centrale dell'analisi degli algoritmi: **come cresce il tempo richiesto da un algoritmo quando cresce l'input?**

Il piano del capitolo è il seguente.

- Nella Sezione 12.1 definiamo il tempo di esecuzione di un algoritmo, le operazioni elementari (**passi**) che esso esegue e il **caso peggiore**. Fissiamo inoltre le convenzioni con cui si contano, riga per riga, i passi di uno pseudocodice.
- Nelle Sezioni 12.2–12.5 applichiamo il metodo a quattro algoritmi già incontrati nei capitoli precedenti: `FindMaximum`, `InsertionSort`, `ChangeToBase2` e `PrimalityTest`. Per ciascuno di essi contiamo prima il numero di passi $S(n)$ in funzione di un parametro naturale n , e poi esprimiamo l'ordine di crescita in funzione della dimensione dell'input $|I|$.
- Le quattro analisi portano a quattro comportamenti diversi (lineare, quadratico, lineare in $|I|$ benché logaritmico in n , esponenziale in $|I|$ benché solo $\Theta(\sqrt{n})$ in n), che vengono confrontati nella Sezione 12.6.

12.1 Tempo di esecuzione di un algoritmo

Definizione 12.1: tempo di esecuzione

Il **tempo di esecuzione** di un algoritmo è una stima del tempo necessario per eseguirlo, in funzione della **dimensione dell'input** $|I|$.

- Non ci interessa il tempo esatto in secondi, che dipende dal calcolatore, dal compilatore e da molti altri dettagli, ma l'**ordine di crescita** del tempo di esecuzione, espresso con la notazione asintotica O , Ω e Θ (capitolo 9 «Notazione asintotica», richiamata nel capitolo «Dimensione dell'input»).
- Il tempo di esecuzione di un algoritmo dipende dal numero di **operazioni elementari** che l'algoritmo esegue su un input di una data dimensione.

12.1.1 Operazioni elementari

Definizione 12.2: operazione elementare

Un'**operazione elementare** è una singola operazione che può essere eseguita da un calcolatore.

- Le principali operazioni elementari sono le seguenti:
 1. operazioni aritmetiche su numeri di dimensione fissata;
 2. confronto di due numeri di dimensione fissata;
 3. assegnamento di un valore a una variabile;
 4. accesso a un elemento di un array tramite indice;
 5. operazioni di controllo del flusso (per esempio, la valutazione della condizione di un'istruzione `if`, oppure l'iterazione di un ciclo).

Assumiamo che per eseguire ciascuna operazione elementare di un algoritmo sia richiesta una **quantità di tempo costante**. In questo modo l'analisi risulta il più possibile indipendente dalla macchina.

- Per brevità, chiamiamo **passo** ogni operazione elementare.

12.1.2 Contare i passi di uno pseudocodice

- Ogni riga degli pseudocodici di queste dispense esegue un numero **costante** di operazioni elementari (qualche confronto, operazione aritmetica e assegnamento). Poiché ci interessa l'ordine di crescita, contiamo **un passo per ogni esecuzione di una riga**: questo modifica il numero di passi solo per un fattore costante.
- Per contare i passi si determina, per ogni riga, quante volte viene eseguita, e si sommano questi numeri. Le convenzioni seguenti sono usate in tutto il capitolo (e nei suoi esercizi).

Conteggio riga per riga.

- Ogni esecuzione di una riga numerata è un **passo**, comprese le righe **return**.
- La condizione di un ciclo **while** il cui corpo viene eseguito k volte viene valutata $k+1$ volte: l'ultima valutazione è quella che trova la condizione falsa e termina il ciclo.
- La condizione di un **if** viene valutata ogni volta che si raggiunge la riga; le righe del ramo che non viene percorso non vengono eseguite.
- Quando il numero di esecuzioni di una riga dipende dallo specifico input (e non soltanto dalla sua dimensione), lo descriviamo con una funzione ausiliaria, di solito indicata con t , e poi cerchiamo l'input che rende il numero totale di passi il più grande possibile.

12.1.3 Tempo di esecuzione nel caso peggiore

- Anche per input di una data dimensione, il tempo di esecuzione di un algoritmo può dipendere da quale specifico input di quella dimensione viene fornito.
- Tipicamente, nell'analisi degli algoritmi si considera il caso peggiore possibile, perché esso determina il tempo di esecuzione più lungo per qualunque input di una data dimensione.

Definizione 12.3: tempo di esecuzione nel caso peggiore

Il **tempo di esecuzione nel caso peggiore** $T(|I|)$ di un algoritmo è il massimo numero di passi eseguiti dall'algoritmo su un qualunque input I di una data dimensione $|I|$.

- Di solito è più facile contare i passi in funzione di un **parametro naturale** dell'input, come il numero n di elementi di un array, o il valore n del numero in input. La dimensione dell'input $|I|$, invece, è il numero di bit dell'input. Questo porta al metodo seguito in tutti gli esempi del capitolo.

I due passi dell'analisi.

1. Contare il numero di passi $S(n)$ eseguiti nel caso peggiore, in funzione del parametro naturale n , e determinarne l'ordine di crescita (per esempio $S(n) = \Theta(n^2)$).
2. Determinare la relazione tra n e la dimensione dell'input $|I|$ (per esempio $|I| = \Theta(n)$, oppure $|I| = \Theta(\log_2 n)$), e usarla per esprimere l'ordine di crescita del tempo di esecuzione nel caso peggiore $T(|I|)$.

- Il secondo passo non è una formalità: come vedremo, lo stesso numero di passi $\Theta(n)$ corrisponde a un tempo di esecuzione lineare in $|I|$ se n è il numero di elementi di un array, ed esponenziale in $|I|$ se n è il valore del numero in input.

12.2 Ricerca del massimo

Dato un array $a[1..n]$ di $n \geq 1$ numeri interi, consideriamo il problema di determinare il valore massimo nell'array.

- Questo è lo pseudocodice di un semplice algoritmo, chiamato **FindMaximum**, che risolve il problema.

