

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	numeri di elementi della stringa $ abc = 3, \epsilon = 0$
concatenazione	unione di due o più stringhe in una singola stringa <p>es. se $w_1 = abc$ e $w_2 = def$ allora $w_1.w_2 = abcdef$</p>
riflesso	inversione degli ordini dei caratteri di una stringa <p>$w = c_1...c_n \iff w^R = c_n...c_1$ con $c_i = 1$</p> <p>es. se $w = abc$ allora $w^R = cba$</p>
Σ^*	insieme di tutte le stringe su Σ <p>es. $\Sigma = \{0,1\}$, $\Sigma^* = \{\epsilon, 0, 1, 00, 01, \dots\}$</p>
Σ^+	insieme di tutte le stringe non vuote su Σ <p>es. $\Sigma = \{0,1\}$, $\Sigma^+ = \{0, 1, 00, 01, \dots\}$</p>

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 $i \in I, y \in I^*$

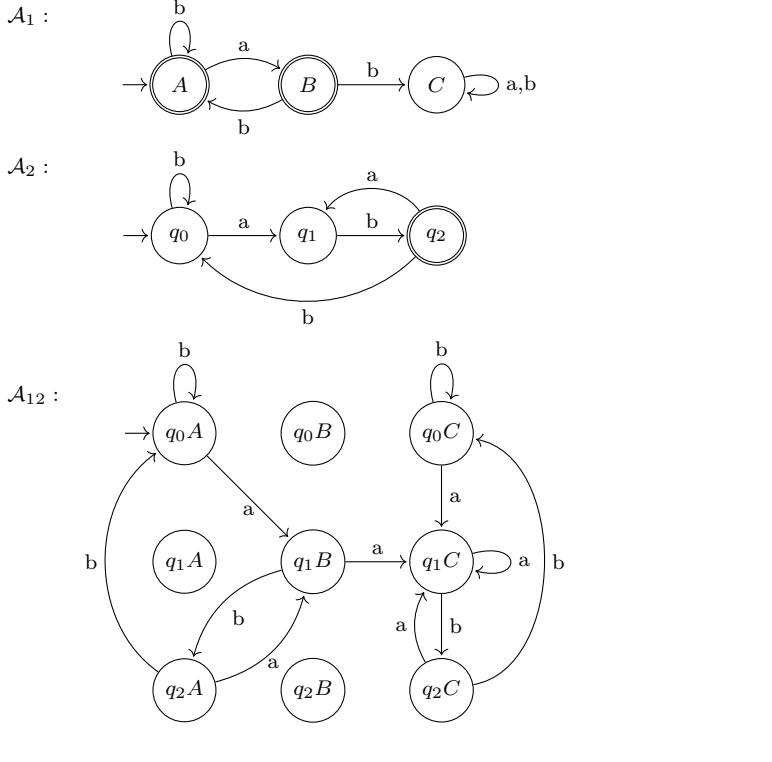
FSA traduttore

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

Operazioni su FSA

- 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.
- Dati due automi $\mathcal{A}_1, \mathcal{A}_2$ che riconoscono i linguaggi L_1, L_2 rispettivamente, si può costruire algorithmicamente 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 $q_0A, q_1A, q_2A, q_0B, q_1B, q_2B, q_2A, q_2B, q_2C$ 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:

- $|xy| \leq p$
- $y \neq \epsilon$
- $\forall i \geq 0, xy^iz \in L$

Esempio Dimostriamo che il linguaggio $L = \{0^n1^n | n \geq 0\}$ non è riconosciuto da un FSA usando il pumping lemma.

- Supponiamo per assurdo che L sia riconosciuto da un FSA e scegliamo $w = 0^p1^p$
- 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 $\alpha \geq 0, \beta \geq 1, \alpha + \beta \leq p$

Troviamo $i \geq 0$ tale che $xy^iz \notin L$

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

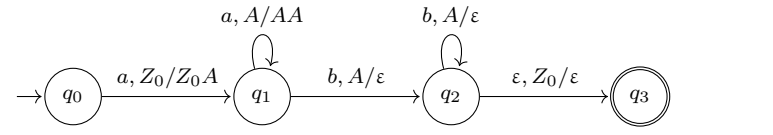
Automi a pila (AP/PDA)

Formalmente definiti come una settupla $(Q, I, \Gamma, \delta, q_0, Z_0, F)$ dove Q è un insieme finito di stati, I è l'alfabeto di input, Γ l'alfabeto di pila, $\delta: Q \times (I \cup \{\epsilon\}) \times \Gamma \rightarrow Q \times \Gamma^*$ è la funzione di transizione, che mappa lo stato corrente, un simbolo $I \in I$ e il simbolo $\gamma \in \Gamma$ in cima alla pila a un nuovo stato e una nuova stringa $\gamma' \in \Gamma^*$ che sostituisce la cima della pila, $q_0 \in Q$ è lo stato di partenza, Z_0 è il simbolo che denota il fondo della pila e $F \subseteq Q$ è l'insieme degli stati finali.

