

Algoritmi e Principi dell'Informatica

Linguaggi

Definizioni

alfabeto	insieme finito di simboli
stringa	sequenza ordinata e finita di elementi dell'alfabeto (ϵ denota la stringa vuota)
lunghezza concatenazione	numeri di elementi della stringa $ abc = 3, \epsilon = 0$
riflesso	unione di due o più stringhe in una singola stringa es. se $w_1 = abc$ e $w_2 = def$ allora $w_1.w_2 = abcdef$
Σ^*	inversione degli ordini dei caratteri di una stringa $w = c_1 \dots c_n \iff w^R = c_n \dots c_1$ con $ c_i = 1$
Σ^+	es. se $w = abc$ allora $w^R = cba$
Σ^*	insieme di tutte le stringhe su Σ
Σ^+	es. $\Sigma = \{0, 1\}, \Sigma^+ = \{\epsilon, 0, 1, 00, 01, \dots\}$
insieme di tutte le stringhe non vuote su Σ	es. $\Sigma = \{0, 1\}, \Sigma^+ = \{0, 1, 00, 01, \dots\}$

Operazioni su linguaggi

unione	$L_1 \cup L_2 = \{w \mid w \in L_1 \vee w \in L_2\}$
intersezione	$L_1 \cap L_2 = \{w \mid w \in L_1 \wedge w \in L_2\}$
differenza	$L_1 \setminus L_2 = \{w \mid w \in L_1 \wedge w \notin L_2\}$
complemento	$L^c = A^* \setminus L = \{w \mid w \in A^* \wedge w \notin L\}$
concatenazione	$L_1.L_2 = \{w_1.w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}$
riflesso	$L^R = \{w^R \mid w \in L\}$

Modelli di calcolo

Automi a stati finiti (ASF/FSA)

Formalmente definiti come una quintupla (Q, I, δ, q_0, F) dove Q è un insieme finito di stati, I è l'alfabeto di input, $\delta : Q \times I \rightarrow Q$ (o, $\delta : Q \times I \rightarrow \mathcal{P}(Q)$ se l'automa è non deterministico) è la funzione di transizione che mappa una coppia (stato corrente, input) a uno stato (o stati, se l'automa è non deterministico) di destinazione, q_0 è lo stato di partenza e $F \subseteq Q$ è l'insieme degli stati finali.

FSA riconoscitore

Un linguaggio L su I è riconosciuto dall'FSA se e solo se data una stringa $w \in L$ si ha $\delta^*(q_0, w) \in F$ dove δ^* è un'estensione di δ definita ricorsivamente con caso base $\delta^*(q, \epsilon) := q$ e passo ricorsivo $\delta^*(q, y.i) := \delta(\delta^*(q, y), i)$ con $i \in I, y \in I^*$.

FSA traduttore

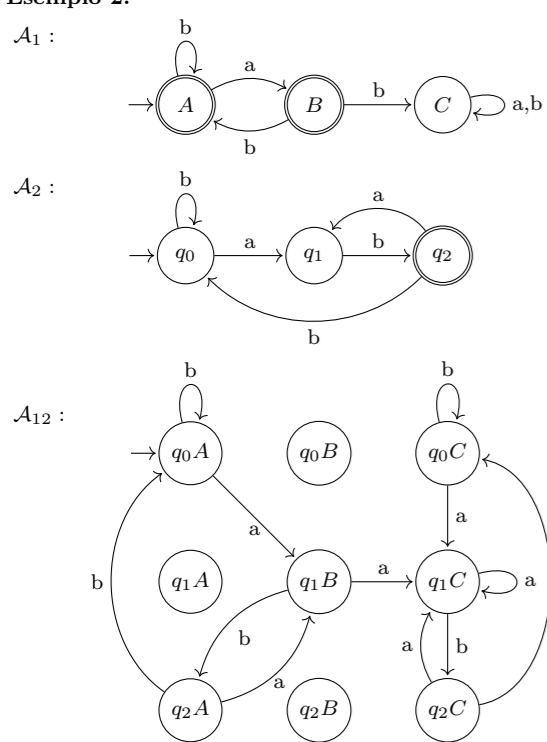
Un FSA traduttore associa un simbolo letto a uno scritto a ogni transizione tramite una funzione di traduzione η .

Operazioni su FSA

1. Dato un'automa \mathcal{A} che riconosce il linguaggio L , possiamo costruire l'automa \mathcal{A}' che riconosce il linguaggio L^C completando l'automa \mathcal{A} e invertendo stati finali con iniziali.

2. Dati due automi $\mathcal{A}_1, \mathcal{A}_2$ che riconoscono i linguaggi L_1, L_2 rispettivamente, si può costruire algoritmamente un'automa che riconosce $L_1 \cap L_2, L_1 \cup L_2, L_1 \setminus L_2$ partendo dall'automa prodotto \mathcal{A}_{12} .

Esempio 2.



Affinché l'automa \mathcal{A}_{12} riconosca $L_1 \cap L_2$, dobbiamo impostare come stati finali gli stati che sono finali in \mathcal{A}_1 e \mathcal{A}_2 quindi q_2A e q_2B . Affinché riconosca

$L_1 \cup L_2$, dobbiamo impostare come stati finali gli stati che sono finali in \mathcal{A}_1 o \mathcal{A}_2 quindi $q_0A, q_1A, q_2A, q_0B, q_1B, q_2B, q_2C$ e affinché riconosca $L_1 \setminus L_2$ dobbiamo impostare come stati finali gli stati che sono finali in \mathcal{A}_1 ma non \mathcal{A}_2 , quindi q_0A, q_1A, q_0B, q_1B .

Pumping lemma per FSA

Sia L un linguaggio riconosciuto da un FSA, allora $\exists p \in \mathbb{N}^+$ tale che ogni stringa $w \in L$ con $|w| \geq p$ può essere scritta come $w = xyz$ con:

1. $|xy| \leq p$
2. $y \neq \epsilon$
3. $\forall i \geq 0, xy^i z \in L$

Esempio Dimostriamo che il linguaggio $L = \{0^n 1^n \mid n \geq 0\}$ non è riconosciuto da un FSA usando il pumping lemma.

1. Supponiamo per assurdo che L sia riconosciuto da un FSA e scegliamo $w = 0^p 1^p$

2. Scomponiamo w in xyz con:

- $|xy| \leq p$
- $|y| \geq 1$

Scegliamo $x = 0^\alpha, y = 0^\beta, z = 0^{p-\alpha-\beta} 1^p$ con $\alpha \geq 0, \beta \geq 1, \alpha + \beta \leq p$

3. Troviamo $i \geq 0$ tale che $xy^i z \notin L$

$xy^i z = 0^\alpha 0^{\beta i} 0^{p-\alpha-\beta} 1^p = 0^{p+i\beta-\beta} 1^p$. Da cui abbiamo

$xy^i z \in L \iff p + i\beta - \beta = p \iff i = 1$. Scegliendo quindi $i \neq 1$, ad esempio $i = 0$, abbiamo che $xy^i z \notin L$ che contraddice la nostra supposizione iniziale.