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    $i := 2$ 
4   while  $i \leq n$  do
5     if  $a[i] > max$  then
6        $max \leftarrow a[i]$ 
7     end
8      $i \leftarrow i + 1$ 
9   end
10  return  $max$ 
11 end

```

12.2.1 Numero di passi

- Sia $S(n)$ il numero di passi eseguiti da FindMaximum nel caso peggiore, dove n è la dimensione dell'array.
- Il corpo del ciclo viene eseguito per $i = 2, 3, \dots, n$, cioè $n-1$ volte, quindi la condizione della riga 4 viene valutata n volte. L'esecuzione della riga 6, invece, dipende dall'iterazione. Definiamo una funzione

$$t : \{2, 3, \dots, n\} \rightarrow \{0, 1\},$$

dove $t(i) = 1$ se la riga 6 viene eseguita all'iterazione i , e $t(i) = 0$ altrimenti.

- Il numero di volte in cui ciascuna riga di codice viene eseguita è:

riga	numero di esecuzioni
2	1
3	1
4	n
5	$n - 1$
6	$\sum_{i=2}^n t(i)$
8	$n - 1$
10	1

- Quindi il numero totale di passi è

$$1 + 1 + n + (n - 1) + \sum_{i=2}^n t(i) + (n - 1) + 1 = 3n + 1 + \sum_{i=2}^n t(i).$$

- Il valore di $t(i)$ dipende dallo specifico array in input, quindi analizziamo il caso peggiore possibile. Il caso peggiore possibile si ha quando gli elementi dell'array sono in ordine crescente, cioè

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

Allora $t(i) = 1$ per ogni $i \in \{2, 3, \dots, n\}$: entriamo nell'**if** a ogni iterazione e aggiorniamo ogni volta il valore massimo. Il numero totale di passi nel caso peggiore è

$$3n + 1 + \sum_{i=2}^n 1 = 3n + 1 + (n - 1) = 4n.$$

- Quindi,

$$S(n) = 4n = \Theta(n).$$

Dunque $S(n)$ cresce linearmente in n . Infatti,

$$\lim_{n \rightarrow \infty} \frac{4n}{n} = 4 \in \mathbb{R}_{>0}.$$

Esempio 12.1: conteggio dei passi di FindMaximum

Consideriamo l'array $a[1..4] = (3, 1, 4, 5)$.

- Iterazione $i = 2$: $a[2] = 1 > 3$ è falsa, quindi $t(2) = 0$.
- Iterazione $i = 3$: $a[3] = 4 > 3$ è vera, max diventa 4, quindi $t(3) = 1$.
- Iterazione $i = 4$: $a[4] = 5 > 4$ è vera, max diventa 5, quindi $t(4) = 1$.

L'algoritmo restituisce 5 dopo

$$3 \cdot 4 + 1 + (t(2) + t(3) + t(4)) = 13 + 2 = 15$$

passi. Sull'array crescente $(1, 2, 3, 4)$, invece, $t(2) = t(3) = t(4) = 1$ e l'algoritmo esegue $S(4) = 4 \cdot 4 = 16$ passi, il massimo per $n = 4$.

12.2.2 Tempo di esecuzione in funzione della dimensione dell'input

- Determiniamo ora il tempo di esecuzione nel caso peggiore dell'algoritmo, esprimendo l'ordine di crescita del numero di passi in funzione della dimensione dell'input $|I|$.

Caso generale. Poiché l'input dell'algoritmo è costituito da un array $a[1..n]$ di n numeri interi, la dimensione dell'input soddisfa

$$|I| = \Omega(n),$$

perché l'input deve contenere almeno gli n elementi dell'array. Questo significa che, per una certa costante $c > 0$ e per ogni n sufficientemente grande,

$$|I| \geq cn.$$

Equivalentemente,

$$n \leq \frac{1}{c}|I|,$$

e quindi $n = O(|I|)$. Poiché $S(n) = \Theta(n)$, esiste una costante $a > 0$ tale che

$$S(n) \leq an$$

per ogni n sufficientemente grande. Poiché $n \leq \frac{1}{c}|I|$, otteniamo

$$S(n) \leq \frac{a}{c}|I|.$$

Dunque il numero di passi è limitato superiormente da una costante per la dimensione dell'input.

Il tempo di esecuzione nel caso peggiore dell'algoritmo **FindMaximum**, in funzione della dimensione dell'input $|I|$, è

$$T(|I|) = O(|I|).$$

Dunque, nel caso generale, il tempo di esecuzione nel caso peggiore è al più lineare nella dimensione dell'input.

Elementi con un numero fissato di bit. Spesso si fa l'ipotesi che tutti i valori dell'input possano essere memorizzati usando un numero costante fissato di bit, per esempio 32 bit. Sotto questa ipotesi, la dimensione dell'input soddisfa

$$|I| = \Theta(n),$$

perché l'input è costituito da n elementi dell'array e ciascun elemento richiede un numero costante di bit. Questo significa che esistono costanti $c_1, c_2 > 0$ tali che, per ogni n sufficientemente grande,

$$c_1 n \leq |I| \leq c_2 n.$$

Equivalentemente,

$$\frac{1}{c_2}|I| \leq n \leq \frac{1}{c_1}|I|,$$

cioè $n = \Theta(|I|)$. Poiché $S(n) = \Theta(n)$, esistono costanti $a_1, a_2 > 0$ tali che, per ogni n sufficientemente grande,

$$a_1 n \leq S(n) \leq a_2 n.$$

Usando i limiti su n in funzione di $|I|$, otteniamo

$$\frac{a_1}{c_2}|I| \leq S(n) \leq \frac{a_2}{c_1}|I|,$$

cioè $S(n) = \Theta(|I|)$.

Se tutti i valori dell'input usano un numero costante fissato di bit, allora il tempo di esecuzione nel caso peggiore dell'algoritmo **FindMaximum** è

$$T(|I|) = \Theta(|I|).$$

Dunque, sotto questa ipotesi, l'algoritmo è lineare nella dimensione dell'input.

12.3 Insertion sort

- In `FindMaximum` il corpo del ciclo esegue un numero costante di passi. In `InsertionSort` il corpo del ciclo esterno contiene un altro ciclo, il cui numero di iterazioni dipende dall'input: questo produce un numero quadratico di passi.

Dato un array $a[1..n]$ di n numeri naturali, consideriamo il problema di ordinare l'array in ordine non decrescente:

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

- Questo è lo pseudocodice dell'algoritmo `InsertionSort`.

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      $key := a[i]$ 
5      $j := i - 1$ 
6     while  $j \geq 1$  and  $a[j] > key$  do
7        $a[j + 1] \leftarrow a[j]$ 
8        $j \leftarrow j - 1$ 
9     end
10     $a[j + 1] \leftarrow key$ 
11     $i \leftarrow i + 1$ 
12  end
13 end

```

12.3.1 Numero di passi

- Sia $S(n)$ il numero di passi eseguiti da `InsertionSort` nel caso peggiore, dove n è la dimensione dell'array.
- L'esecuzione delle righe 7–8 dipende dall'iterazione. Definiamo una funzione

$$t : \{2, 3, \dots, n\} \rightarrow \{0, 1, \dots, n - 1\},$$

dove $t(i)$ indica il numero di volte in cui il corpo del ciclo interno (righe 7–8) viene eseguito durante l'iterazione i del ciclo esterno. La condizione del ciclo interno (riga 6) viene quindi valutata $t(i) + 1$ volte durante la stessa iterazione.

- Il numero di volte in cui ciascuna riga di codice viene eseguita è:

riga	numero di esecuzioni
2	1
3	n
4	$n - 1$
5	$n - 1$
6	$(n - 1) + \sum_{i=2}^n t(i)$
7	$\sum_{i=2}^n t(i)$
8	$\sum_{i=2}^n t(i)$
10	$n - 1$
11	$n - 1$

- Quindi il numero totale di passi è

$$1 + n + (n - 1) + (n - 1) + \left((n - 1) + \sum_{i=2}^n t(i) \right) + \sum_{i=2}^n t(i) + \sum_{i=2}^n t(i) + (n - 1) + (n - 1)$$

$$= 6n - 4 + 3 \sum_{i=2}^n t(i).$$

- Il valore di $t(i)$ dipende dallo specifico array in input, quindi analizziamo il caso peggiore possibile. Poiché j parte da $i - 1$ e decresce al più fino a 0, si ha sempre $t(i) \leq i - 1$. Il caso peggiore possibile si ha quando gli elementi dell'array sono in ordine strettamente decrescente, cioè

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

In questo caso, a ogni iterazione esterna i , l'elemento $key = a[i]$ è più piccolo di tutti gli elementi che lo precedono. Quindi il ciclo while interno viene eseguito fino in fondo:

$$t(i) = i - 1 \quad \forall i \in \{2, 3, \dots, n\}.$$

Perciò,

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

- Il numero totale di passi nel caso peggiore è

$$6n - 4 + 3 \cdot \frac{n(n - 1)}{2} = \frac{12n - 8 + 3n^2 - 3n}{2} = \frac{3n^2 + 9n - 8}{2}.$$

Quindi,

$$S(n) = \frac{3n^2 + 9n - 8}{2} = \Theta(n^2).$$

Dunque $S(n)$ cresce quadraticamente in n . Infatti,

$$\lim_{n \rightarrow \infty} \frac{3n^2 + 9n - 8}{2n^2} = \lim_{n \rightarrow \infty} \frac{1}{2} \left(3 + \frac{9}{n} - \frac{8}{n^2} \right) = \frac{3}{2} \in \mathbb{R}_{>0}.$$

Esempio 12.2: la funzione t di InsertionSort

Consideriamo l'array $a[1..6] = (5, 2, 4, 6, 1, 3)$. A ogni iterazione i , $t(i)$ è il numero di elementi di $a[1..i-1]$ più grandi di $key = a[i]$, cioè il numero di elementi spostati di una posizione verso destra.

i	key	$t(i)$	array dopo l'iterazione
2	2	1	(2, 5, 4, 6, 1, 3)
3	4	1	(2, 4, 5, 6, 1, 3)
4	6	0	(2, 4, 5, 6, 1, 3)
5	1	4	(1, 2, 4, 5, 6, 3)
6	3	3	(1, 2, 3, 4, 5, 6)

Abbiamo $\sum_{i=2}^6 t(i) = 9$, quindi l'algoritmo esegue

$$6 \cdot 6 - 4 + 3 \cdot 9 = 59$$

passi. Su un array decrescente di 6 elementi, invece, $\sum_{i=2}^6 t(i) = 1 + 2 + 3 + 4 + 5 = 15$ e l'algoritmo esegue

$$S(6) = \frac{3 \cdot 36 + 9 \cdot 6 - 8}{2} = 77$$

passi.