PDA riconoscitore

Un linguaggio L è riconosciuto da un PDA se e solo se $\forall s \in L$, dopo aver scandito la stringa s , si trova in uno stato $q_f \in F$.

Esempio Automa a pila che riconosce il linguaggio $L = \{a^n b^n \mid n > 0\}$



$\epsilon, Z_0/\epsilon$ è una ϵ -mossa, cioè una mossa che non consuma simboli in ingresso.

PDA traduttore

Un PDA traduttore è dotato di un nastro di uscita su cui può scrivere ad ogni transizione.

Macchine di Turing (MT/TM)

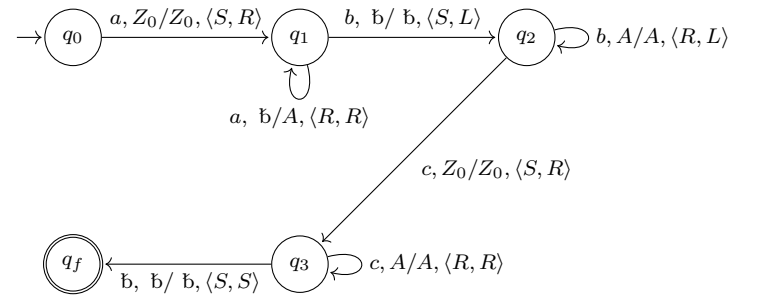
Le macchine di Turing a k nastri di memoria sono macchine dotate di un nastro di input e k nastri di memoria. Sono formalmente definite come una settupla $(Q, I, \Gamma, \mathfrak{b}, \delta, q_0, F)$ dove Q, I, q_0, F sono gli stessi di un PDA/FSA, Γ è l'alfabeto di nastro, $\mathfrak{b} \in \Gamma$ denota una cella di nastro vuota e $\delta: Q \times (I \cup \{\mathfrak{b}\}) \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, S, R\}^{k+1}$ è la funzione di transizione dove L_i, S_i, R_i denotano il movimento dell testina sull’ i -esimo nastro ($k + 1$ perché oltre ai k nastri di memoria ci si può muovere anche sul nastro di input). Le macchine di Turing a nastro singolo sono un modello di calcolo equivalente alle macchine di Turing a k nastri dove input e memoria si trovano su un unico nastro.

Def. Una macchina di Turing universale (MTU) è una macchina di Turing in grado di simulare qualsiasi altra macchina di Turing.

MT riconoscitore

Una MT riconosce il linguaggio L se per ogni stringa $w \in L$ si ferma in uno stato di accettazione in un numero finito di transizioni.

Esempio Macchina di Turing a $k = 1$ nastro di memoria che riconosce il linguaggio $L = \{a^n b^n c^n \mid n > 0\}$



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

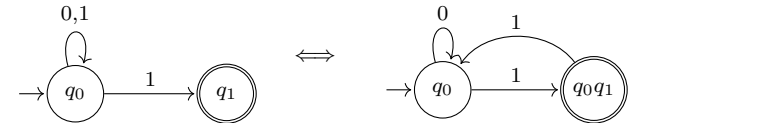
MT traduttore

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 espressive della loro versione deterministica. Gli automi a pila 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] ← In	WRITE* X	Out ← M[M[X]]
READ* X	M[M[X]] ← In	JUMP 1	PC ← <i>l</i>
LOAD X	M[0] ← M[X]	JZ 1	se M[0]=0
LOAD= X	M[0] ← X		PC ← <i>l</i>
LOAD* X	M[0] ← M[M[X]]	JGT 1	se M[0]>0
STORE X	M[X] ← M[0]		PC ← <i>l</i>
STORE* X	M[M[X]] ← M[0]	JGZ 1	se M[0]≥0
ADD X	M[0] ← M[0] + M[X]		PC ← <i>l</i>
SUB X	M[0] ← M[0] - M[X]	JLT 1	se M[0]<0
MUL X	M[0] ← M[0] × M[X]		PC ← <i>l</i>
DIV X	M[0] ← M[0] / M[X]	JLZ 1	se M[0]≤0
WRITE X	Out ← M[X]		PC ← <i>l</i>
WRITE= X	Out ← 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:

last(x) $\neg \exists y(y > x)$
y=S(x) $y = x + 1$

Logica monadica del II ordine (MSO)

La logica monadica del II ordine è un modello di calcolo con le stesse capacità espressive 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 $L = \{a^{2n} \mid n \in \mathbb{N}^+\}$.

