Algoritmi e principi dell'informatica

Raffaele Daniele Facendola

January 23, 2012

Contents

Part I
Teoria della computazione

La TdC ci permette di capire quali problemi possiamo risolvere, ed eventualmente dirci con quale macchina.

In informatica tutti i problemi informatici possono essere ricondotti ai seguenti due:

- $x \in L$? ovvero, la stringa x appartiene al linguaggio L?
- $y = \tau(x)$ ovvero, y è il risultato di una determinata funzione (algoritmo) τ con argomento x (parametro di input)?

In realtà i due problemi possono essere ricondotti l'uno all'altro e viceversa:

- Se ho una macchina che calcola la funzione $y = \tau(x)$ mi basta definire $\tau(x) = 1$ se $x \in L$, $\tau(x) = 0$ se $x \notin L$. In questo modo ho ricondotto il primo problema al secondo.
- Se ho una macchina che ci dice se una determinata stringa x appartiene ad un linguaggio L definiamo L come $L = \{x\$y : y = \tau(x)\}$ (con \$ carattere speciale non usato nè per x nè per y). Il linguaggio così ottenuto è formato da tutte le parole formate dall'argomento x unite al valore restituito da $\tau(x)$ mediante \$. Dopodichè, fissato un x, enumero tutte le possibili stringhe y e per ciascuna di esse chiedo alla macchina se $x\$y \in L$. Se la funzione è definita prima o poi la macchina termina restituendo l'esito.

1 Potenza dei modelli di calcolo

2 Tesi di Church

La tesi di Church asserisce che non esiste meccanismo di calcolo automatico superiore alla macchina di Turing.

In poche parole non esiste nessun algoritmo che può risolvere problemi che la macchina di Turing non sia in grado di risolvere.

Dalla precedente possiamo concludere quindi che tutti i problemi risolvibili sono tutti e soli quelli risolvibili dalla MT (è possibile cioè trovare un algoritmo in grado di portare in tempo finito ad una soluzione).

3 Enumerazione delle macchine di Turing

Tutte le macchine di Turing sono enumerabili: è possibile, cioè, associare a ciascuna macchina di Turing un indice sull'insieme dei naturali (detto numero di Goedel) che la descrive in maniera univoca.

4 Macchine di Turing universali

Sia y il numero di Goedel relativo alla macchina di Turing y-esima, definiamo $f_y : \mathbb{N} \to \mathbb{N}$ la funzione (problema) calcolata dalla y-esima macchina di Turing. Per convenzione $f_y(x) = \perp$ se la MT y-esima non si ferma mai quando in ingresso le viene fornito x.

Una macchina di Turing universale è una MT che calcola la funzione $g(y, x) = f_y(x)$. Essa è in grado di simulare il comportamento di una qualsiasi macchina di Turing (definita mediante il parametro y) quando in ingresso le viene fornito il parametro (x).

5 Problemi computabili

Giacchè le macchine di Turing possono essere enumerate e che tutti i problemi computabili sono tutti e soli quelli risolvibili dalle MT, possiamo concludere che l'insieme dei problemi computabili ha la stessa cardinalità dei numeri naturali, ovvero $\aleph_0$.

Dal teorema di Cantor si sa che la cardinalità dell'insieme di tutte le funzioni $\mathcal{F} = \{f | f : \mathbb{N} \to \mathbb{N}\}$ è $2^{\aleph_0}$, quindi possiamo concludere che l'insieme dei problemi computabili ha cardinalità inferiore a tutti i problemi "esistenti" e quindi che che gran parte dei problemi non è risolvibile algoritmicamente.

Due problemi che non è possibile risolvere sono i seguenti:

Halt della MT: è possibile costruire una macchina che dato in ingresso un programma e relativi parametri di input ci sappia dire in anticipo se il programma non termina? In formule vogliamo una MT che calcoli $g(x, y)$: $\begin{matrix} 1\, se\, f_y(x) \neq \perp \\ 0\, se\, f_y(x) = \perp \end{matrix}$. La risposta al problema è **NO!**

Supponiamo per assurdo che $g(x, y)$ sia computabile. Definiamo quindi $h(x)$: $\begin{matrix} 1\, se\, g(x,x) = 0 \\ \perp\, se\, g(x,x) = 1 \end{matrix}$. Se h è computabile allora esiste una macchina di Turing (diciamo la x_0-esima) in grado di calcolarla.