L'esempio utilizza la convenzione della MT infinita a destra con il simbolo Z_0 che denota la prima cella del nastro di memoria.

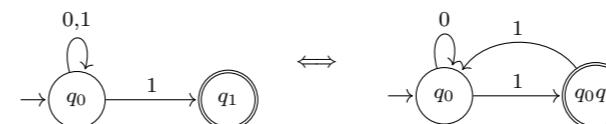
MT traduttore

Una MT trad. è dotata di un addizionale nastro di output su cui può solo scrivere e in cui la testina può muoversi solo a destra (R) o stare ferma (S).

Non determinismo

Informalmente, un modello di calcolo si dice non deterministico se esiste almeno una configurazione da cui è possibile prendere più di una strada. Gli FSA e le MT non deterministici hanno le stesse capacità esppressive della loro versione deterministica. Gli automi a pilha non deterministici (NPDA), invece, sono più espressivi di quelli deterministici. Per esempio, possono riconoscere $L = \{a^n b^n\} \cup \{a^n b^{2n}\}$. Dato un FSA non deterministico con insieme di stati Q , è possibile ricavare la sua versione non deterministica con un numero di stati che è al più $2^{|Q|}$ usando la costruzione per sottoinsiemi (powerset construction).

Esempio



Macchine RAM

Le macchine RAM sono un modello di calcolo equivalente alle macchine di Turing. Hanno un nastro di lettura In e uno di scrittura Out e sono dotate di memoria infinita con accesso a indirizzamento diretto. Supportano le seguenti istruzioni:

READ X	$M[X] \leftarrow \text{In}$	WRITE* X	$X \text{ Out} \leftarrow M[M[X]]$
READ* X	$M[M[X]] \leftarrow \text{In}$	JUMP 1	$PC \leftarrow l$
LOAD X	$M[0] \leftarrow M[X]$	JZ 1	$se M[0]=0$
LOAD* X	$M[0] \leftarrow X$	PC $\leftarrow l$	
LOAD* X	$M[0] \leftarrow M[M[X]]$	JGT 1	$se M[0]>0$
STORE X	$M[X] \leftarrow M[0]$	PC $\leftarrow l$	
STORE* X	$M[M[X]] \leftarrow M[0]$	JGZ 1	$se M[0]\geq 0$
ADD X	$M[0] \leftarrow M[0] + M[X]$	PC $\leftarrow l$	
SUB X	$M[0] \leftarrow M[0] - M[X]$	JLT 1	$se M[0]<0$
MUL X	$M[0] \leftarrow M[0] \times M[X]$	PC $\leftarrow l$	
DIV X	$M[0] \leftarrow M[0] / M[X]$	JLZ 1	$se M[0]\leq 0$
WRITE X	$Out \leftarrow M[X]$	PC $\leftarrow l$	
WRITE= X	$Out \leftarrow X$	HALT	termina exec.

Esistono le varianti con = e * anche per le istruzioni ADD, SUB, MUL e DIV.

Logica monadica del I ordine (MFO)

La logica monadica del I ordine è un modello di calcolo in grado di riconoscere linguaggi star-free, ossia linguaggi regolari esprimibili senza l'utilizzo della star di Kleene. La sintassi di una formula φ della logica monadica del I ordine è $\varphi := a(x) \mid x < y \mid \neg \varphi \mid \varphi \wedge \varphi \mid \forall x(\varphi)$. Dove $x, y \in \mathbb{N}$, $a(x)$ è un predicato unario che è vero se e solo se alla posizione x c'è il simbolo a e $<$ è un predicato binario che corrisponde alla relazione di minore. Partendo da questi assiomi si può definire il connettivo \vee , il quantificatore esistenziale \exists , le relazioni aritmetiche $=, \leq, >, \geq$ e somme e sottrazioni tra variabili e costanti ($y = x \pm k$). Si possono inoltre definire abbreviazioni ausiliarie, come:

$$\begin{aligned} \text{last}(x) &\quad \neg \exists y (y > x) \\ y = S(x) &\quad y = x + 1 \end{aligned}$$

Logica monadica del II ordine (MSO)

La logica monadica del II ordine è un modello di calcolo con le stesse capacità espensive degli automi a stati finiti. Differisce da MFO per il fatto che consente la quantificazione su insiemi di posizioni.

Esempio La seguente formula della logica monadica del II ordine riconosce il linguaggio $L = \{a^{2n} \mid n \in \mathbb{N}^+\}$.

$$\exists P (\forall x(a(x) \wedge \neg P(0)) \wedge \forall y(y = x + 1 \implies (\neg P(x) \iff P(y))) \wedge (\text{last}(x) \implies P(x)))$$

Dove $X(x)$ significa $x \in X$.

Grammatiche

Una grammatica è una quadrupla $G = (T, N, P, S)$ dove T è l'alfabeto terminale, N è l'alfabeto non terminale, $S \in N$ è il simbolo iniziale e $P \subseteq \mathbb{N}^+ \times (N \cup T)^*$ è l'insieme delle produzioni sintattiche. Data una grammatica G si definisce derivazione immediata la relazione binaria \Rightarrow_G definita come:

$$x \Rightarrow_G y \iff \exists u, v, p, q \in (N \cup T)^* : (x = upv) \wedge (p \rightarrow q \in P) \wedge (y = uqv)$$

Esempio Prendiamo in considerazione la grammatica $G = \langle \{a, b\}, \{S, R\}, \{S \rightarrow aR, R \rightarrow bS, S \rightarrow \epsilon\}, S \rangle$, abbiamo che $abaR \Rightarrow_G ababS$. Il linguaggio $L(G)$ generato dalla grammatica G è l'insieme di tutte e sole le stringhe s di soli caratteri di T tali che $S \xrightarrow{*} s$ dove $\xrightarrow{*}$ è la chiusura riflessiva e transitiva di \Rightarrow_G . A seconda delle limitazioni imposte sulle regole di produzione $\alpha \rightarrow \beta$ si hanno 4 tipi di grammatiche

Tipo	Nome	Regole prod.	Equiv.
0	Non ristretta	nessuna	TM
1	Monotone oppure Context sensitive	$ \alpha \leq \beta $ oppure $\alpha = \gamma A \delta, \beta = \gamma \chi \delta$ con $\chi \neq \epsilon$	(LBA)
2	Context free	$ \alpha = 1$	NPDA
3	Regolare	$A \rightarrow \alpha, A \rightarrow \alpha A$ o $A \rightarrow \alpha, A \rightarrow A \alpha$	FSA

Si può dimostrare che le grammatiche monotone e dipendenti dal contesto hanno la stessa capacità espressiva.