12.3.2 Tempo di esecuzione in funzione della dimensione dell'input

- Determiniamo ora il tempo di esecuzione nel caso peggiore dell'algoritmo, esprimendo l'ordine di crescita del numero di passi in funzione della dimensione dell'input $|I|$.

Caso generale. Poiché l'input dell'algoritmo è costituito da un array $a[1..n]$ di n numeri naturali, la dimensione dell'input soddisfa

$$|I| = \Omega(n),$$

perché l'input deve contenere almeno gli n elementi dell'array. Questo significa che, per una certa costante $c > 0$ e per ogni n sufficientemente grande,

$$|I| \geq cn.$$

Equivalentemente,

$$n \leq \frac{1}{c}|I|,$$

e quindi $n = O(|I|)$. Poiché $S(n) = \Theta(n^2)$, esiste una costante $a > 0$ tale che

$$S(n) \leq an^2$$

per ogni n sufficientemente grande. Poiché $n \leq \frac{1}{c}|I|$, otteniamo

$$S(n) \leq a \left(\frac{1}{c}|I| \right)^2 = \frac{a}{c^2}|I|^2.$$

Dunque il numero di passi è limitato superiormente da una costante per $|I|^2$.

Il tempo di esecuzione nel caso peggiore dell'algoritmo `InsertionSort`, in funzione della dimensione dell'input $|I|$, è

$$T(|I|) = O(|I|^2).$$

Dunque, nel caso generale, il tempo di esecuzione nel caso peggiore è al più quadratico nella dimensione dell'input.

Elementi con un numero fissato di bit. Spesso si fa l'ipotesi che tutti i valori dell'input possano essere memorizzati usando un numero costante fissato di bit, per esempio 32 bit. Sotto questa ipotesi, la dimensione dell'input soddisfa

$$|I| = \Theta(n),$$

perché l'input è costituito da n elementi dell'array e ciascun elemento richiede un numero costante di bit. Questo significa che esistono costanti $c_1, c_2 > 0$ tali che, per ogni n sufficientemente grande,

$$c_1 n \leq |I| \leq c_2 n.$$

Equivalentemente,

$$\frac{1}{c_2} |I| \leq n \leq \frac{1}{c_1} |I|.$$

Poiché $S(n) = \Theta(n^2)$, esistono costanti $a_1, a_2 > 0$ tali che, per ogni n sufficientemente grande,

$$a_1 n^2 \leq S(n) \leq a_2 n^2.$$

Usando i limiti su n in funzione di $|I|$, otteniamo

$$a_1 \left(\frac{1}{c_2} |I| \right)^2 \leq S(n) \leq a_2 \left(\frac{1}{c_1} |I| \right)^2, \quad \text{cioè} \quad \frac{a_1}{c_2^2} |I|^2 \leq S(n) \leq \frac{a_2}{c_1^2} |I|^2.$$

Dunque $S(n) = \Theta(|I|^2)$.

Se tutti i valori dell'input usano un numero costante fissato di bit, allora il tempo di esecuzione nel caso peggiore dell'algoritmo `InsertionSort` è

$$T(|I|) = \Theta(|I|^2).$$

Dunque, sotto questa ipotesi, l'algoritmo è quadratico nella dimensione dell'input.

12.4 Cambio di base

- Nei primi due esempi l'input è un array e il parametro naturale n è il numero dei suoi elementi. Consideriamo ora un algoritmo il cui input è un **singolo numero** n : il parametro n è ora un valore, e la relazione tra n e $|I|$ cambia completamente.

Dato un numero naturale $n \geq 1$ rappresentato in base 10, consideriamo il problema di convertire n in base 2.

- Questo è lo pseudocodice dell'algoritmo **ChangeToBase2**. È l'algoritmo **ChangeOfBase** del capitolo «Algoritmi numerici», con base $b = 2$. Le cifre binarie sono memorizzate in un array $d[0..nd - 1]$ indicizzato a partire da 0, in modo che $d[k]$ sia la cifra di peso 2^k .

Algorithm: ChangeToBase2

Input : Un numero naturale $n \geq 1$ in base 10

Output: Un array d di cifre binarie e il numero di cifre nd

```

1 function ChangeToBase2( $n$ )
2    $nd := \lfloor \log_2 n \rfloor + 1$ 
3    $d :=$  nuovo array[0.. $nd - 1$ ]
4    $k := 0$ 
5   while  $n > 0$  do
6      $q := \lfloor n/2 \rfloor$ 
7      $r := n \bmod 2$ 
8      $d[k] \leftarrow r$ 
9      $n \leftarrow q$ 
10     $k \leftarrow k + 1$ 
11  end
12  return  $d, nd$ 
13 end
```

12.4.1 Numero di passi

- Sia $S(n)$ il numero di passi eseguiti da **ChangeToBase2**, dove n è il valore in input da convertire. Per ogni valore di n c'è un solo input, quindi non c'è distinzione tra il caso peggiore e gli altri casi.
- Il numero di cifre necessarie per rappresentare n in base 2 è

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

A ogni iterazione del ciclo while, il valore corrente di n viene sostituito con $\lfloor n/2 \rfloor$. Dopo s iterazioni, il valore corrente è

$$\left\lfloor \frac{n}{2^s} \right\rfloor.$$

Il ciclo termina quando questo valore diventa 0, cioè quando

$$\left\lfloor \frac{n}{2^s} \right\rfloor = 0.$$

Questo accade esattamente quando $2^s > n$. Il più piccolo intero s tale che $2^s > n$ è

$$s = \lfloor \log_2 n \rfloor + 1.$$

Quindi il corpo del ciclo while viene eseguito esattamente $\lfloor \log_2 n \rfloor + 1$ volte (un'iterazione per ogni cifra binaria), e la sua condizione viene valutata $\lfloor \log_2 n \rfloor + 2$ volte.

- Il numero di volte in cui ciascuna riga di codice viene eseguita è:

riga	numero di esecuzioni
2, 3, 4	1 ciascuna
5	$\lfloor \log_2 n \rfloor + 2$
6, 7, 8, 9, 10	$\lfloor \log_2 n \rfloor + 1$ ciascuna
12	1

- Quindi il numero totale di passi è

$$1 + 1 + 1 + (\lfloor \log_2 n \rfloor + 2) + 5(\lfloor \log_2 n \rfloor + 1) + 1 = 6(\lfloor \log_2 n \rfloor + 1) + 5 = 6\lfloor \log_2 n \rfloor + 11.$$

Quindi,

$$S(n) = 6\lfloor \log_2 n \rfloor + 11 = \Theta(\log_2 n).$$

In parole, il numero di passi cresce in modo logaritmico nel valore numerico n .

- Infatti, per $n > 1$,

$$\log_2 n - 1 < \lfloor \log_2 n \rfloor \leq \log_2 n.$$

Dividendo per $\log_2 n > 0$, otteniamo

$$1 - \frac{1}{\log_2 n} < \frac{\lfloor \log_2 n \rfloor}{\log_2 n} \leq 1.$$

Poiché sia $1 - \frac{1}{\log_2 n}$ sia 1 convergono a 1, per il teorema del confronto (dei due carabinieri),

$$\lim_{n \rightarrow \infty} \frac{\lfloor \log_2 n \rfloor}{\log_2 n} = 1.$$

Quindi,

$$\lim_{n \rightarrow \infty} \frac{6\lfloor \log_2 n \rfloor + 11}{\log_2 n} = \lim_{n \rightarrow \infty} \left(6 \frac{\lfloor \log_2 n \rfloor}{\log_2 n} + \frac{11}{\log_2 n} \right) = 6 \cdot 1 + 0 = 6 \in \mathbb{R}_{>0}.$$

Esempio 12.3: ChangeToBase2 su $n = 13$

Abbiamo $\lfloor \log_2 13 \rfloor + 1 = 4$, quindi viene creato l'array $d[0..3]$ e il ciclo esegue 4 iterazioni:

iterazione	n	$q = \lfloor n/2 \rfloor$	$r = n \bmod 2$	assegnamento
1	13	6	1	$d[0] = 1$
2	6	3	0	$d[1] = 0$
3	3	1	1	$d[2] = 1$
4	1	0	1	$d[3] = 1$