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

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

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

Esempio Prendiamo in considerazione la grammatica $G = (\langle a, b \rangle, \{S, R\}, \{S \rightarrow aR, R \rightarrow bS, S \rightarrow \epsilon\}, S)$, abbiamo che $abaR \xRightarrow{G} ababS$. 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 $S \xRightarrow{*G} s$ dove $\xRightarrow{*G}$ è la chiusura riflessiva e transitiva di \xRightarrow{G} . A seconda delle limitazioni imposte sulle regole di produzione $\alpha \rightarrow \beta$ si hanno 4 tipi di grammatiche

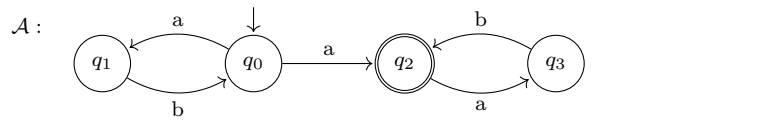
Tipo	Nome	Regole prod.	Equiv.
0	Non ristretta	nessuna	TM
1	Monotone oppure Context sensitive	$ \alpha \leq \beta $ oppure $\alpha = \gamma A \delta, \beta = \gamma \chi \delta$ con $\chi \neq \epsilon$	(LBA)
2	Context free	$ a = 1$	NPDA
3	Regolare	$A \rightarrow \alpha, A \rightarrow \alpha A$ o $A \rightarrow \alpha, A \rightarrow A\alpha$	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 linguaggio $L(\mathcal{A})$, è possibile ricavare la grammatica \mathcal{G} tale che $L(\mathcal{G}) = L(\mathcal{A})$ (o viceversa).

Esempio FSA Linguaggio $L = (ab)^*a(ab)^*$



$\mathcal{G} = (T = \{a, b\}, N = \{q_0, q_1, q_2, q_3\}, P, S = q_0)$

P :

$q_0 \rightarrow aq_1|aq_2|a$

$q_1 \rightarrow bq_0$

$q_2 \rightarrow aq_3$

$q_3 \rightarrow bq_2|b$

Proprietà di chiusura dei linguaggi

	REG	DCFL	CFL	CSL	RIC	RE
Unione	✓	×	✓	✓	✓	✓
Intersezione	✓	×	×	✓	✓	✓
Differenza	✓	×	×	✓	✓	×
Complemento	✓	✓	×	✓	✓	×
Concatenazione	✓	×	✓	✓	✓	✓
Star di Kleene	✓	×	✓	✓	✓	✓

REG	Linguaggi regolari (riconosciuti da una grammatica regolare / esprimibili mediante un’espressione regolare)
DCFL	Linguaggi context free deterministici (riconosciuti da una grammatica context free deterministica)
CFL	Linguaggi context free (riconosciuti da una grammatica context free)
CSL	Linguaggi context sensitive (riconosciuti da una grammatica context sensitive)
RIC	Linguaggi ricorsivi (esiste una MT che per ogni stringa appartenente al linguaggio, accetta o rifiuta)
RE	Linguaggi ricorsivamente enumerabili (riconosciuti da una grammatica non ristretta)

FIN (linguaggi finiti) \subset **REG** \subset **DCFL** \subset **CFL** \subset **CSL** \subset **RIC** \subset **RE**

Teoria della computazione

Tesi di Church	tutti i problemi computabili sono descrivibili da una macchina di Turing. <i>(MT, programma, algoritmo, procedura sono usati come sinonimi)</i>
problema	richiesta di calcolo algoritmico di $f: A \rightarrow B$ con $ A , B \leq \mathbb{N} $. (In generale f è parziale, cioè $\exists n(f(n) = \perp)$)
f. computabile	se esiste un algoritmo che calcola f , f è calcolabile e il problema è computabile. <i>f_i</i> denota l’ <i>i</i> -esima funzione computabile. <i>(calcolabile, computabile e risolvibile sono sinonimi)</i>
ins. dec.	un insieme <i>S</i> è decidibile (o ricorsivo) se esiste una funzione 1 _S che restituisce 1 se $n \in S$ e 0 se $n \notin S$.
ins. semidec.	un insieme <i>S</i> è semidecidibile (o ricorsivamente enumerabile o r.e.) se esiste una funzione 1’ _S (<i>n</i>) che restituisce 1 se $n \in S$.

Se un problema è chiuso, cioè che la sua soluzione non dipende da un valore in ingresso, allora è decidibile.