Conversione Automa ↔ Grammatica

Dato un automa \mathcal{A} che riconosce un certo

Teorema di Rice

Sia $F = \{f_i \mid i \in \mathbb{N}\}$ l'insieme di tutte le funzioni computabili, sia $P \subseteq F = \{f_i \mid f_i \text{ soddisfa una certa proprietà}\}$ un sottoinsieme di F di funzioni computabili che soddisfano una certa proprietà e sia $S = \{x \mid f_x \in P\}$ l'insieme degli indici delle funzioni che appartengono a P . Allora S ric. $\iff P = \emptyset \vee P = F$. Più informalmente, il teorema di Rice afferma che per qualsiasi proprietà non banale (cioè vera per almeno una funzione e falsa per almeno una funzione) delle funzioni computabili, non esiste un algoritmo che possa decidere se una data funzione computabile soddisfi o meno quella proprietà.

Riduzione

La riduzione può essere usata per dimostrare che un dato problema è decidibile o indecidibile.

Caso decidibile Sia P_1 un noto problema decidibile e sia P_2 un problema che si sospetta essere decidibile. Si può dimostrare che P_2 è decidibile riducendolo a P_1 ($P_2 \leq P_1$).

```
RISOLVIP2(x)
y ← RIDUCI(x)
sol ← RISOLVIP1(y)
return sol
```

Caso indecidibile Sia P_1 un noto problema indecidibile e sia P_2 un problema che si sospetta essere indecidibile. Si può dimostrare che P_2 è indecidibile riducendo P_1 a P_2 ($P_1 \leq P_2$).

```
RISOLVIP1(x)
y ← RIDUCI(x)
sol ← RISOLVIP2(y)
return sol
```

In altre parole, se riuscissimo a risolvere P_2 , riusciremmo a risolvere anche P_1 , ma questo è impossibile in quanto P_1 è un noto problema indecidibile.

Noti problemi indecidibili

Problema dell'arresto (Halting problem)

Data una macchina di Turing \mathcal{M} e un input w , determinare se \mathcal{M} si arresta su w .

$$H = \{\langle \mathcal{M}, w \rangle \mid \mathcal{M} \text{ si arresta sull'input } w\}$$

Problema dell'equivalenza delle MT

Date due macchine di Turing \mathcal{M}_1 e \mathcal{M}_2 , determinare se calcolano la stessa funzione.

$$EQ = \{\langle \mathcal{M}_1, \mathcal{M}_2 \rangle \mid \mathcal{M}_1 \text{ e } \mathcal{M}_2 \text{ calcolano la stessa funzione}\}$$

Problema della totalità delle MT

Data una macchina di Turing \mathcal{M} , determinare se \mathcal{M} si arresta su tutti gli input.

$$TOT = \{\langle \mathcal{M} \rangle \mid \mathcal{M} \text{ si arresta su tutti gli input}\}$$

Problema del linguaggio vuoto

Data una macchina di Turing \mathcal{M} , determinare se il linguaggio riconosciuto da \mathcal{M} è vuoto.

$$E = \{\langle \mathcal{M} \rangle \mid L(\mathcal{M}) = \emptyset\}$$

Analisi di complessità degli algoritmi

Formule note

$$\sum_{i=1}^n i = \frac{n(n+1)}{2} \quad \sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6} \quad \sum_{i=1}^n i^3 = \frac{n^2(n+1)^2}{4}$$

$$\sum_{i=1}^n x^i = \frac{x^{n+1}-1}{x-1} \quad \sum_{i=1}^{\infty} x^i = \frac{1}{1-x} \text{ (se } |x| < 1\text{)} \quad \sum_{i=1}^n \log(i) = \log(n!)$$

Stirling approx. $n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$ per $n \rightarrow \infty$. Da questo si può dimostrare che $\log(n!) \sim \infty \log n$.

Definizioni di O -grande, Ω -grande, Θ -grande

O -grande

$O(g(n))$ è l'insieme delle funzioni che sono asintoticamente dominate da $g(n)$ a meno di una costante.

$$O(g(n)) = \{f(n) \mid \exists c > 0, n_0 > 0 \text{ tali che } \forall n \geq n_0, f(n) \leq c \cdot g(n)\}$$

Ω -grande

$\Omega(g(n))$ è l'insieme delle funzioni che dominano asintoticamente $g(n)$ a meno di una costante.

$$\Omega(g(n)) = \{f(n) \mid \exists c > 0, n_0 > 0 \text{ tali che } \forall n \geq n_0, f(n) \geq c \cdot g(n)\}$$

Θ -grande

$\Theta(g(n))$ è l'insieme delle funzioni che hanno approssimativamente lo stesso comportamento asintotico di $g(n)$.

$$\Theta(g(n)) = \{f(n) \mid \exists c_1 > 0, c_2 > 0, n_0 > 0 \text{ tali che } \forall n \geq n_0, c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$$

N.B. Spesso si utilizza la notazione $f(n) = O(g(n))$ invece di $f(n) \in O(g(n))$ per indicare che la funzione $f(n)$ appartiene all'insieme di funzioni $O(g(n))$ anche se non è una notazione formalmente corretta.

Criterio di costo costante e logaritmico

Nella valutazione della complessità algoritmica con criterio di costo costante, ogni istruzione ha costo $\Theta(1)$. Con il criterio di costo logaritmico, invece, leggere (READ), copiare (LOAD), spostare (STORE), scrivere (WRITE), eseguire somme (ADD) e sottrazioni (SUB) ha costo $\Theta(\log(n))$, eseguire moltiplicazioni (MUL) e divisioni (DIV) ha costo $\Theta(\log^2(n))$ e il resto (JUMP/HALT) ha costo $\Theta(1)$.

Esempio Calcolo di 2^{2^n} con macchina RAM con criterio di costo logaritmico.

1	READ 2	$\log(n)$
2	LOAD= 2	$\log(2) = k$
3	STORE 1	$\log(2) = k$
4	loop: LOAD 1	$\log(2^{2^n-1}) = 2^{n-1}$
5	MUL 1	$(\log(2^{2^n-1}))^2 = (2^{n-1})^2 = 2^{2n-2}$
6	STORE 1	$\log(2^{2n}) = 2n$
7	LOAD 2	$\log(n)$
8	SUB= 1	$\log(n)$
9	STORE 2	$\log(n-1)$
10	JGT loop	1
11	WRITE 1	$\log(2^n) = 2^n$
12	HALT	1

$$T(n) = \log(n) + n(2^{n-1} + 2^{2n-2} + 2^n + 3\log(n)) + 2^n = \Theta(n2^{2n-2})$$

Ricorrenze

Master theorem

Data l'equazione di ricorrenza $T(n) = aT(\frac{n}{b}) + f(n)$ con $a \geq 1, b > 1$.