Dopo la quarta iterazione $n = 0$: la condizione della riga 5 viene valutata per la quinta volta, è falsa, e l'algoritmo restituisce le cifre di $13 = (1101)_2$. Il numero di passi è

$$S(13) = 3 + 5 + 5 \cdot 4 + 1 = 29 = 6 \cdot 3 + 11.$$

12.4.2 Tempo di esecuzione in funzione della dimensione dell'input

- Esprimiamo ora il tempo di esecuzione in funzione della dimensione dell'input $|I|$. Questo è il delicato cambio di variabile.
- L'input non è il valore astratto n , ma una rappresentazione di n . Poiché n è rappresentato in base 10, il numero di cifre necessarie per scrivere n è

$$\lfloor \log_{10} n \rfloor + 1.$$

Ogni cifra decimale richiede un numero costante di bit. Quindi la dimensione dell'input soddisfa

$$|I| = \Theta(\lfloor \log_{10} n \rfloor + 1) = \Theta(\log_{10} n).$$

Poiché il cambio di base di un logaritmo lo modifica solo per un fattore costante ($\log_{10} n = \log_2 n / \log_2 10$), otteniamo

$$|I| = \Theta(\log_2 n),$$

lo stesso ordine di crescita ottenuto nel capitolo «Dimensione dell'input» per un numero scritto in binario. Questo significa che esistono costanti $c_1, c_2 > 0$ tali che, per ogni n sufficientemente grande,

$$c_1 \log_2 n \leq |I| \leq c_2 \log_2 n.$$

Equivalentemente,

$$\frac{1}{c_2} |I| \leq \log_2 n \leq \frac{1}{c_1} |I|.$$

- Poiché $S(n) = \Theta(\log_2 n)$, esistono costanti $a_1, a_2 > 0$ tali che, per ogni n sufficientemente grande,

$$a_1 \log_2 n \leq S(n) \leq a_2 \log_2 n.$$

Usando i limiti su $\log_2 n$ in funzione di $|I|$, otteniamo

$$\frac{a_1}{c_2} |I| \leq S(n) \leq \frac{a_2}{c_1} |I|,$$

cioè $S(n) = \Theta(|I|)$.

Il tempo di esecuzione dell'algoritmo **ChangeToBase2** è

$$T(|I|) = \Theta(|I|).$$

Dunque l'algoritmo è logaritmico nel valore numerico n , ma lineare nella dimensione dell'input $|I|$, perché la dimensione dell'input è essa stessa proporzionale a $\log_2 n$.

12.5 Test di primalità

- L'input dell'ultimo esempio è di nuovo un singolo numero n , ma ora il numero di passi cresce come $\sqrt{n}$, molto più rapidamente di $\log_2 n$. Vedremo che questo rende l'algoritmo esponenziale nella dimensione dell'input.

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

- Questo è lo pseudocodice dell'algoritmo **PrimalityTest** del capitolo «Algoritmi numerici».