Supponiamo che $h(x_0) = 1$, ciò significa che $g(x_0, x_0) = 0$ ovvero che $f_{x_0}(x_0) = \perp$. Ma questo non è possibile perchè $h(x_0) = f_{x0}(x_0)$ e $1 \neq \perp$.

Supponiamo allora che $h(x_0) = \perp$, cioè significa che $g(x_0, x_0) = 1$ ovvero che $f_{x0}(x_0) \neq \perp$. Ma questo non è possibile per lo stesso motivo di sopra.

Usando quindi la dimostrazione per assurdo concludiamo che $g(x, y)$ non è computabile.

Dimostrare che una funzione è totale: E' impossibile costruire una macchina di Turing in grado di dire se una funzione y è definita (è cioè $\neq \perp$) per ogni x su $\mathbb{N}$. In formule $k(y) = \begin{matrix} 1\, se\, f_y(x) \neq \perp \forall x \in \mathbb{N} \\ 0\, altrimenti \end{matrix}$ è non computabile.

In realtà questo problema è una versione ancor più generale del precedente, in quanto ci chiediamo se dato un programma esso terminerà sempre a prescindere dagli input (nel problema dell'halt invece si chiede se un dato programma con un dato input termina o meno).

6 Problemi indecidibili

Sia dato un insieme S, definiamo la sua funzione caratteristica $c_S(x)$: $\begin{matrix} 1\, se\, x \in S \\ 0\, se\, x \notin S \end{matrix}$.

Un insieme S è **decidibile** o **ricorsivo** se e solo se la sua funzione caratteristica è computabile (e totale).

Un insieme S è **semidecidibile** o **ricorsivamente enumerabile** (RE) se e solo se $S = \varnothing$ o $S = \{x | x = g_S(y), y \in \mathbb{N}\}$. Con g_S funzione totale e computabile. In generale per un insieme RE posso dire se $x \in S$ in quanto enumerando tutti gli elementi di S prima o poi lo trovo e posso dire di "sì", però se $x \notin S$ non posso dire nulla se S è infinito (in sostanza se dopo aver enumerato i primi k elementi non riesco a trovare x non posso dire nulla perchè il k+1-esimo elemento potrebbe essere quello che cerco).

In generale valgono le seguenti:

S è ricorsivo allora S è anche ricorsivamente enumerabile

S è ricorsivo se e solo se o S e il suo complemento sono ricorsivamente enumerabili.

7 Teorema di Rice

Sia F un insieme di funzioni computabili. Definiamo S come segue: $S = \{x | f_x \in F\}$.

S è decidibile se e solo se $F = \varnothing$ o F è l'insieme di tutte le funzioni computabili.

Come conseguenza del teorema di Rice si ha che S è non decidibile per tutti i casi non banali, quindi i seguenti sono indecidibili:

- Un programma computa una determinata funzione?
- Due programmi computano la medesima funzione?
- La funzione computata da un programma gode di una certa proprietà (è sempre pari, ha un codominio limitato, ecc.)?

8 Tecniche di dimostrazione di indecidibilità

Se siamo in grado di trovare un algoritmo che termina sempre il problema è **decidibile**.

Se troviamo un algoritmo che potrebbe non terminare ma se lo fa restituisce "Sì" come risposta in problema è **semidecidibile**.

Dimostrare che un problema è **indedicibile** è leggermente più complesso:

In primo luogo potremmo ricorrere al teorema di Rice, in alternativa le opzioni sono tre:

- **Riduzione dei problemi**: ci si riconduce a problemi noti non decidibili (halt della MT, funzione totale, ecc.)

- **Enumerazione diagonale**: si prova ad eseguire la soluzione su più input contemporaneamente (si esegue il primo ed il secondo step del primo input, quindi si esegue il primo step del secondo input, il terzo step del primo, il secondo step del secondo ed il primo del terzo e così via su tutti gli input in parallelo: se qualche input porta il programma in una condizione di stallo gli altri sono ancora in grado di andare avanti e ciò fornisce un valido metodo per la dimostrazione della semidecidibilità).

- **Dimostrazione mediante reductio ad absurdum:** si suppone per assurdo che un problema sia decidibile: se si giunge a delle contraddizioni allora la tesi è sbagliata ed il problema non è decidibile (potrebbe ancora essere semidecidibile però!)