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{se } f(n) = O(n^c) \text{ con } c < \log_b a \\ \Theta(n^{\log_b a} \log n) & \text{se } f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)) & \text{se } f(n) = \Omega(n^c) \text{ con } c > \log_b a \text{ e } af(\frac{n}{b}) \leq kf(n) \end{cases}$$

Per qualche $k < 1$.

Generalizzazione del caso 2

Se $f(n) = \Theta(n^{\log_b a} \log^k n)$ per qualche $k \in \mathbb{R}$, allora:

$$T(n) = \begin{cases} \Theta(n^{\log_b a} \log^{k+1} n) & \text{se } k > -1 \\ \Theta(n^{\log_b a} \log \log n) & \text{se } k = -1 \\ \Theta(n^{\log_b a}) & \text{se } k < -1 \end{cases}$$

Notiamo che se $k = 0$ ci troviamo nel caso semplice visto sopra.

Sostituzione

Consiste nell'intuire una soluzione, sostituirla nell'equazione di ricorrenza $T(n)$ e verificare che esistano c e n_0 che rispettino la definizione di O -grande.

Esempio Consideriamo $T(n) = 3T(\log(n)) + 2^n$. Intuiamo che $T(n) = O(2^n)$. Verifichiamo che esistano $c > 0, n_0 > 0$ tale che $\forall n \geq n_0, T(n) \leq c2^n$ da cui segue che $T(\log(n)) \leq c2^{\log n} = cn$

$$T(n) = 3T(\log(n)) + 2^n \leq 3cn + 2^n \leq \frac{c}{2}2^n + 2^n = (\frac{c}{2} + 1)2^n \leq c2^n. \text{ Si, basta prendere } c \geq 2. \square$$

L'intuizione per la O -grande si può ottenere espandendo le chiamate ricorsive e individuando il termine dominante o trovando due funzioni tra cui $T(n)$ è compresa. In generale, per dimostrare che $T(n) \in \Theta(f(n))$ bisogna dimostrare che $T(n) \in O(f(n))$ e che $T(n) \in \Omega(f(n))$ in modo analogo a quanto visto nell'esempio.

Strutture dati

Vettori

I vettori sono strutture dati che consentono l'accesso diretto ad ogni elemento data la sua posizione.

Operazione	Non ord.	Ord.
Search	$\Theta(n)$	$\Theta(\log(n))$
Minimum	$\Theta(n)$	$\Theta(1)$
Maximum	$\Theta(n)$	$\Theta(1)$
Successor	$\Theta(n)$	$\Theta(\log(n))$
Insert	$\Theta(n)$	$\Theta(n)$
Delete	$\Theta(n)$ (oppure $O(1)$ usando dei simboli di cella vuota)	$\Theta(n)$

L'inserimento in un vettore pieno può essere rifiutato $O(1)$ o può causare una riallocazione $O(n)$.

Liste

Le liste semplici sono strutture dati che memorizzano elementi sparsi in memoria dove ogni elemento contiene un riferimento al successivo.

Operazione	Non ord.	Ord.
Search	$\Theta(n)$	$\Theta(n)$
Minimum	$\Theta(n)$	$\Theta(1) \circ \Theta(n)$
Maximum	$\Theta(n)$	$\Theta(n) \circ \Theta(1)$
Successor	$\Theta(n)$	$\Theta(n)$
Insert	$O(1)$	$O(n)$
Delete	$O(n)$	$O(n)$

A seconda di come è ordinata la lista, uno tra **Minimum** e **Maximum** è $\Theta(1)$ e l'altro è $\Theta(n)$.

Pile (Stack)

Le pile sono strutture dati con le seguenti operazioni:

- Push(S, e) aggiunge l'elemento e in cima alla pila S
- Pop(S) restituisce e cancella l'elemento in cima alla pila
- Empty(S) restituisce true se la pila è vuota

Implementazione con vettori

Possono essere implementate con dei vettori memorizzando l'indice della cima della pila (Top of Stack, ToS).

- Push(S, e) se c'è spazio incrementa ToS e salva e in A[ToS] $O(1)$ altrimenti rifiuta $O(1)$ o rialloca $O(n)$
- Pop(S) restituisce A[ToS] e decrementa ToS $O(1)$
- Empty(S) restituisce true se ToS=0 $O(1)$

Implementazione con liste

Possono essere implementate con delle liste dove la testa della lista è la cima della pila.

- Push(S, e) inserisce e in testa alla lista $O(1)$
- Pop(S) restituisce e cancella l'elemento in testa alla lista $O(1)$
- Empty(S) restituisce true se il successore della testa è NIL $O(1)$

Code (Queues)

Le code sono strutture dati con le seguenti operazioni:

- Enqueue(Q, e) aggiunge l'elemento e alla fine della coda Q
- Dequeue(Q) restituisce e cancella l'elemento all'inizio della coda
- Empty(Q) restituisce true se la coda è vuota

Implementazione con vettori

Possono essere implementate con dei vettori tenendo traccia della posizione dove va inserito un nuovo elemento e di quella dell'elemento più vecchio con due indici tail e head e del numero di elementi contenuti n. Sia A il vettore e $l := A.length$

- Enqueue(Q, e) se $n < l$ salva e in A[tail] e incrementa n e tail

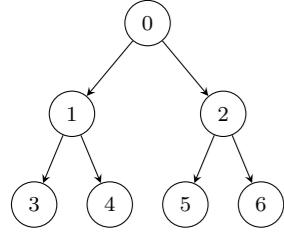
Alberi binari

Un albero binario è un albero in cui ogni nodo ha al più due figli. Dato un nodo A:

A.left ref. a figlio sinistro A.right ref. a figlio destro
A.p ref. al padre A.root ref. alla radice

A.p è NIL solo per la radice.

Implementazione con vettore



- La radice ha indice 0.
- Dato un nodo con indice i , il suo figlio sinistro ha indice $2i + 1$ e il suo figlio destro ha indice $2i + 2$
- Il padre del nodo con indice i ha indice $\lfloor \frac{i-1}{2} \rfloor$

Visita di un albero

Le principali procedure di visita di un albero sono INORDER, PREORDER e POSTORDER.