Teorema Sia *S* un insieme, allora:

S ric. $\iff S$ r.e. $\wedge S^C$ r.e.

Corollario Se $\neg S$ ric., allora:

(*S* r.e. $\wedge \neg S^C$ r.e.) \vee ($\neg S$ r.e. $\wedge S^C$ r.e.) \vee ($\neg S$ r.e. $\wedge \neg S^C$ r.e.)

Dimostrazione di semidecidibilità (Dovetailing)

Per stabilire se $\exists z \mid f_x(z) \neq \perp$ si può usare la tecnica del *dovetailing* che consiste nell’eseguire una sola mossa alla volta per ogni input fin quando non ci si arresta. Se $\exists z \mid f_x(z) \neq \perp$ sicuramente prima o poi viene incontrato e ci si arresta. Non si possono eseguire in sequenza $f_x(0)$ e poi $f_x(1)$ e così via perché se $f_x(0) = \perp$ non si arriverà mai ad eseguire $f_x(1)$.

Teorema di Rice

Sia *F* = {*f*_{*i*} | *i* ∈ ℕ} l'insieme di tutte le funzioni computabili, sia *P* ⊆ *F* = {*f*_{*i*} | *f*_{*i*} soddisfa una certa proprietà} un sottoinsieme di *F* di funzioni computabili che soddisfano una certa proprietà e sia *S* = {*x* | *f*_{*x*} ∈ *P*} l'insieme degli indici delle funzioni che appartengono a *P*. Allora *S* ric. ⇔⇒ *P* = ∅ ∨ *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 ≤ *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 ≤ *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 *M* e un input *w*, determinare se *M* si arresta su *w*.

H = {⟨*M*, *w*⟩ | *M* si arresta sull'input *w*}

Problema dell’equivalenza delle MT

Date due macchine di Turing *M*₁ e *M*₂, determinare se calcolano la stessa funzione.

EQ = {⟨*M*₁, *M*₂⟩ | *M*₁ e *M*₂ calcolano la stessa funzione}

Problema della totalità delle MT

Data una macchina di Turing *M*, determinare se *M* si arresta su tutti gli input.

TOT = {⟨*M*⟩ | *M* si arresta su tutti gli input}

Problema del linguaggio vuoto

Data una macchina di Turing *M*, determinare se il linguaggio riconosciuto da *M* è vuoto.

E = {⟨*M*⟩ | *L*(*M*) = ∅}

Analisi di complessità degli algoritmi

Formule note

∑

i
=
1

n

i
=

n
(
n
+
1
)

2

∑

i
=
1

n

i

2

=

n
(
n
+
1
)
(
2
n
+
1
)

6

∑

i
=
1

n

i

3

=

n

2

(
n
+
1

)

2

4

∑

i
=
1

n

x

i

=

x

n
+
1

x
−
1

−
1

∑

i
=
1

∞

x

i

=

1
1
−
x

(
se
|
x
|
<
1
)

∑

i
=
1

n

log
⁡
(
i
)
=
log
⁡
(
n
!
)

Stirling approx. *n*! ~ √2π*n*(

n
e

{\displaystyle {\frac {n}{e}}}

)^{*n*} per *n* → ∞. Da questo si può dimostrare che log(*n*!) ^{*n*→∞}~ *n* 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*) | ∃*c* > 0, *n*₀ > 0 tali che ∀*n* ≥ *n*₀, *f*(*n*) ≤ *c* · *g*(*n*)}

Ω-grande

Ω(*g*(*n*)) è l'insieme delle funzioni che dominano asintoticamente *g*(*n*) a meno di una costante.

Ω(*g*(*n*)) = {*f*(*n*) | ∃*c* > 0, *n*₀ > 0 tali che ∀*n* ≥ *n*₀, *f*(*n*) ≥ *c* · *g*(*n*)}

Θ-grande

Θ(*g*(*n*)) è l'insieme delle funzioni che hanno approssimativamente lo stesso comportamento astintotico di *g*(*n*).

Θ(*g*(*n*)) = {*f*(*n*) | ∃*c*₁ > 0, *c*₂ > 0, *n*₀ > 0 tali che ∀*n* ≥ *n*₀,

c

1

⋅
g
(
n
)
≤
f
(
n
)
≤

c

2

⋅
g
(
n
)

}

N.B. Spesso si utilizza la notazione *f*(*n*) = *O*(*g*(*n*)) invece di *f*(*n*) ∈ *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 Θ(1). Con il criterio di costo logaritmico, invece, leggere (**READ**), copiare (**LOAD**), spostare (**STORE**), scrivere (**WRITE**), eseguire somme (**ADD**) e sottrazioni (**SUB**) ha costo Θ(log(*n*)), eseguire moltiplicazioni (**MUL**) e divisioni (**DIV**) ha costo Θ(log²(*n*)) e il resto (**JUMP**/**HALT**) ha costo Θ(1).