Dai capitoli precedenti possiamo concludere che:

Se un problema è **decidibile** il suo complemento è **decidibile**.

Se un problema è **semidecidibile** il suo complemento è **indecidibile**.

Se un problema è **indecidibile** il suo complemento è o **semidecidibile o indecidibile**.

Se un problema è **decidibile**, una sua specializzazione è **decidibile**.

Se un problema è **indecidibile**, una sua generalizzazione è **indecidibile**.

9 Appartenenza di stringhe a determinati linguaggi

Il problema generico che ci apprestiamo a risolvere è il seguente $S \in L$? Ovvero la stringa S appartiene al linguaggio L?

- Se L è un linguaggio ricorsivamente enumerabile (riduzioni del tipo $\alpha \to \beta$, riconosciuto dalle MT) allora S è **semidecidibile.**

- Se L è un linguaggio ricorsivo o linear bound (produzioni del tipo $|\alpha| \leq |\beta|$, riconosciuto da una MT che termina sempre) allora S è **decidibile**.

- Se L è un linguaggio context-free (produzioni del tipo $|\alpha| = 1$, riconosciuto mediante NPDA) allora S è **decidibile**.

- Se L è un linguaggio regolare (produzioni del tipo $\alpha \to a$, $\alpha \to aB$ o $\alpha \to a$, $\alpha \to Ba$, riconosciuto tramite FSA) allora S è **decidibile**.

In realtà ci basta dimostrare che S è decidibile per L ricorsivo in quanto sappiamo che $regular \subset context-free \subset linear-bound \subset RE$. Da ciò concludiamo che se S è decidibile per linguaggi linear-bound allora lo è anche per L context free e L regolare (questo perchè gli ultimi due sono una specializzazione del primo).

La dimostrazione è semplice: uso tutte le produzioni per generare tutte le parole di L lunghe quanto la stringa |S| (inizio cioè a generare parole a caso e mi fermo quando supero la lunghezza di |S|). Scarto tutte le parole che contengono dei simboli non terminali: le rimanenti sono tutte le parole lunghe |S| generate da L. Siccome l'insieme di parole rimasto è finito il problema è decidibile (basta confontare le parole una alla volta con S, tanto prima o poi finisco).

Un altro problema generale:

- Dati due linguaggi L_1 e L_2 voglio sapere se $L_1 \subseteq L_2$: il problema è **indecidibile** per le grammatiche context-free (e quindi anche per le linear bounded e le RE), risulta **decidibile** per le regolari.

Part II
Teoria della complessità

In questa parte ci occuperemo di valutare l'efficienza dei vari programmi (ovvero quanto tempo impiegano per svolgere un determinato compito e quanta memoria consumano nel farlo). In generale concentreremo la nostra analisi nella valutazione del caso pessimo (il più rilevante).

10 Nozioni e notazioni fondamentali per l'analisi di complessità

Sia "n" la mole di elementi passati in input ad un determinato programma. In generale l'efficienza temporale di un programma sarà della forma $T(n)$. Questo valore dipende dal numero (a volte anche dal tipo) di istruzioni eseguite dalla macchina associando a ciascuna di esse un "costo" (un valore simbolico per indicare quanto "veloce" o "efficiente" è la sua esecuzione), tuttavia non sempre si è in grado di stabilire con precisione la **complessità** di un algoritmo (in realtà a volte non ci interessa nemmeno) quindi procederemo con un analisi "asintotica", ovvero ne valuteremo l'efficienza per valori di n molto grandi.

- **Notazione O-grande**: indica un limite asintotico superiore ed è definita come $f(n) = O(g(n))$ se e solo se $f(n) < cg(n)$ per qualche c ed n sufficientemente elevato.
- **Notazione Ω-grande**: indica un limite asintotico inferiore ed è definita come $f(n) = \Omega(g(n))$ se e solo se $f(n) > cg(n)$ per qualche c ed n sufficientemente elevato.
- **Notazione Θ-grande**: indica un limite asintotico inferiore e superiore ed è definita come $f(n) = \Theta(g(n))$ se e solo se $c_1 g(n) < f(n) < c_2 g(n)$ con $c_1 < c_2$ e per n sufficientemente elevato. Vale anche la seguente $f(n) = O(g(n)) \wedge f(n) = \Omega(g(n)) \Leftrightarrow \Theta(g(n))$.