INORDER(T)	PREORDER(T)	POSTORDER(T)
1 if $T \neq \text{NIL}$	1 if $T \neq \text{NIL}$	1 if $T \neq \text{NIL}$
2 INORDER($T.\text{left}$)	2 PRINT($T.\text{key}$)	2 POSTORDER($T.\text{left}$)
3 PRINT($T.\text{key}$)	3 PREORDER($T.\text{left}$)	3 POSTORDER($T.\text{right}$)
4 INORDER($T.\text{right}$)	4 PREORDER($T.\text{right}$)	4 PRINT($T.\text{key}$)
5 return	5 return	5 return
3, 1, 4, 0, 5, 2, 6	0, 1, 3, 4, 2, 5, 6	3, 4, 1, 5, 6, 2, 0

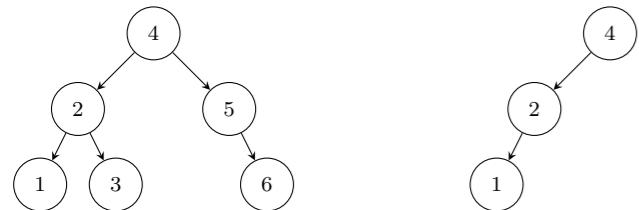
Tutte queste procedure hanno complessità temporale $\Theta(n)$ in quanto toccano ogni nodo dell'albero esattamente una volta.

Alberi binari di ricerca (BST)

Un albero binario di ricerca è un albero binario che, per ogni nodo x , soddisfa le seguenti condizioni:

- Se y è contenuto nel sottoalbero sinistro di x , allora $y.\text{key} \leq x.\text{key}$
- Se y è contenuto nel sottoalbero destro di x , allora $y.\text{key} \geq x.\text{key}$

Esempi



Operazioni su BST

SEARCH(T, x) : $O(h)$
1 if $T = \text{NIL}$ or $T.\text{key} = x.\text{key}$
2 return T
3 if $T.\text{key} < x.\text{key}$
4 return SEARCH($T.\text{right}, x$)
5 else return SEARCH($T.\text{left}, x$)

MAX(T) : $O(h)$
1 cur $\leftarrow T$
2 while $cur.\text{right} \neq \text{NIL}$
3 cur $\leftarrow cur.\text{right}$
4 return cur

MIN(T) : $O(h)$
1 cur $\leftarrow T$
2 while $cur.\text{left} \neq \text{NIL}$
3 cur $\leftarrow cur.\text{left}$
4 return cur

SUCCESSOR(x) : $O(h)$
1 if $x.\text{right} \neq \text{NIL}$
2 return MIN($x.\text{right}$)
3 cur $\leftarrow x.\text{right}$
4 while $y = \text{NIL}$ and $y.\text{right} = x$
5 $x \leftarrow y$
6 $y \leftarrow y.p$
7 return y

BUILDMAXHEAP(A) : $O(n)$
1 $A.\text{heapsize} \leftarrow A.\text{length}$
2 for $i \leftarrow \lfloor \frac{A.\text{length}}{2} \rfloor$ down to 1
3 MAXHEAPIFY(A, i)

EXTRACTMAX(A) : $O(\log(n))$
1 if $A.\text{heapsize} < 1$
2 return \perp
3 $max \leftarrow A[1]$
4 $A[1] \leftarrow A[A.\text{heapsize}]$
5 $A.\text{heapsize} \leftarrow A.\text{heapsize} - 1$
6 MAXHEAPIFY($A, 1$)
7 return max

MAX(A) : $O(1)$
1 return $A[1]$

INSERT(A, key) : $O(\log(n))$
1 $A.\text{heapsize} \leftarrow A.\text{heapsize} + 1$
2 $A[A.\text{heapsize}] \leftarrow key$
3 $i \leftarrow A.\text{heapsize}$
4 while $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
5 $SWAP(A[\text{PARENT}(i)], A[i])$
6 $i \leftarrow \text{PARENT}(i)$

Se il grafo è denso, cioè $|E| \approx |\mathbf{V}|^2$, è conveniente usare la matrice di adiacenza. Se il grafo è sparso, cioè $|E| \ll |\mathbf{V}|^2$, è conveniente usare le liste di adiacenza.

INSERT(T, x) : $O(h)$

```

1 pre  $\leftarrow \text{NIL}$ 
2 cur  $\leftarrow T.\text{root}$ 
3 while cur  $\neq \text{NIL}$ 
4 pre  $\leftarrow cur$ 
5 if  $x.\text{key} < cur.\text{key}$ 
6 cur  $\leftarrow cur.\text{left}$ 
7 cur  $\leftarrow cur.\text{right}$ 
8 x.p  $\leftarrow pre$ 
9 if pre  $= \text{NIL}$ 
10 T.root  $\leftarrow x$ 
11 elseif  $x.\text{key} < pre.\text{key}$ 
12 pre.left  $\leftarrow x$ 
13 else pre.right  $\leftarrow x$ 
14 if del  $\neq x$ 
15 else x.key  $\leftarrow del.\text{key}$ 
16 FREE(del)

```

DELETE(T, x) : $O(h)$

```

1 if  $x.\text{left} = \text{NIL}$  or  $x.\text{right} = \text{NIL}$ 
2 del  $\leftarrow x$ 
3 else del  $\leftarrow \text{SUCCESSOR}(x)$ 
4 if del.left  $\neq \text{NIL}$ 
5 subtree  $\leftarrow del.\text{left}$ 
6 else subtree  $\leftarrow del.\text{right}$ 
7 if subtree  $\neq \text{NIL}$ 
8 subtree.p  $\leftarrow del.p$ 
9 if del.p  $= \text{NIL}$ 
10 T.root  $\leftarrow subtree$ 
11 elseif del  $= del.p.\text{left}$ 
12 del.p.left  $\leftarrow subtree$ 
13 else del.p.right  $\leftarrow subtree$ 
14 if del  $\neq x$ 
15 else x.key  $\leftarrow del.\text{key}$ 
16 FREE(del)

```

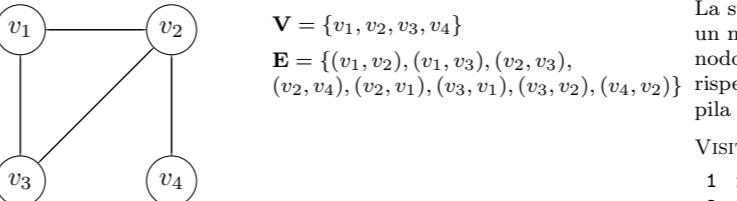
Grafi

Un grafo è una coppia $\mathcal{G} = (\mathbf{V}, \mathbf{E})$ con \mathbf{V} un insieme di nodi (detti anche vertici) ed \mathbf{E} un insieme di archi (detti anche lati).

Def.