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

1	READ	2	log(<i>n</i>)
2	LOAD=	2	log(2) = <i>k</i>
3	STORE	1	log(2) = <i>k</i>
4	loop=	LOAD 1	log(2 ^{2^{<i>n</i>}−1}) = 2 ^{<i>n</i>−1}
5	MUL	1	(log(2 ^{2^{<i>n</i>}−1})) ² = (2 ^{<i>n</i>} −1) ² = 2 ^{2<i>n</i>−2}
6	STORE	1	log(2 ^{2^{<i>n</i>}}) = 2 <i>n</i>
7	LOAD	2	log(<i>n</i>)
8	SUB=	1	log(<i>n</i>)
9	STORE	2	log(<i>n</i> − 1)
10	JGT	loop	1
11	WRITE	1	log(2 ^{2^{<i>n</i>}}) = 2 ^{<i>n</i>}
12	HALT		1

T(*n*) = log(*n*) + *n*(2^{*n*−1} + 2^{2*n*−2} + 2^{*n*} + 3 log(*n*)) + 2^{*n*} = Θ(*n*2^{2*n*−2})

Ricorrenze

Master theorem

Data l'equazione di ricorrenza *T*(*n*) = *aT*(

n

b

{\displaystyle {\frac {n}{b}}}

) + *f*(*n*) con *a* ≥ 1, *b* > 1.

T
(
n
)
=
⎧
⎩

Θ
(

n

log
⁡

b

a

)

se

f
(
n
)
=

O

(

n

c

)

con

c
<
log
⁡

b

a

Θ
(

n

log
⁡

b

a

log
⁡
n
)

se

f
(
n
)
=

Θ
(

n

log
⁡

b

a

)

Θ
(
f
(
n
)
)

se

f
(
n
)
=

Ω
(

n

c

)

con

c
>
log
⁡

b

a

e

a
f
(

n

b

)
≤
k
f
(
n
)

⎫
⎭

{\displaystyle \left\{\begin{array}{ll}\Theta (n^{\log _{b}a})& {\rm {se}}\,f(n)=O(n^{c})\,{\rm {con}}\,c<\log _{b}a\\ \Theta (n^{\log _{b}a}\log n)& {\rm {se}}\,f(n)=\Theta (n^{\log _{b}a})\\ \Theta (f(n))& {\rm {se}}\,f(n)=\Omega (n^{c})\,{\rm {con}}\,c>\log _{b}a\,{\rm {e}}\,af({\frac {n}{b}})\leq kf(n)\end{array}\right.}

Per qualche *k* < 1.

Generalizzazione del caso 2

Se *f*(*n*) = Θ(*n*^{log_{*b*} *a*} log^{*k*} *n*) per qualche *k* ∈ ℝ, allora:

T
(
n
)
=
⎧
⎩

Θ
(

n

log
⁡

b

a

log

k
+
1

n
)

se

k
>
−
1

Θ
(

n

log
⁡

b

a

log
⁡
n
)

se

k
=
−
1

Θ
(

n

log
⁡

b

a

)

se

k
<
−
1

⎫
⎭

{\displaystyle \left\{\begin{array}{ll}\Theta (n^{\log _{b}a}\log ^{k+1}n)& {\rm {se}}\,k>-1\\ \Theta (n^{\log _{b}a}\log n)& {\rm {se}}\,k=-1\\ \Theta (n^{\log _{b}a})& {\rm {se}}\,k<-1\end{array}\right.}

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

Sostituzione

Consiste nell'intuire una soluzione, sostituirla nell'equazione di ricorrenze *T*(*n*) e verificare che esistano *c* e *n*₀ che rispettino la definizione di *O*-grande.

Esempio Consideriamo *T*(*n*) = 3*T*(log(*n*)) + 2^{*n*}. Intuiamo che *T*(*n*) = *O*(2^{*n*}). Verifichiamo che esistano *c* > 0, *n*₀ > 0 tale che ∀*n* ≥ *n*₀, *T*(*n*) ≤ *c*2^{*n*} da cui segue che *T*(log(*n*)) ≤ *c*2^{log*n*} = *cn*

T(*n*) = 3*T*(log(*n*)) + 2^{*n*} ≤ 3*cn* + 2^{*n*} ≤

c
2

2

n

+

2

n

=
(

c
2

+
1
)

2

n

≤

c
2

n

?