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
4   end
5    $d := 2$ 
6   while  $d \leq \lfloor \sqrt{n} \rfloor$  do
7     if  $n \bmod d = 0$  then
8       return false
9     end
10     $d \leftarrow d + 1$ 
11  end
12  return true
13 end
```

12.5.1 Numero di passi

- Sia $S(n)$ il numero di passi eseguiti da **PrimalityTest** nel caso peggiore, dove n è il valore in input. Assumiamo $n \geq 2$, perché per $n < 2$ l'algoritmo si ferma dopo 2 passi.
- **Il caso peggiore si ha quando n è primo.** In questo caso l'algoritmo non esce mai in anticipo alla riga 8, quindi il ciclo while viene eseguito fino in fondo. La variabile d assume i valori

$$2, 3, \dots, \lfloor \sqrt{n} \rfloor,$$

quindi il corpo del ciclo while viene eseguito $\lfloor \sqrt{n} \rfloor - 1$ volte. La condizione del ciclo while viene valutata ancora una volta, quando $d = \lfloor \sqrt{n} \rfloor + 1$, quindi la riga 6 viene eseguita $\lfloor \sqrt{n} \rfloor$ volte.

- Il numero di volte in cui ciascuna riga di codice viene eseguita nel caso peggiore è:

riga	numero di esecuzioni (n primo)
2	1
3	0 (poiché $n \geq 2$)
5	1
6	$\lfloor \sqrt{n} \rfloor$
7	$\lfloor \sqrt{n} \rfloor - 1$
8	0 (poiché n è primo)
10	$\lfloor \sqrt{n} \rfloor - 1$
12	1

- Quindi il numero totale di passi nel caso peggiore è

$$1 + 1 + \lfloor \sqrt{n} \rfloor + (\lfloor \sqrt{n} \rfloor - 1) + (\lfloor \sqrt{n} \rfloor - 1) + 1 = 3\lfloor \sqrt{n} \rfloor + 1.$$

- Questo è effettivamente il caso peggiore. Se $n \geq 2$ non è primo, sia p il suo più piccolo divisore maggiore di 1: allora $p \leq \lfloor \sqrt{n} \rfloor$, e l'algoritmo restituisce **false** alla riga 8 quando $d = p$. La riga 6 viene eseguita $p - 1$ volte, la riga 7 $p - 1$ volte, la riga 10 $p - 2$ volte, e le righe 2, 5, 8 una volta ciascuna, per un totale di

$$3p - 1 \leq 3\lfloor \sqrt{n} \rfloor - 1 < 3\lfloor \sqrt{n} \rfloor + 1$$

passi.

- Quindi,

$$S(n) = 3\lfloor \sqrt{n} \rfloor + 1 = \Theta(\sqrt{n}).$$

Dunque $S(n)$ cresce come la radice quadrata di n . Infatti,

$$\sqrt{n} - 1 < \lfloor \sqrt{n} \rfloor \leq \sqrt{n}.$$

Dividendo per $\sqrt{n} > 0$, otteniamo

$$1 - \frac{1}{\sqrt{n}} < \frac{\lfloor \sqrt{n} \rfloor}{\sqrt{n}} \leq 1.$$

Poiché sia $1 - \frac{1}{\sqrt{n}}$ sia 1 convergono a 1 , per il teorema del confronto (dei due carabinieri),

$$\lim_{n \rightarrow \infty} \frac{\lfloor \sqrt{n} \rfloor}{\sqrt{n}} = 1.$$

Quindi,

$$\lim_{n \rightarrow \infty} \frac{3\lfloor \sqrt{n} \rfloor + 1}{\sqrt{n}} = \lim_{n \rightarrow \infty} \left(3 \frac{\lfloor \sqrt{n} \rfloor}{\sqrt{n}} + \frac{1}{\sqrt{n}} \right) = 3 \cdot 1 + 0 = 3 \in \mathbb{R}_{>0}.$$

Esempio 12.4: PrimalityTest su un numero primo e su un numero composto

- $n = 97$ è primo e $\lfloor \sqrt{97} \rfloor = 9$. L'algoritmo prova i divisori $d = 2, 3, \dots, 9$ (8 iterazioni), non trova alcun divisore, valuta una nona volta la condizione della riga 6 con $d = 10$, e restituisce **true** dopo

$$S(97) = 3 \cdot 9 + 1 = 28$$

passi.

- $n = 91 = 7 \cdot 13$ non è primo, e il suo più piccolo divisore maggiore di 1 è $p = 7$. L'algoritmo prova $d = 2, 3, \dots, 6$ senza successo, trova $91 \bmod 7 = 0$, e restituisce **false** dopo

$$3 \cdot 7 - 1 = 20$$

passi, anche se $\lfloor \sqrt{91} \rfloor = 9$ come per 97 .

12.5.2 Tempo di esecuzione in funzione della dimensione dell'input

- Determiniamo ora il tempo di esecuzione nel caso peggiore, esprimendo l'ordine di crescita del numero di passi in funzione della dimensione dell'input $|I|$.
- L'input è un singolo numero naturale n . Per rappresentare n in binario servono $\lfloor \log_2 n \rfloor + 1$ bit, quindi la dimensione dell'input soddisfa

$$|I| = \Theta(\log_2 n).$$

Questo significa che esistono costanti $c_1, c_2 > 0$ tali che, per ogni n sufficientemente grande,

$$c_1 \log_2 n \leq |I| \leq c_2 \log_2 n.$$

Equivalentemente,

$$\frac{1}{c_2} |I| \leq \log_2 n \leq \frac{1}{c_1} |I|.$$

La funzione $x \mapsto 2^x$ è strettamente crescente. Quindi, applicare 2^x a tutti i termini della disuguaglianza ne preserva l'ordine:

$$2^{\frac{1}{c_2} |I|} \leq 2^{\log_2 n} \leq 2^{\frac{1}{c_1} |I|}.$$

Poiché $2^{\log_2 n} = n$, otteniamo

$$2^{\frac{1}{c_2} |I|} \leq n \leq 2^{\frac{1}{c_1} |I|}.$$

Inoltre, $\sqrt{n} = n^{1/2}$. Estraeendo la radice quadrata di tutti i termini, il che di nuovo preserva l'ordine perché tutti i termini sono positivi, e usando $(2^x)^{1/2} = 2^{x/2}$, otteniamo

$$2^{\frac{1}{2c_2} |I|} \leq \sqrt{n} \leq 2^{\frac{1}{2c_1} |I|}.$$

- Poiché $S(n) = \Theta(\sqrt{n})$, esistono costanti $a_1, a_2 > 0$ tali che, per ogni n sufficientemente grande,

$$a_1\sqrt{n} \leq S(n) \leq a_2\sqrt{n}.$$

Usando i limiti su $\sqrt{n}$ in funzione di $|I|$, otteniamo

$$a_1 2^{\frac{1}{2c_2}|I|} \leq S(n) \leq a_2 2^{\frac{1}{2c_1}|I|}.$$

Dunque,

$$S(n) = 2^{\Theta(|I|)}.$$

- Il caso peggiore viene raggiunto sui numeri primi, e ci sono numeri primi di ogni dimensione: per ogni $k \geq 2$ esiste un numero primo con esattamente k cifre binarie, cioè compreso tra 2^{k-1} e 2^k (postulato di Bertrand). Quindi, per ogni dimensione dell'input, il massimo numero di passi è raggiunto da un primo n e soddisfa i limiti precedenti.

Il tempo di esecuzione nel caso peggiore dell'algoritmo **PrimalityTest**, in funzione della dimensione dell'input $|I|$, è

$$T(|I|) = 2^{\Theta(|I|)}.$$

Di conseguenza, l'ordine di crescita del tempo di esecuzione nel caso peggiore dell'algoritmo **PrimalityTest** è **esponenziale** nella dimensione dell'input.

12.6 Riepilogo

- La tabella seguente raccoglie i risultati del capitolo. Per i due algoritmi su array, il tempo di esecuzione è dato sotto l'ipotesi che tutti gli elementi usino un numero costante fissato di bit.

algoritmo	parametro n	passi nel caso peggiore $S(n)$	dimensione dell'input $ I $	tempo di esecuzione $T(I)$
FindMaximum	elementi dell'array	$4n = \Theta(n)$	$\Theta(n)$	$\Theta(I)$
InsertionSort	elementi dell'array	$\frac{3n^2+9n-8}{2} = \Theta(n^2)$	$\Theta(n)$	$\Theta(I ^2)$
ChangeToBase2	valore del numero	$6\lfloor \log_2 n \rfloor + 11 = \Theta(\log_2 n)$	$\Theta(\log_2 n)$	$\Theta(I)$
PrimalityTest	valore del numero	$3\lfloor \sqrt{n} \rfloor + 1 = \Theta(\sqrt{n})$	$\Theta(\log_2 n)$	$2^{\Theta(I)}$

Punti chiave.

- Il tempo di esecuzione si misura in **passi** (operazioni elementari, ciascuna eseguita in tempo costante); contiamo un passo per ogni esecuzione di una riga di pseudocodice.
- Consideriamo il **caso peggiore**: il massimo numero di passi su tutti gli input di una data dimensione. Quando il numero di esecuzioni di una riga dipende dall'input, una funzione ausiliaria t lo descrive, e cerchiamo l'input che massimizza il totale.
- L'analisi ha **due passi**: prima il numero di passi $S(n)$ in funzione di un parametro naturale n , poi il tempo di esecuzione $T(|I|)$ in funzione della dimensione dell'input.
- Se n è il numero di elementi di un array (con elementi di dimensione fissata), allora $|I| = \Theta(n)$ e l'ordine di crescita non cambia: $S(n) = \Theta(n^k)$ dà $T(|I|) = \Theta(|I|^k)$.

- Se n è il **valore** del numero in input, allora $|I| = \Theta(\log_2 n)$, cioè $n = 2^{\Theta(|I|)}$: un numero di passi $\Theta(\log_2 n)$ è lineare in $|I|$, mentre un numero di passi $\Theta(\sqrt{n})$ o $\Theta(n)$ è **esponenziale** in $|I|$.

Correttezza

13.1 Correttezza di un algoritmo

- I capitoli precedenti hanno misurato quanto costa un algoritmo: la dimensione del suo input e il suo tempo di esecuzione. Prima di chiedersi quanto sia veloce un algoritmo, però, dobbiamo essere sicuri che faccia ciò che promette. Questo capitolo presenta lo strumento standard per dimostrarlo, l'**invariante di ciclo**, e lo applica ad algoritmi già incontrati nei capitoli «Ordinamento», «Ricerca lineare e ricerca binaria» e «Algoritmi numerici»: Find Maximum, Insertion Sort, Binary Search e l'algoritmo di Euclide.
- La prima parte dell'analisi di un algoritmo consiste nel determinarne la **correttezza**, cioè nel dimostrare che calcola l'output corretto per ogni possibile istanza di input.
- Provare un algoritmo su alcune istanze può mostrare che è sbagliato (basta un controesempio), ma non può mai mostrare che è corretto: le istanze di input sono infinite. La correttezza va **dimostrata**.

Definizione 13.1: algoritmo corretto

Un algoritmo è **corretto** se, per ogni istanza di input I del problema, **termina** dopo un numero finito di passi e **restituisce l'output corretto** per I .

- Gli algoritmi che consideriamo sono costituiti da sequenze di istruzioni e da cicli. Le sequenze di istruzioni sono facili da analizzare, mentre un ciclo esegue le stesse istruzioni un numero di volte che dipende dall'input.

La correttezza di un algoritmo si basa sulla correttezza dei suoi cicli. La correttezza di un ciclo si dimostra per mezzo di un **invariante di ciclo**.

13.1.1 Invarianti di ciclo

Definizione 13.2: invariante di ciclo

Un **invariante di ciclo** è una proprietà delle variabili di un algoritmo che è vera ogni volta che viene valutata la condizione di un ciclo, cioè prima della prima iterazione del ciclo e dopo ogni iterazione del ciclo.

- Per dimostrare la correttezza di un algoritmo per mezzo di un invariante di ciclo dobbiamo mostrare tre cose:
 1. **Inizializzazione:** l'invariante di ciclo è vero prima della prima iterazione del ciclo.
 2. **Conservazione:** se l'invariante di ciclo è vero prima di un'iterazione del ciclo, allora rimane vero alla fine dell'iterazione, o equivalentemente, è vero prima dell'iterazione successiva.
 3. **Terminazione:** il ciclo termina dopo un numero finito di iterazioni. Quando il ciclo termina, l'invariante di ciclo, insieme al fatto che la condizione del ciclo è falsa, fornisce una proprietà utile che mostra che l'algoritmo è corretto.
- Notiamo la somiglianza con l'**induzione matematica**, dove per dimostrare che una proprietà vale è necessario dimostrare un caso base e un passo induttivo:
 - mostrare che l'invariante vale prima della prima iterazione corrisponde al **caso base**;
 - mostrare che l'invariante vale da un'iterazione alla successiva corrisponde al **passo induttivo**.

La proprietà di terminazione differisce dal consueto uso dell'induzione matematica, in cui il passo induttivo viene applicato infinite volte: qui l'"induzione" si ferma quando il ciclo termina.

- Per dimostrare che un ciclo termina, di solito si individua un numero naturale (per esempio $n - i + 1$ o la lunghezza di un sotto-array) che diminuisce strettamente a ogni iterazione e che deve essere positivo perché il ciclo continui. Poiché un numero naturale non può diminuire all'infinito, il ciclo termina.

Schema di una dimostrazione di correttezza con un invariante di ciclo

1. Enunciare l'invariante: una proprietà delle variabili, vera ogni volta che viene valutata la condizione del ciclo.
2. **Inizializzazione:** verificarlo con i valori assegnati prima del ciclo.
3. **Conservazione:** assumerlo insieme alla condizione del ciclo, eseguire il corpo una volta, e verificarlo con i nuovi valori.
4. **Terminazione:** trovare un numero naturale che diminuisce strettamente a ogni iterazione; poi combinare l'invariante con la negazione della condizione del ciclo per ottenere l'output corretto.

13.2 Correttezza di Find Maximum

Dato un array $a[1..n]$ di n numeri interi, con $n \geq 1$, consideriamo il problema di determinare il valore massimo dell'array.

- Il primo esempio è il ciclo più semplice: una singola scansione dell'array. Questo è lo pseudocodice dell'algoritmo **FindMaximum**.

Algorithm: FindMaximum

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

Output: Il valore massimo dell'array a