- Un grafo con $|\mathbf{V}|$ nodi ha al più $|\mathbf{V}|^2$ archi
- Due nodi collegati da un arco si dicono *adiacenti*
- Un grafo è detto *orientato* (directed) se gli archi (arcs) che collegano i nodi sono orientati
- Un grafo è detto *connesso* se esiste un percorso per ogni coppia di nodi
- Un grafo è detto *completo* (o *completamente connesso*) se esiste un arco per ogni coppia di nodi.
- Un percorso è un *ciclo* se il nodo di inizio e fine coincidono. Un grafo privo di cicli si dice *acyclico*
- Un *cammino* tra due nodi v_1, v_2 è un insieme di archi di cui il primo ha origine in v_1 , l'ultimo termina in v_2 e ogni nodo compare almeno una volta sia come destinazione di un arco che come sorgente

Esempio Cammino $v_3 \rightarrow v_4 : (v_3, v_2), (v_2, v_4)$

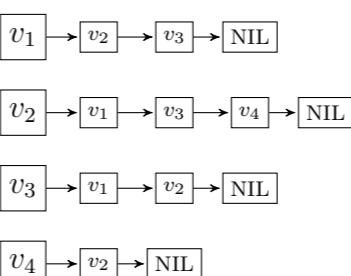


Rappresentazione in memoria

I grafi vengono rappresentati in memoria con *liste di adiacenza* o *matrice di adiacenza*.

Liste di adiacenza

Vettore di liste lungo $|\mathbf{V}|$, indicizzato dai nomi dei nodi dove ogni lista contiene i nodi adiacenti all'indice della sua testa.



Matrice di adiacenza

Matrice $|\mathbf{V}| \times |\mathbf{V}|$ di valori booleani con righe e colonne indicate dai nomi dei nodi dove la cella alla riga i e colonna j contiene 1 se l'arco $(v_i, v_j) \in \mathbf{E}$ e 0 altrimenti.

$$\begin{matrix} & v_1 & v_2 & v_3 & v_4 \\ v_1 & 0 & 1 & 1 & 0 \\ v_2 & 1 & 0 & 1 & 1 \\ v_3 & 1 & 1 & 0 & 0 \\ v_4 & 0 & 1 & 0 & 0 \end{matrix}$$

Confronto

	Liste	Matrice
Complessità spaziale	$\Theta(\mathbf{V} + \mathbf{E})$	$\Theta(\mathbf{V} ^2)$
Determinare se $(v_1, v_2) \in \mathbf{E}$	$O(\mathbf{V})$	$O(1)$
Num. di archi o_e uscenti da un nodo	$\Theta(o_e)$	$O(\mathbf{V})$

Esempio Cammini: $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5$ (lungo 4 archi) e $v_1 \rightarrow v_4 \rightarrow v_5$ (lungo 2 archi).

Operazioni su grafi

Visita in ampiezza (BFS o Breadth-first search)

La strategia di visita in ampiezza visita tutti i nodi di un grafo \mathcal{G} a partire da un nodo sorgente s . I nodi vengono visitati in ordine di distanza, dove i nodi più vicini al nodo sorgente vengono visitati per primi.

VISITAAMPIEZZA(G, s) : $O(|\mathbf{V}| + |\mathbf{E}|)$

```

1 for each  $n \in \mathbf{V} \setminus \{s\}$ 
2 n.color  $\leftarrow \text{white}$ 
3 n.dist  $\leftarrow \infty$ 
4 s.color  $\leftarrow \text{grey}$ 
5 s.dist  $\leftarrow 0$ 
6 Q  $\leftarrow \emptyset$ 
7 ENQUEUE(Q, s)
8 while  $\neg \text{ISEMPTY}(Q)$ 
9 cur  $\leftarrow \text{DEQUEUE}(Q)$ 
10 for each  $v \in cur.\text{adiacenti}$ 
11 if v.color = white
12 v.color  $\leftarrow \text{grey}$ 
13 v.dist  $\leftarrow cur.\text{dist} + 1$ 
14 ENQUEUE(Q, v)
15 cur.color  $\leftarrow \text{black}$ 

```

Visita in profondità (DFS o Depth-first search)

La strategia di visita in profondità visita tutti i nodi di un grafo \mathcal{G} a partire da un nodo sorgente s . I nodi vengono visitati in profondità, cioè che si visita un nodo fino a quando non si raggiungono i nodi più lontani (in profondità) rispetto al nodo sorgente. Il codice è uguale a quello per il BFS usando una pila invece che una coda.

VISITAPROFONDITA(G, s) : $O(|\mathbf{V}| + |\mathbf{E}|)$

```

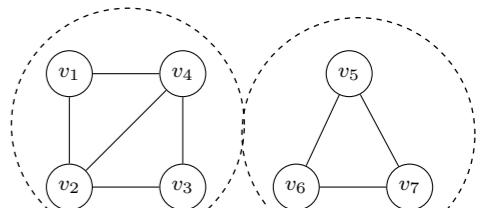
1 for each  $n \in \mathbf{V} \setminus \{s\}$ 
2 n.color  $\leftarrow \text{white}$ 
3 n.dist  $\leftarrow \infty$ 
4 s.color  $\leftarrow \text{grey}$ 
5 s.dist  $\leftarrow 0$ 
6 S  $\leftarrow \emptyset$ 
7 PUSH(S, s)
8 while  $\neg \text{ISEMPTY}(S)$ 
9 cur  $\leftarrow \text{POP}(S)$ 
10 for each  $v \in cur.\text{adiacenti}$ 
11 if v.color = white
12 v.color  $\leftarrow \text{grey}$ 
13 v.dist  $\leftarrow cur.\text{dist} + 1$ 
14 PUSH(S, v)
15 cur.color  $\leftarrow \text{black}$ 

```

Componenti connesse

Una *componente连通* di un grafo è un insieme \mathbf{S} di nodi tali per cui esiste un cammino tra ogni coppia di essi e nessuno di essi è connesso a nodi $\notin \mathbf{S}$.

Esempio Componenti connesse $S_1 = \{v_1, v_2, v_3, v_4\}, S_2 = \{v_5, v_6, v_7\}$



COMPONENTICONNESSE(G) : $O(|\mathbf{V}| + |\mathbf{E}|)$

```

1 for each  $v \in \mathbf{V}$ 
2 v.etichetta  $\leftarrow -1$ 
3 eti  $\leftarrow 1$ 
4 for each  $v \in \mathbf{V}$ 
5 if v.etichetta = -1
6 VISITAETICHETTA(G, v, eti)
7 eti  $\leftarrow eti + 1$ 

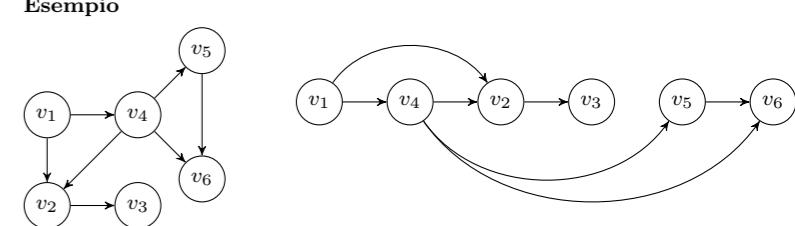
```

VISITAETICHETTA funziona come VISITAAMPIEZZA o VISITAPROFONDITA ma imposta a eti il campo *etichetta* del nodo visitato.