 Sì, basta prendere *c* ≥ 2. □

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*) ∈ Θ(*f*(*n*)) bisogna dimostrare che *T*(*n*) ∈ *O*(*f*(*n*)) e che *T*(*n*) ∈ Ω(*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	<i>O</i> (<i>n</i>)	Θ(log(<i>n</i>))
Minimum	<i>O</i> (<i>n</i>)	Θ(1)
Maximum	<i>O</i> (<i>n</i>)	Θ(1)
Successor	<i>O</i> (<i>n</i>)	Θ(log(<i>n</i>))
Insert	<i>O</i> (<i>n</i>)	<i>O</i> (<i>n</i>)
Delete	<i>O</i> (<i>n</i>) (oppure <i>O</i> (1) usando dei simboli di cella vuota)	<i>O</i> (<i>n</i>)

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	<i>O</i> (<i>n</i>)	<i>O</i> (<i>n</i>)
Minimum	<i>O</i> (<i>n</i>)	Θ(1) o Θ(<i>n</i>)
Maximum	<i>O</i> (<i>n</i>)	Θ(<i>n</i>) o Θ(1)
Successor	<i>O</i> (<i>n</i>)	<i>O</i> (<i>n</i>)
Insert	<i>O</i> (1)	<i>O</i> (<i>n</i>)
Delete	<i>O</i> (<i>n</i>)	<i>O</i> (<i>n</i>)

A seconda di come è ordinata la lista, uno tra Minimum e Maximum è Θ(1) e l'altro è Θ(*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] <i>O</i> (1)
	altrimenti rifiuta <i>O</i> (1) o rialloca <i>O</i> (<i>n</i>)
Pop (S)	restituisce A [ToS] e decrementa ToS <i>O</i> (1)
Empty (S)	restituisce true se ToS=0 <i>O</i> (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 <i>O</i> (1)
Pop (S)	restituisce e cancella l'elemento in testa alla lista <i>O</i> (1)
Empty (S)	restituisce true se il successore della testa è NIL <i>O</i> (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 <i>n</i> < <i>l</i> salva e in A [tail] e incrementa <i>n</i> e tail <i>O</i> (1)
	altrimenti rifiuta <i>O</i> (1) o rialloca Θ(<i>n</i>)
Dequeue (Q)	restituisce A [head], decrementa <i>n</i> e incrementa head <i>O</i> (1)
Empty (Q)	restituisce true se <i>n</i> = 0 <i>O</i> (1)

Implementazione con liste

Possono essere implementate con delle liste tenendo traccia sia della testa (**head**) che della coda (**tail**) della lista.

Enqueue (Q , e)	inserisce l'elemento e in coda alla lista <i>O</i> (1)
Dequeue (Q)	restituisce e cancella l'elemento in testa alla lista <i>O</i> (1)
Empty (Q)	restituisce true se head = tail <i>O</i> (1)

Code doppie (Double-ended queues o Deques)

Le code doppie sono strutture dati in cui è possibile inserire sia in testa che in coda. Sono dotate delle seguenti operazioni:

PushFront (Q , e)	inserisce l'elemento e in testa
PushBack (Q , e)	inserisce l'elemento e in coda
PopFront (Q)	restituisce e cancella l'elemento in testa
PopBack (Q)	restituisce e cancella l'elemento in coda
Empty (Q)	restituisce true se la deque è vuota

Implementazione con vettori

PushBack, **PopFront** e **Empty** sono analoghi a **Enqueue** e **Dequeue** della coda realizzata con vettore.

PushFront (Q , e)	se <i>n</i> < <i>l</i> decr. head , ins. e in A [head] e incr. <i>n</i> <i>O</i> (1)
	altrimenti segnala l'errore <i>O</i> (1)
PopBack (Q)	se <i>n</i> > 0, restituisce A [tail] e decr. <i>n</i> e tail <i>O</i> (1)

Implementazione con liste *doppiamente concatenate*

PushBack e **PopFront** sono analoghi a **Enqueue** e **Dequeue** della coda realizzata con lista. Una lista vuota è rappresentata con **head** e **tail** che puntano l'uno all'altro.

PushFront (Q , e)	ins. e in testa aggiornando head e il suo succ. <i>O</i> (1)
PopBack (Q)	restituisce e cancella tail . prev <i>O</i> (1)
Empty (Q)	se <i>n</i> > 0, restituisce true se head . next = tail <i>O</i> (1)

Dizionari

I dizionari sono strutture dati che contengono elementi accessibili direttamente data la loro chiave. Supportano le operazioni: **Insert**, **Delete**, **Search** e sono implementati come vettori di puntatori dove le chiavi vengono usate come indici del vettore.