Esempi:
$3n^2 + 12n + 35 = O(n^2) = \Omega(n^2) = \Theta(n^2) = O(n^{1000})$
$2nlog(n) = O(nlogn) = \Omega(n) = \Theta(nlog(n)) = \Omega(nlogn) = O(1)$
Volendo calcolare la complessità temporale di un algoritmo si ha che:

- Le istruzioni di base (somma, sottrazione, moltiplicazione, assegnazione) sono O(1)
- Blocchi di codice hanno complessità che è pari alla somma della complessità dei singoli blocchi\istruzioni
- Tutti i cicli hanno una complessità pari al prodotto della complessità del corpo del ciclo per il numero di volte che esso viene eseguito (per indici

incrementali da 0 ad q la complessità è O(q), se l'indice viene moltiplicato o diviso per un numero k ad ogni ciclo allora la complessità è $log_k(q)$).

Per valutare la complessità spaziale bisogna sommare alla memoria necessaria per memorizzare l'input anche quella allocata dinamicamente durante l'esecuzione del programma (se viene allocato un array di q elementi la complessità spaziale aumenta di q). Se vi è allocazione dinamica all'interno dei cicli si valuta la complessità del ciclo come sopra e la si moltiplica per la memoria allocata (all'interno di esso) ogni volta.

11 Master Theorem

Il master theorem consente di valutare al volo la complessità temporale di algoritmi ricorsivi (entro certi limiti). Considereremo soltanto algoritmi la cui ricorsione è della forma $T(n) = aT(\frac{n}{b}) + f(n)$ dove il primo termine indica che ad ogni ricorsione il problema viene suddiviso in "a" problemi, ciascuno dei quali lavora su un problema grande $\frac{n}{b}$ volte quello della ricorsione precedente ed ha una complessità algoritmica pari ad f(n).

Caso 1: $f(n) = O(n^{log_b a+\epsilon})$ per $\epsilon > 0$. In questo caso la complessità dell'algoritmo è dominata dalla complessità della ricorsione e quindi $T(n) = \Theta(n^{log_b a})$.

Caso 2: $f(n) = \Theta(n^{log_b a})$. In questo caso sia la ricorsione che la complessità dell'algoritmo concorrono a quella totale e quindi $T(n) = \Theta(n^{log_b a} log(n))$.

Caso 3: $f(n) = \Omega(n^{log_b a+\epsilon})$ per $\epsilon > 0$ e $\exists c \in \mathbb{R}(c < 1 \wedge a \cdot f(\frac{n}{b}) \leq c \cdot f(n))$. In questo caso la complessità intrinseca domina quella della ricorsione e quindi $T(n) = \Theta(f(n))$.

Il master theorem è valido solo se la complessità della ricorsione $n^{log_b a}$ è **polinomialmente** comparabile ad f(n). Il teorema non funziona se $T(n) = 4T(\frac{n}{4}) + n \cdot log(n)$ perché $n^{log_4 4}$ è più piccolo di $n \cdot log(n)$ ma non polinomialmente più piccolo.

12 Metodo di sostituzione

In alternativa al Master theorem è possibile sviluppare la ricorsione $T(n) = aT(\frac{n}{b}) + f(n) = a(a(\frac{n}{b^2}) + f(\frac{n}{b})) + f(n) = ...$ e capire a che valore sta convergendo la serie. A questo punto bisogna valutare se la funzione individuata è effettivamente quella corretta usando le definizioni di notazioni di complessità.

13 I modelli di calcolo e le relazioni tra le loro complessità computazionali

13.1 Complessità di automi a stati finiti, a pila e macchina di Turing a nastro singolo

E' possibile dimostrare che:

- Gli automi a stati finiti hanno sempre complessità spaziale $S(n) = \Theta(k)$ (costante) e complessità temporale $T(n) = \Theta(n)$ (in realtà è esattamente uguale ad n, e vengono dette macchine real time).
- Gli automi a pila hanno sempre $S(n) \leq \Theta(n)$ e $T(n) = \Theta(n)$
- Le macchine di Turing a nastro singolo hanno sempre $S(n) > \Theta(n)$ (questo perchè la memoria è condivisa e ovviamente ci sono almeno gli n caratteri di input su di esso). La complessità temporale invece va valutata caso per caso (ad esempio per riconoscere wcw^R la $T(n) \geq \Theta(n^2)$). Possiamo quindi concludere che le MT a nastro singolo sono più potenti delle PDA ma talvolta sono meno efficienti.