Ordinamento topologico

L'ordinamento topologico è una sequenza di nodi di un grafo *orientato aciclico* tale per cui nessun nodo compare prima di un suo predecessore.

Esempio



```

ORDINAMENTOTOPLOGICO( $G$ )
1 for each  $v \in V$ 
2    $v.color \leftarrow white$ 
3 for each  $v \in V$ 
4   if  $v.color = white$ 
5     VISITAPROFOT( $G, v, L$ )
6 return  $L$ 

VISITAPROFOT( $G, s, L$ )
1  $s.color \leftarrow grey$ 
2 for each  $v \in cur.adiacenti$ 
3   if  $v.color = white$ 
4     VISITAPROFOT( $G, v, L$ )
5    $s.color \leftarrow black$ 
6 PUSHFRONT( $L, s$ )

```

Percorso più breve

Dato un grafo e un suo nodo s , individua i percorsi più brevi da s a qualunque altro nodo.

DIJKSTRAQUEUE(G, s) : $O((|V| + |E|) \log(|V|))$

```

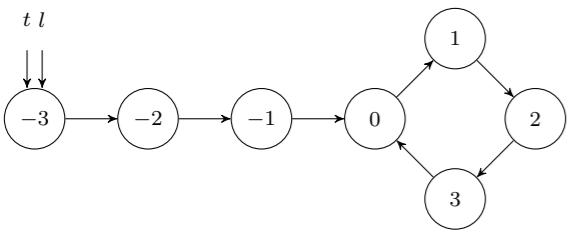
1  $Q \leftarrow \emptyset$ 
2  $s.dist \leftarrow 0$ 
3 for each  $v \in |V|$ 
4   if  $v \neq s$ 
5      $v.dist \leftarrow \infty$ 
6      $v.pred \leftarrow NIL$ 
7   ACCODAPRI( $Q, v, v.dist$ )
8 while  $Q \neq \emptyset$ 
9    $u \leftarrow CANCELLAMIN(Q)$ 
10  for each  $v \in u.succ$ 
11     $ndis \leftarrow u.dist + peso(u, v)$ 
12    if  $v.dist > ndis$ 
13       $v.dist \leftarrow ndis$ 
14       $v.prev \leftarrow u$ 
15    DECREMENTAPRI( $Q, v, ndis$ )

```

Rilevamento cicli (Floyd lepre e tartaruga)

Dato un grafo orientato, per cui ogni nodo ha un solo successore determina, dato un nodo di partenza s , se il cammino che parte da s ha cicli.

Grafo di esempio



L'idea è quella di usare due riferimenti t e l che partono dal nodo iniziale (-3 nell'esempio) che vengono spostati a ogni passo. Nello specifico, t viene spostato dal nodo a cui punta al successore e l viene spostato dal nodo a cui punta al successore del successore. È garantito che se c'è un ciclo, i due riferimenti si incontreranno.

Sia C la lunghezza del ciclo (4 nell'esempio) e T la lunghezza della "coda" che lo precede (3 nell'esempio), allora l'algoritmo è il seguente:

FLOYDLT(G, x) : $\Theta(2(T + C) - r)$

```

1  $t \leftarrow x.succ$ 
2  $l \leftarrow x.succ.succ$ 
3 while  $l \neq t$ 
4    $t \leftarrow t.succ$ 
5    $l \leftarrow l.succ.succ$ 
6    $T \leftarrow 0$ 
7    $t \leftarrow x$ 
8   while  $l \neq t$ 
9      $t \leftarrow t.succ$ 
10     $l \leftarrow l.succ$ 
11     $T \leftarrow T + 1$ 
12     $l \leftarrow t$ 
13     $C \leftarrow 0$ 
14    while  $l \neq t$ 
15       $l \leftarrow l.succ$ 
16       $C \leftarrow C + 1$ 
17 return  $T, C$ 

```

Algoritmi di ordinamento

Bubble sort

Bubble Sort funziona passando ripetutamente attraverso l'array da ordinare, confrontando gli elementi adiacenti e scambiandoli se sono nell'ordine sbagliato. Prosegue attraverso l'array più volte fino a quando non sono più necessari scambi, indicando che l'array è ordinato.

BUBBLESORT(A) : $O(n^2)$

```

1 for  $i \leftarrow 1$  to  $A.length - 1$ 
2   for  $j \leftarrow 1$  to  $A.length - i$ 
3     if  $A[j] > A[j + 1]$ 
4       SWAP( $A[j], A[j + 1]$ )

```

Selection sort

Selection Sort seleziona ripetutamente l'elemento più piccolo dall'array non ordinato e lo scambia con l'elemento nella posizione corrente. Continua questo processo, aumentando progressivamente la porzione ordinata dell'array fino a quando l'intero array è ordinato.

```

SELECTIONSORT( $A$ ) :  $O(n^2)$ 
1 for  $i \leftarrow 1$  to  $A.length - 1$ 
2    $min\_index \leftarrow i$ 
3   for  $j \leftarrow i + 1$  to  $A.length$ 
4     if  $A[j] < A[min\_index]$ 
5        $min\_index \leftarrow j$ 
6     SWAP( $A[i], A[min\_index]$ )

```

Insertion sort

Insertion Sort costruisce l'array ordinato uno alla volta. Prende ciascun elemento dall'input e lo inserisce nella sua posizione corretta rispetto agli elementi già ordinati. Questo processo continua fino a quando l'intero array è ordinato.

```

INSERTIONSORT( $A$ ) :  $O(n^2)$ 
1 for  $i \leftarrow 2$  to  $A.length$ 
2    $tmp \leftarrow A[i]$ 
3    $j \leftarrow i - 1$ 
4   while  $j \geq 1$  and  $A[j] > tmp$ 
5      $A[j + 1] \leftarrow A[j]$ 
6      $j \leftarrow j - 1$ 
7    $A[j + 1] \leftarrow tmp$ 

```

Merge sort

Merge Sort è un algoritmo di divide et impera. Divide l'array di input in due metà, ordina ricorsivamente ciascuna metà e poi fonde le due metà ordinate in un singolo array ordinato.

```

MERGESORT( $A, p, r$ ) :  $\Theta(n \log n)$ 
1 if  $p < r - 1$ 
2    $q \leftarrow \lfloor \frac{p+r}{2} \rfloor$ 
3   MERGESORT( $A, p, q$ )
4   MERGESORT( $A, q + 1, r$ )
5   MERGE( $A, p, q, r$ )
6   else if  $A[p] > A[r]$ 
7      $tmp \leftarrow A[r]$ 
8      $A[r] \leftarrow A[p]$ 
9      $A[p] \leftarrow tmp$ 
10  MERGE( $A, p, q, r$ )
11  for  $i \leftarrow 1$  to  $len_1$ 
12     $L[i] \leftarrow A[p + i - 1]$ 
13  for  $i \leftarrow 1$  to  $len_2$ 
14     $R[i] \leftarrow A[q + i]$ 
15   $L[len_1 + 1] \leftarrow \infty$ 
16   $R[len_2 + 1] \leftarrow \infty$ 
17   $i \leftarrow 1$ 
18  for  $k \leftarrow p$  to  $r$ 
19    if  $L[i] \leq R[j]$ 
20       $A[k] \leftarrow L[i]$ 
21       $i \leftarrow i + 1$ 
22    else
23       $A[k] \leftarrow R[j]$ 
24       $j \leftarrow j + 1$ 

```

Quick sort

Quick Sort è anch'esso un algoritmo di divide et impera. Sceglie un elemento pivot e suddivide l'array intorno a esso, in modo che gli elementi più piccoli del pivot siano a sinistra e quelli più grandi a destra. Ordina ricorsivamente le sotto-liste formate dalla partizione fino a quando l'intero array è ordinato.