Insert (D , e)	D [e .key] ← e	Θ(1)
Delete (D , e)	D [e .key] ← NIL	Θ(1)
Search (D , e .key)	return D [e .key]	Θ(1)

La complessità spaziale è *O*(|**D**|) con **D** il dominio delle chiavi.

Tabelle di hash

Una tabella di hash è un caso speciale di dizionario con una complessità spaziale pari al numero di chiavi *m* per cui è effettivamente presente un valore. L'indice associato a una certa chiave è dato dal risultato di una funzione *h*(·) : **D** → {0, . . . , *m* − 1} detta funzione di hash.

Def. Il fattore di carico *α* è definito come

n

m

{\displaystyle {\frac {n}{m}}}

 dove *n* è il numero di elementi contenuti nella tabella e *m* è il numero di righe totali.

Collisioni

Date due chiavi *k*₁, *k*₂ con *k*₁ ≠ *k*₂ abbiamo una collisione se *h*(*k*₁) = *h*(*k*₂). Le collisioni si risolvono principalmente con due metodi.

Indirizzamento chiuso (closed addressing/open hashing/chaining)

Ogni riga della tabella (bucket) contiene la testa di una lista. Nel caso di collisione, l'elemento viene inserito in testa alla lista Θ(1). Nel caso peggiore tutti gli elementi collidono dando origine a una lista lunga *m* elementi, quindi **Insert/Delete/Search** in *O*(*m*).

Def. Ipotesi di Hashing Uniforme Semplice (IHUS) è un’opportuna funzione di hash dove ogni chiave ha probabilità

1
m

{\displaystyle {\frac {1}{m}}}

 di finire in una qualsiasi delle *m* celle.

In caso di IHUS, la lunghezza media di una listà è *α* e il tempo medio per cercare una chiave è Θ(1 + *α*). Se *α* non è eccessivo, tutte le operazioni sono *O*(1) in media. Per mantenere l'efficienza delle operazioni, si sceglie un *α* oltre il quale viene fatta una riallocazione in una tabella più grande (**rehashing**). (i solito *α*_{max} ≈

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.

0	1	2	3	4	5	6
---	---	---	---	---	---	---

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.

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

The left diagram shows a binary tree with root 4. Node 4 has a left child 2 and a right child 5. Node 2 has a left child 1 and a right child 3. Node 5 has a left child 6.

The right diagram shows a binary tree with root 4. Node 4 has a left child 2. Node 2 has a left child 1.

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

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

INSERT(T, x) : $O(h)$ 1 $pre \leftarrow NIL$ 2 $cur \leftarrow T.root$ 3 while $cur \neq NIL$ 4 $pre \leftarrow cur$ 5 if $x.key < cur.key$ 6 $cur \leftarrow cur.left$ 7 $cur \leftarrow cur.right$ 8 $x.p \leftarrow pre$ 9 if $pre = NIL$ 10 $T.root \leftarrow x$ 11 elseif $x.key < pre.key$ 12 $pre.left \leftarrow x$ 13 else $pre.right \leftarrow x$	DELETE(T, x) : $O(h)$ 1 if $x.left = NIL$ or $x.right = NIL$ 2 $del \leftarrow x$ 3 else $del \leftarrow SUCCESSOR(x)$ 4 if $del.left \neq NIL$ 5 $subtree \leftarrow del.left$ 6 else $subtree \leftarrow del.right$ 7 if $subtree \neq NIL$ 8 $subtree.p \leftarrow del.p$ 9 if $del.p = NIL$ 10 $T.root \leftarrow subtree$ 11 elseif $del = del.p.left$ 12 $del.p.left \leftarrow subtree$ 13 else $del.p.right \leftarrow subtree$ 14 if $del \neq x$ 15 else $x.key \leftarrow del.key$ 16 $FREE(del)$
---	--

La complessità di tutte le operazioni è $O(h)$ con h altezza dell'albero. Nel caso ottimo (albero bilanciato) è $O(\log(n))$, nel caso pessimo (albero degenerato in una lista) è $O(n)$.

Gli alberi rosso-neri sono BST i cui nodi hanno un colore $\in \{\text{rosso}, \text{nero}\}$ e che soddisfano le seguenti proprietà:

1. Ogni nodo è rosso o nero
2. La radice è nera
3. Le foglie sono nere
4. I figli di un nodo rosso sono entrambi neri
5. Per ogni nodo dell'albero, tutti i cammini dai suoi discendenti alle foglie contenute nei suoi sottoalberi hanno lo stesso numero di nodi neri

Le operazioni che non modificano la struttura dell'albero sono le stesse dei BST (SEARCH, MIN, MAX, SUCCESSOR). Le procedure di inserimento (INSERT) e cancellazione (DELETE) sono modificate per mantenere l'albero bilanciato assicurando una complessità temporale di $O(\log(n))$ per tutte le operazioni.