```

1 function FindMaximum( $a, n$ )
2    $max := a[1]$ 
3    $i := 2$ 
4   while  $i \leq n$  do
5     if  $a[i] > max$  then
6        $max \leftarrow a[i]$ 
7     end
8      $i \leftarrow i + 1$ 
9   end
10  return  $max$ 
11 end

```

- L'invariante di ciclo del ciclo while di FindMaximum è:

Ogni volta che viene valutata la condizione alla riga 4, la variabile max è uguale al valore massimo del sotto-array $a[1..i-1]$:

$$max = \max\{a[1], a[2], \dots, a[i-1]\}.$$

Esempio 13.1: Invariante di ciclo di FindMaximum

Consideriamo l'array $a = [9, 12, 3, 16, 19, 11, 12]$ con $n = 7$. La tabella mostra i valori di i e max ogni volta che viene valutata la condizione alla riga 4:

i	max	$a[1..i-1]$	$\max\{a[1], \dots, a[i-1]\}$
2	9	[9]	9
3	12	[9, 12]	12
4	12	[9, 12, 3]	12
5	16	[9, 12, 3, 16]	16
6	19	[9, 12, 3, 16, 19]	19
7	19	[9, 12, 3, 16, 19, 11]	19
8	19	[9, 12, 3, 16, 19, 11, 12]	19 (il ciclo termina)

A ogni valutazione l'invariante vale. Quando il ciclo termina $i = n + 1 = 8$, e l'algoritmo restituisce 19, il massimo dell'intero array.

- Usando l'invariante di ciclo possiamo dimostrare la correttezza dell'algoritmo **FindMaximum**, enunciata dalla seguente osservazione.

Osservazione 13.3

Dato un array $a[1..n]$ di $n \geq 1$ numeri interi, l'algoritmo **FindMaximum** termina e restituisce

$$\max\{a[1], a[2], \dots, a[n]\}.$$

Dimostrazione. Dimostriamo le tre proprietà dell'invariante di ciclo.

- **Inizializzazione.** Prima della prima iterazione, le righe 2 e 3 pongono $max = a[1]$ e $i = 2$. Il sotto-array $a[1..i - 1] = a[1..1]$ è costituito dal solo elemento $a[1]$, il cui massimo è $a[1] = max$. Quindi l'invariante vale.
- **Conservazione.** Supponiamo che, prima di un'iterazione, $max = \max\{a[1], \dots, a[i - 1]\}$. Il corpo del ciclo confronta $a[i]$ con max (riga 5):
 - se $a[i] > max$, allora la riga 6 pone $max \leftarrow a[i]$, e $a[i]$ è maggiore di tutti gli elementi di $a[1..i - 1]$;
 - se $a[i] \leq max$, allora max non cambia ed è maggiore o uguale a $a[i]$.

In entrambi i casi, alla fine del corpo

$$max = \max\{a[1], \dots, a[i - 1], a[i]\}.$$

La riga 8 incrementa i di 1, quindi, con il nuovo valore di i , si ha di nuovo $max = \max\{a[1], \dots, a[i - 1]\}$. Quindi l'invariante vale prima dell'iterazione successiva.

- **Terminazione.** La variabile i parte da $2 \leq n + 1$ e aumenta di 1 a ogni iterazione, quindi il ciclo termina dopo esattamente $n - 1$ iterazioni, quando per la prima volta $i = n + 1$. Sostituendo $i = n + 1$ nell'invariante di ciclo otteniamo

$$max = \max\{a[1], a[2], \dots, a[n]\},$$

che è il valore restituito alla riga 10. Quindi l'algoritmo è corretto. □

13.3 Correttezza di Insertion Sort

Dato un array $a[1..n]$ di n numeri naturali, con $n \geq 1$, consideriamo il problema di riordinare gli elementi dell'array in modo non decrescente:

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

- Con due cicli annidati la dimostrazione richiede due invarianti, uno per ciascun ciclo. Questo è lo pseudocodice dell'algoritmo **InsertionSort**.

Algorithm: InsertionSort

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

Output: L'array a ordinato in modo 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 posizione corretta
11     $i \leftarrow i + 1$ 
12  end
13 end
  
```

- La condizione alla riga 6 è valutata da sinistra a destra, e la seconda parte è valutata solo se la prima è vera: quando $j = 0$ la condizione è falsa e l'elemento inesistente $a[0]$ non viene mai acceduto. La condizione C++ `i > -1 and a[i] > key` si comporta allo stesso modo.
- L'algoritmo contiene due cicli annidati. Dimostriamo prima che cosa fa una singola esecuzione del ciclo while **interno** (righe 6–8), e poi usiamo questo risultato per dimostrare la correttezza del ciclo while **esterno** (righe 3–11).

13.3.1 Il ciclo interno

- Consideriamo un'iterazione del ciclo esterno, con un dato valore di $i \in \{2, 3, \dots, n\}$. Indichiamo con

$$b[1..i]$$

il contenuto del sotto-array $a[1..i]$ all'inizio di questa iterazione (cioè quando la condizione alla riga 3 è vera), e supponiamo che $b[1..i-1]$ sia ordinato in modo non decrescente. Dopo la riga 4 si ha $key = b[i]$.

- L'**invariante di ciclo** del ciclo while interno è:

Ogni volta che viene valutata la condizione alla riga 6:

- (I1) $a[1..j] = b[1..j]$ (i primi j elementi non sono stati spostati);
- (I2) $a[j+2..i] = b[j+1..i-1]$ (gli elementi $b[j+1], \dots, b[i-1]$ sono stati spostati di una posizione a destra);
- (I3) ogni elemento di $a[j+2..i]$ è strettamente maggiore di key .

Le componenti $a[i+1..n]$ non vengono mai modificate dal ciclo interno.

- La posizione $j+1$ è il “buco” in cui verrà inserito key : il suo contenuto attuale è una copia che verrà sovrascritta. Un sotto-array $a[l..r]$ con $l > r$ è **vuoto**, e qualunque proprietà di tutti i suoi elementi è banalmente vera.

Esempio 13.2: Invariante di ciclo del ciclo interno

Consideriamo l'array $a = [5, 2, 4, 1, 3]$ all'inizio dell'iterazione $i = 5$ del ciclo esterno: $b[1..5] = [1, 2, 4, 5, 3]$ e $key = 3$. La tabella mostra l'array ogni volta che viene valutata la condizione alla riga 6.

j	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[1..j]$	$a[j+2..5]$	condizione
4	1	2	4	5	3	[1, 2, 4, 5]	[]	$a[4] = 5 > 3$: vera
3	1	2	4	5	5	[1, 2, 4]	[5]	$a[3] = 4 > 3$: vera
2	1	2	4	4	5	[1, 2]	[4, 5]	$a[2] = 2 \leq 3$: falsa

A ogni valutazione $a[1..j] = b[1..j]$, $a[j+2..5] = b[j+1..4]$ e tutti gli elementi di $a[j+2..5]$ sono maggiori di 3. Quando il ciclo termina $j = 2$, e la riga 10 pone $a[3] \leftarrow 3$, ottenendo $a[1..5] = [1, 2, 3, 4, 5]$.