Quick sort con schema di partizione di Lomuto

Lo schema di partizione di Lomuto seleziona l'ultimo elemento dell'array come pivot. Successivamente, scorre l'array e posiziona gli elementi più piccoli del pivot a sinistra e quelli più grandi a destra. Infine, sposta il pivot nella sua posizione corretta tra i due gruppi.

```

QUICKSORT( $A, lo, hi$ )
1 if  $lo < hi$ 
2    $p \leftarrow PARTLOMUTO( $A, lo, hi$ )$ 
3   QUICKSORT( $A, lo, p - 1$ )
4   QUICKSORT( $A, p + 1, hi$ )
PARTLOMUTO( $A, lo, hi$ )
1  $pivot \leftarrow A[hi]$ 
2  $i \leftarrow lo - 1$ 
3 for  $j \leftarrow lo$  a  $hi - 1$ 
4   if  $A[j] \leq pivot$ 
5      $i \leftarrow i + 1$ 
6     SWAP( $A[i], A[j]$ )
7   SWAP( $A[i + 1], A[hi]$ )
8 return  $i + 1$ 

```

Quick sort con schema di partizione di Hoare

Lo schema di partizione di Hoare seleziona il primo elemento dell'array come pivot. Utilizza due indici, uno che procede dall'inizio dell'array verso destra e uno che procede dalla fine dell'array verso sinistra, scambiando gli elementi fuori posizione fino a quando i due indici si incontrano. Alla fine, il pivot viene inserito tra i due gruppi e l'indice di partizione è determinato.

Quicksort

```

QUICKSORT( $A, lo, hi$ )
1 if  $lo < hi$ 
2    $p \leftarrow PARTHOARE( $A, lo, hi$ )$ 
3   QUICKSORT( $A, lo, p$ )
4   QUICKSORT( $A, p + 1, hi$ )
PARTHOARE( $A, lo, hi$ )
1  $pivot \leftarrow A[lo]$ 
2  $i \leftarrow lo - 1$ 
3  $j \leftarrow hi + 1$ 
4 while true
5   repeat
6      $j \leftarrow j - 1$ 
7     until  $A[j] \leq pivot$ 
8   repeat
9      $i \leftarrow i + 1$ 
10    until  $A[i] \geq pivot$ 
11   if  $i < j$ 
12     SWAP( $A[i], A[j]$ )
13   else return  $j$ 

```

Lo schema di partizione di Hoare effettua in media $\frac{1}{3}$ degli scambi effettuati dallo schema di partizione di Lomuto. (Entrambi hanno complessità $\Theta(n)$)

Heap sort

Heap Sort ordina l'array costruendo un max heap. Successivamente, estrae ripetutamente l'elemento massimo dall'heap e lo posiziona alla fine dell'array. Questo processo continua fino a quando l'intero array è ordinato.

HEAPSORT(A): $\Theta(n \log n)$

```

1 BUILDMAXHEAP( $A$ )
2 for  $i \leftarrow A.length$  to 2
3   swap( $A[1], A[i]$ )
4    $A.heapsize \leftarrow A.heapsize - 1$ 
5   MAXHEAPIFY( $A, 1$ )

```

Counting sort

Counting Sort è un algoritmo usato per ordinare un array di interi quando si conosce in anticipo il range dei valori possibili. Calcola un istogramma delle occorrenze di ciascun valore, quindi costruisce l'array ordinato posizionando ogni elemento nella sua posizione corretta basata sul conteggio accumulato degli elementi precedenti.

Non stabile

COUNTINGSORT(A) : $O(n + k)$

```

1  $Ist[0..k] \leftarrow 0$ 
2 for  $i \leftarrow 0$  to  $A.length - 1$ 
3    $Ist[A[i]] \leftarrow Ist[A[i]] + 1$ 
4    $idxA \leftarrow 0$ 
5   for  $i \leftarrow 0$  to  $k$ 
6     while  $Ist[i] > 0$ 
7        $A[idxA] \leftarrow i$ 
8        $idxA \leftarrow idxA + 1$ 
9      $Ist[i] \leftarrow Ist[i] - 1$ 

```

Stabile

COUNTINGSORT(A) : $O(n + k)$

```

1  $B[0..A.length - 1] \leftarrow 0$ 
2  $Ist[0..k] \leftarrow 0$ 
3 for  $i \leftarrow 0$  to  $A.length - 1$ 
4    $Ist[A[i]] \leftarrow Ist[A[i]] + 1$ 
5    $sum \leftarrow 0$ 
6   for  $i \leftarrow 0$  to  $k$ 
7      $sum \leftarrow sum + Ist[i]$ 
8      $Ist[i] \leftarrow sum$ 
9   for  $i \leftarrow A.length - 1$  to 0
10     $idx \leftarrow Ist[A[i]]$ 
11     $B[idx - 1] \leftarrow A[i]$ 
12     $Ist[A[i]] \leftarrow Ist[A[i]] - 1$ 
13 return  $B$ 

```

Confronto

Confronto delle complessità temporali e spaziali degli algoritmi di ordinamento:

Algoritmo	Stabile?	$T(n)$ (pessimo)	$T(n)$ (ottimo)	$S(n)$
Bubble	✓	$\Theta(n^2)$	(inv.)	$\Theta(n)$
Selection	✗	$\Theta(n^2)$	$\Theta(n^2)$	$O(1)$
Insertion	✓	$\Theta(n^2)$	(inv.)	$\Theta(n)$
Merge	✓	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$
Quick	✗	$\Theta(n^2)$	(ord.)	$\Omega(n \log n)$
Heap	✗	$\Theta(n \log n)$	$\Theta(n \log n)$	$O(1)$
Counting	✓	$O(n + k)$	$O(n + k)$	$O(n + k)$

N.B. Il quick sort nel caso medio ha complessità $\Theta(n \log n)$