Gli heap binari sono alberi binari quasi-completi dove la chiave del nodo padre è sempre maggiore o minore di quella dei figli. Nel primo caso si parla di max-heap nel secondo di min-heap.

```

graph TD
    30((30)) --> 19((19))
    30 --> 20((20))
    19 --> 7((7))
    19 --> 11((11))
    20 --> 3((3))
  
```

MAXHEAPIFY(A, n) riceve un array e una posizione in esso: assume che i due sottoalberi con radice in $\text{LEFT}(n) = 2n$ e $\text{RIGHT}(n) = 2n + 1$ siano dei max-heap e modifica A in modo che l'albero radicato in n sia un max-heap.

```

MAXHEAPIFY( $A, n$ ) :  $O(\log(n))$ 
1   $l \leftarrow \text{LEFT}(n)$ 
2   $r \leftarrow \text{RIGHT}(n)$ 
3  if  $l \leq A.\text{heapsize}$  and  $A[l] > A[n]$ 
4       $\text{posmax} \leftarrow l$ 
5  else  $\text{posmax} \leftarrow n$ 
6  if  $r \leq A.\text{heapsize}$  and  $A[r] > A[\text{posmax}]$ 
7       $\text{posmax} \leftarrow r$ 
8  if  $\text{posmax} \neq n$ 
9      SWAP( $A[n], A[\text{posmax}]$ )
10  MAXHEAPIFY( $A, \text{posmax}$ )

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

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

```

EXTRACTMAX(A) : $O(\log(n))$	INSERT(A, key) : $O(\log(n))$
1 if $A.heapsize < 1$	1 $A.heapsize \leftarrow A.heapsize + 1$
2 return \perp	2 $A[A.heapsize] \leftarrow key$
3 $max \leftarrow A[i]$	3 $i \leftarrow A.heapsize$
4 $A[1] \leftarrow A[A.heapsize]$	4 while $i > 1$ and $A[PARENT(i)] < A[i]$
5 $A.heapsize \leftarrow A.heapsize - 1$	5 SWAP($A[PARENT(i)], A[i]$)
6 MAXHEAPIFY($A, 1$)	6 $i \leftarrow PARENT(i)$
7 return max	

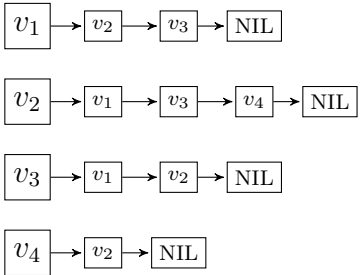
Un grafo è una coppia $\mathcal{G} = (\mathbf{V}, \mathbf{E})$ con \mathbf{V} un insieme di nodi (detti anche vertici) ed \mathbf{E} un insieme di archi (detti anche lati).

- Un grafo con $|V|$ nodi ha al più $|V|^2$ 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 *aciclico*
- 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

$\mathbf{V} = \{v_1, v_2, v_3, v_4\}$
 $\mathbf{E} = \{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_2, v_4), (v_3, v_4)\}$

I grafi vengono rappresentati in memoria con *liste di adiacenza* o *matrice 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 $|\mathbf{V}| \times |\mathbf{V}|$ di valori booleani con righe e colonne indicizzate 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.

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

	Liste	Matrice
Comlessità 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	$\Theta(o_e)$	$O(\mathbf{V})$

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

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.

```

1  for each  $n \in \mathbf{V} \setminus \{s\}$ 
2     $n.color \leftarrow white$ 
3     $n.dist \leftarrow \infty$ 
4   $s.color \leftarrow grey$ 
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  $v \in cur.adacenti$ 
11     if  $v.color = white$ 
12        $v.color \leftarrow grey$ 
13        $v.dist \leftarrow cur.dist + 1$ 
14       ENQUEUE( $Q, v$ )
15    $cur.color \leftarrow black$ 

```

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 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.

```

1  for each  $n \in V \setminus \{s\}$ 
2       $n.color \leftarrow white$ 
3       $n.dist \leftarrow \infty$ 
4   $s.color \leftarrow grey$ 
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  $v \in cur.adacenti$ 
11         if  $v.color = white$ 
12              $v.color \leftarrow grey$ 
13              $v.dist \leftarrow cur.dist + 1$ 
14             PUSH( $S, v$ )
15      $cur.color \leftarrow black$ 

```

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

```

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

```

VISITAEDETICHETTA funziona come VISITAAMPIEZZA o VISITAPROFONDITÀ m
imposta a *eti* il campo *etichetta* del nodo visitato.

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