Lemma 13.4: ciclo interno di Insertion Sort

Consideriamo un'iterazione del ciclo esterno di **InsertionSort** con $i \in \{2, \dots, n\}$, e sia $b[1..i]$ il contenuto di $a[1..i]$ all'inizio dell'iterazione. Se $b[1..i-1]$ è ordinato in modo non decrescente, allora il ciclo while interno termina e, dopo la riga 10, il sotto-array $a[1..i]$ contiene gli elementi $b[1], \dots, b[i]$ ordinati in modo non decrescente, mentre $a[i+1..n]$ è invariato.

Dimostrazione. Dimostriamo le tre proprietà dell'invariante di ciclo (I1)–(I3).

- **Inizializzazione.** Prima della prima valutazione della condizione, $j = i - 1$ e nulla è stato modificato. Quindi $a[1..i - 1] = b[1..i - 1]$, cioè (I1) vale. I sotto-array $a[j + 2..i] = a[i + 1..i]$ e $b[j + 1..i - 1] = b[i..i - 1]$ sono entrambi vuoti, quindi (I2) e (I3) valgono banalmente.
- **Conservazione.** Supponiamo che (I1)–(I3) valgano e che la condizione alla riga 6 sia vera, cioè $j \geq 1$ e $a[j] > key$. Per (I1), $a[j] = b[j]$, quindi $b[j] > key$. La riga 7 pone $a[j + 1] \leftarrow a[j] = b[j]$; quindi, per (I2),

$$a[j + 1..i] = b[j..i - 1],$$

e tutti questi elementi sono maggiori di key (per (I3) e $b[j] > key$). Le posizioni $1, \dots, j$ non vengono modificate. La riga 8 pone $j \leftarrow j - 1$. Indicando con $j' = j - 1$ il nuovo valore, si ha $a[1..j'] = b[1..j']$, $a[j' + 2..i] = a[j + 1..i] = b[j..i - 1] = b[j' + 1..i - 1]$, e tutti gli elementi di $a[j' + 2..i]$ sono maggiori di key . Quindi (I1)–(I3) valgono prima della valutazione successiva della condizione.

- **Terminazione.** La variabile j parte da $i - 1$ e diminuisce di 1 a ogni iterazione; poiché il ciclo si ferma al più tardi quando $j = 0$, il ciclo interno esegue al più $i - 1$ iterazioni e termina. Quando termina, si ha $j = 0$ oppure $a[j] = b[j] \leq key$. Per (I1)–(I3):

- $a[1..j] = b[1..j]$ è ordinato e, se $j \geq 1$, il suo elemento più grande $b[j]$ è $\leq key$;
- $a[j + 2..i] = b[j + 1..i - 1]$ è ordinato e tutti i suoi elementi sono $> key$.

La riga 10 pone $a[j + 1] \leftarrow key = b[i]$. Quindi

$$a[1..i] = [b[1], \dots, b[j], key, b[j + 1], \dots, b[i - 1]],$$

che contiene esattamente gli elementi $b[1], \dots, b[i]$ ed è ordinato in modo non decrescente, perché $b[1..j]$ è ordinato, $b[j] \leq key < b[j + 1]$, e $b[j + 1..i - 1]$ è ordinato. Le posizioni $i + 1, \dots, n$ non vengono mai accedute.

□

13.3.2 Il ciclo esterno

- L'invariante di ciclo del ciclo while esterno è:

Ogni volta che viene valutata la condizione alla riga 3, il sotto-array $a[1..i-1]$ è costituito dagli elementi originariamente in $a[1..i-1]$, ma ordinati in modo non decrescente, e il sotto-array $a[i..n]$ è uguale al sotto-array originale $a[i..n]$.

Esempio 13.3: Invariante di ciclo del ciclo esterno

Consideriamo l'array $a = [5, 2, 4, 1, 3]$ con $n = 5$. La tabella mostra l'array ogni volta che viene valutata la condizione alla riga 3; il sotto-array ordinato $a[1..i-1]$ è evidenziato in blu.

i	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[1..i-1]$ originale
2	5	2	4	1	3	[5]
3	2	5	4	1	3	[5, 2]
4	2	4	5	1	3	[5, 2, 4]
5	1	2	4	5	3	[5, 2, 4, 1]
6	1	2	3	4	5	[5, 2, 4, 1, 3] (il ciclo termina)

- Usando l'invariante di ciclo e il Lemma 13.4 possiamo dimostrare la correttezza dell'algoritmo InsertionSort.

Osservazione 13.5

Dato un array $a[1..n]$ di $n \geq 1$ numeri naturali, l'algoritmo InsertionSort termina e, alla fine, l'array $a[1..n]$ contiene gli elementi originali riordinati in modo non decrescente:

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

Dimostrazione. Dimostriamo le tre proprietà dell'invariante di ciclo del ciclo esterno.

- **Inizializzazione.** Prima della prima iterazione, $i = 2$ (riga 2) e l'array non è stato modificato. Il sotto-array $a[1..i-1] = a[1..1]$ è costituito dal singolo elemento originale $a[1]$, ed è banalmente ordinato. Il sotto-array $a[2..n]$ è quello originale. Quindi l'invariante vale.
- **Conservazione.** Supponiamo che l'invariante valga e che $i \leq n$. Allora $a[1..i-1]$ contiene gli elementi originali di $a[1..i-1]$ ordinati, e $a[i]$ è l'elemento originale $a[i]$. Per il Lemma 13.4, dopo la riga 10 il sotto-array $a[1..i]$ contiene questi i elementi, cioè gli elementi originali di $a[1..i]$, ordinati in modo non decrescente, e $a[i+1..n]$ è invariato, quindi ancora uguale a quello originale. La riga 11 pone $i \leftarrow i+1$, quindi l'invariante vale con il nuovo valore di i .
- **Terminazione.** La variabile i parte da $2 \leq n+1$ e aumenta di 1 a ogni iterazione, e ogni ciclo interno termina (Lemma 13.4). Quindi il ciclo esterno termina dopo

esattamente $n - 1$ iterazioni, quando per la prima volta $i = n + 1$. Sostituendo $i = n + 1$ nell'invariante di ciclo, il sotto-array $a[1..n]$, cioè l'intero array, è costituito dagli elementi originali ordinati in modo non decrescente. Quindi l'algoritmo è corretto. □

13.4 Correttezza di Binary Search

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.

- Nei due algoritmi precedenti il ciclo viene sempre eseguito finché la sua condizione diventa falsa. La ricerca binaria può invece fermarsi in anticipo, con un **return** dentro il ciclo, e la dimostrazione deve tenere conto di entrambe le uscite. Questo è lo pseudocodice dell'algoritmo **BinarySearch** (iterativo).

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

```

- L'invariante di ciclo del ciclo while di BinarySearch è:

Ogni volta che viene valutata la condizione alla riga 4:

(I1) $1 \leq left$ e $right \leq n$;

(I2) se m è in a , allora m è in $a[left..right]$, cioè

$$m \in \{a[1], \dots, a[n]\} \implies m \in \{a[left], \dots, a[right]\}.$$

- A parole: l'algoritmo non scarta mai una porzione dell'array che può contenere m . Il sotto-array $a[left..right]$ è detto **intervallo di ricerca** e contiene $right - left + 1$ elementi (è vuoto se $left > right$).

Esempio 13.4: Invariante di ciclo di BinarySearch

Ricerca di $m = 10$ in $a = [3, 9, 11, 12, 12, 16, 19]$ ($n = 7$). La tabella mostra l'intervallo di ricerca ogni volta che viene valutata la condizione alla riga 4.

$left$	$right$	$a[left..right]$	azione
1	7	[3, 9, 11, 12, 12, 16, 19]	$mid = 4, a[4] = 12 > 10 \Rightarrow right = 3$
1	3	[3, 9, 11]	$mid = 2, a[2] = 9 < 10 \Rightarrow left = 3$
3	3	[11]	$mid = 3, a[3] = 11 > 10 \Rightarrow right = 2$
3	2	[]	$left > right \Rightarrow$ restituisce -1

Poiché 10 non è in a , l'invariante (I2) è banalmente vero; quando l'intervallo diventa vuoto l'algoritmo restituisce correttamente -1 .

Osservazione 13.6

Dati un array $a[1..n]$ di n numeri naturali ordinato in modo non decrescente e un numero naturale m , l'algoritmo **BinarySearch** termina; restituisce un indice k tale che $a[k] = m$ se m è in a , e restituisce -1 altrimenti.

Dimostrazione. Dimostriamo le tre proprietà dell'invariante di ciclo.

- **Inizializzazione.** Le righe 2 e 3 pongono $left = 1$ e $right = n$. Allora (I1) vale, e $a[left..right] = a[1..n]$ è l'intero array, quindi (I2) vale.
- **Conservazione.** Supponiamo che (I1)–(I2) valgano e che $left \leq right$. La riga 5 calcola $mid = left + \lfloor (right - left) / 2 \rfloor$; poiché $0 \leq \lfloor (right - left) / 2 \rfloor \leq right - left$, si ha

$$1 \leq left \leq mid \leq right \leq n,$$

quindi $a[mid]$ è una componente valida dell'array. Sono possibili tre casi.

- $a[mid] = m$: l'algoritmo restituisce mid alla riga 7, che è un output corretto.

- $a[mid] < m$: poiché l'array è ordinato, per ogni $k \leq mid$ si ha $a[k] \leq a[mid] < m$. Quindi m non è in $a[left..mid]$, e se m è in $a[left..right]$ allora è in $a[mid + 1..right]$. La riga 10 pone $left \leftarrow mid + 1 \geq 1$, quindi (I1) e (I2) valgono ancora.
- $a[mid] > m$: poiché l'array è ordinato, per ogni $k \geq mid$ si ha $a[k] \geq a[mid] > m$. Quindi, se m è in $a[left..right]$, allora è in $a[left..mid - 1]$. La riga 13 pone $right \leftarrow mid - 1 \leq n$, quindi (I1) e (I2) valgono ancora.
- **Terminazione.** Consideriamo il numero di elementi dell'intervallo di ricerca, $right - left + 1$. Nel secondo caso il nuovo numero è $right - mid \leq right - left$, nel terzo caso è $mid - left \leq right - left$: in entrambi i casi diminuisce di almeno 1. Poiché il ciclo continua solo finché $right - left + 1 \geq 1$, il ciclo termina dopo al più n iterazioni. L'algoritmo finisce in uno di due modi:
 - restituisce mid con $a[mid] = m$, che è corretto;
 - la condizione $left \leq right$ è falsa, cioè l'intervallo di ricerca $a[left..right]$ è vuoto. Poiché un sotto-array vuoto non contiene m , per (I2) m non è in a , e l'algoritmo restituisce correttamente -1 alla riga 16.

Quindi l'algoritmo è corretto.

□

13.5 Correttezza dell'algoritmo di Euclide

Dati due numeri naturali $n, m \geq 1$, consideriamo il problema di calcolare il loro massimo comune divisore $\gcd(n, m)$.

- Nell'ultimo esempio l'invariante non è una proprietà di una porzione di un array, ma un'uguaglianza fra due numeri che non cambia mai. Questo è lo 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
```

- La dimostrazione si basa sulla seguente proprietà del massimo comune divisore, dimostrata nel capitolo «Algoritmi numerici»:

$$\gcd(n, m) = \gcd(m, n \bmod m) \quad \text{per ogni } n \geq 1, m \geq 1.$$

- Usiamo anche la convenzione $\gcd(n, 0) = n$ per $n \geq 1$: ogni numero naturale divide 0, quindi i divisori comuni di n e 0 sono i divisori di n , e il più grande di essi è n .
- Indichiamo con n_0 e m_0 i valori di n e m dati in input. L'**invariante di ciclo** del ciclo while è:

Ogni volta che viene valutata la condizione alla riga 2, $n \geq 1, m \geq 0$, e

$$\gcd(n, m) = \gcd(n_0, m_0).$$

Esempio 13.5: Invariante di ciclo dell'algoritmo di Euclide

Consideriamo $n_0 = 252$ e $m_0 = 105$. La tabella mostra i valori di n e m ogni volta che viene valutata la condizione alla riga 2:

n	m	$\gcd(n, m)$
252	105	21
105	42	21
42	21	21
21	0	21 (il ciclo termina)

Il valore $\gcd(n, m) = 21$ non cambia mai, e quando $m = 0$ l'algoritmo restituisce $n = 21 = \gcd(252, 105)$.

Osservazione 13.7

Dati due numeri naturali $n_0, m_0 \geq 1$, l'algoritmo `EuclideanAlgorithm` termina e restituisce $\gcd(n_0, m_0)$.

Dimostrazione. Dimostriamo le tre proprietà dell'invariante di ciclo.

- **Inizializzazione.** Prima della prima iterazione $n = n_0 \geq 1$ e $m = m_0 \geq 1$, quindi banalmente $\gcd(n, m) = \gcd(n_0, m_0)$.
- **Conservazione.** Supponiamo che l'invariante valga e che $m \neq 0$, cioè $m \geq 1$. La riga 3 calcola $r = n \bmod m$, con $0 \leq r < m$. Le righe 4–5 sostituiscono la coppia (n, m) con (m, r) . Il nuovo valore di n è $m \geq 1$, il nuovo valore di m è $r \geq 0$, e, poiché $n, m \geq 1$,

$$\gcd(m, r) = \gcd(m, n \bmod m) = \gcd(n, m) = \gcd(n_0, m_0).$$

Quindi l'invariante vale prima dell'iterazione successiva.

- **Terminazione.** A ogni iterazione il valore di m viene sostituito da $r < m$, quindi m è un numero naturale che diminuisce strettamente. Quindi il ciclo termina dopo al più m_0 iterazioni, quando $m = 0$. Sostituendo $m = 0$ nell'invariante di ciclo otteniamo

$$\gcd(n_0, m_0) = \gcd(n, 0) = n,$$

che è il valore restituito alla riga 7. Quindi l'algoritmo è corretto.

□

- La dimostrazione non usa l'ipotesi $n_0 \geq m_0$: se $n_0 < m_0$, allora $n_0 \bmod m_0 = n_0$ e la prima iterazione si limita a scambiare i due numeri.
- Il limite “al più m_0 iterazioni” basta a dimostrare la terminazione, ma è molto pessimistico: il numero di iterazioni è in realtà $O(\log_2 m_0)$ (nell'esempio, 3 iterazioni per $m_0 = 105$).

13.6 Correttezza degli algoritmi ricorsivi

- Gli invarianti di ciclo sono lo strumento per dimostrare la correttezza degli algoritmi **iterativi**, cioè basati su cicli.
- Un algoritmo **ricorsivo** risolve un'istanza di un problema chiamando se stesso su una o più istanze più piccole. La sua correttezza si dimostra direttamente per **induzione matematica** sulla dimensione dell'istanza:
 - **caso base:** l'algoritmo restituisce l'output corretto sulle istanze più piccole, che sono risolte senza chiamate ricorsive;
 - **passo induttivo:** sotto l'ipotesi induttiva che le chiamate ricorsive restituiscano l'output corretto sulle istanze più piccole, l'algoritmo ne combina i risultati nell'output corretto per l'istanza corrente.
- La terminazione segue dal fatto che ogni chiamata ricorsiva è fatta su un'istanza strettamente più piccola, quindi prima o poi si raggiunge il caso base.

La correttezza degli algoritmi ricorsivi si dimostra per induzione sulla dimensione dell'input. Questa tecnica viene applicata insieme allo studio degli algoritmi ricorsivi.

Punti chiave del capitolo

- Un algoritmo è corretto se, per ogni istanza di input, termina e restituisce l'output corretto; i test possono solo mostrare che un algoritmo è sbagliato.
- La correttezza di un ciclo si dimostra con un invariante di ciclo: inizializzazione, conservazione e terminazione.
- L'invariante, insieme alla negazione della condizione del ciclo, fornisce l'output: $i = n + 1$ per Find Maximum e Insertion Sort, un intervallo di ricerca vuoto per Binary Search, $m = 0$ per l'algoritmo di Euclide.

- I cicli annidati richiedono un invariante per ciascun ciclo (Insertion Sort); un **return** dentro il ciclo è una seconda uscita che va dimostrata corretta (Binary Search).
- Gli algoritmi ricorsivi si dimostrano corretti per induzione sulla dimensione dell'input.