13.2 Accelerazione lineare

- Se una macchina di Turing a k nastri può riconoscere una stringa di un linguaggio L con complessità spaziale $S(n)$ è sempre possibile costruire una macchina di Turing (sempre a k nastri) con complessità spaziale inferiore (di un fattore moltiplicativo c > 0) tale che $S'(n) < c \cdot S(n)$. Per fare ciò è "sufficiente" simulare in un passo della seconda macchina più passi (precisamente c passi) della prima "comprimendo" l'input. Supponendo che la prima MT abbia un alfabeto di ingresso $A = \{a, b, c\}$ è possibile stabilire un nuovo alfabeto A' tale per cui $|A'| = |A|^c$ e associare a ciascun simbolo in A' una combinazione di c simboli di A: se c $= 2$ allora avremmo $A' = \{a = aa, b = ab, c = ac, d = ba, e = bb, f = bc, g = ca, h = cb, i = cc\}$. In questo caso dopo la codifica dell'input, la lettura di un carattere simulerà 2 passi della MT precedente.
- Se L è accettato da una MT a k nastri con complessità spaziale $S(n)$ si può sempre costruire una MT ad un nastro (non a nastro singolo!) con complessità spaziale uguale: basta simulare sul nastro singolo il comportamento degli altri k mettendo le parti significative dei k nastri una dietro l'altra.
- Dal primo teorema e dal secondo si ha che se una MT a k nastri riconosce L con complessità spaziale $S(n)$ allora è sempre possibile costruire una MT ad 1 nastro con complessità spaziale inferiore di un fattore c>0 $S'(n) < c \cdot S(n)$.

- Se L è accettato da una MT a k nastri con complessità temporale $T(n)$ è sempre possibile costruire una MT a k+1 nastri che lo riconosce con complessità temporale $T'(n) = max(n+1, cT(n))$ per ogni c>0 e$c \in \mathbb{R}$.

Come conseguenza dei teoremi precedenti si ha che è sempre possibile migliorare la velocità di esecuzione di un algoritmo (a patto di utilizzare più memoria), tuttavia i miglioramenti sono al più lineari (non è possibile ridurre la classe di complessità di un algoritmo "parallelizzando" il calcolo).

14 La macchina RAM

La macchina RAM è un modello astratto del calcolatore reale. Esso è costituito da un organo di controllo che esegue un programma, un nastro di input (che può essere letto **solo** sequenzialmente e solo dall'inizio alla fine), uno di output ed una memoria infinita ad accesso casuale.

La RAM è in grado di leggere algoritmi in codice simil-assembly le cui istruzioni sono sono:

ISTRUZIONE	**SEMANTICA**
$LOAD = x$	$M[0] = x$
$LOAD\, x$	$M[0] = M[x]$
$LOAD * x$	$M[0] = M[M[x]]$
$STORE\, x$	$M[x] = M[0]$
$STORE * x$	$M[M[x]] = M[0]$
$ADD = x$	$M[0] = M[0] + x$
$ADD\, x$	$M[0] = M[0] + M[x]$
$ADD * x$	$M[0] = M[0] + M[M[x]]$
$SUB = x$	$M[0] = M[0] - x$
$SUB\, x$	$M[0] = M[0] - M[x]$
$SUB * x$	$M[0] = M[0] - M[M[x]]$
$MULT = x$	$M[0] = M[0] * x$
$MULT\, x$	$M[0] = M[0] * M[x]$
$MULT * x$	$M[0] = M[0] * M[M[x]]$
$DIV = x$	$M[0] = M[0]\, div\, x$
$DIV\, x$	$M[0] = M[0]\, div\, M[x]$
$DIV * x$	$M[0] = M[0]\, div\, M[M[x]]$
$READ\, x$	$M[x] = valore\, in\, input$
$READ * x$	$M[M[x]] = valore\, in\, input$
$WRITE = x$	$stampa\, x\, in\, output$
$WRITE\, x$	$stampa\, M[x]\, in\, output$
$WRITE * x$	$stampa\, M[M[x]]\, in\, output$
$JUMP\, etich$	$salta\, all'istruzione$ "$etich$"
$JGZ\, etich$	$salta\, all'istruzione$ "$etich$" $se\, M[0] > 0$
$JZ\, etich$	$salta\, all'istruzione$ "$etich$" $se\, M[0] = 0$
$HALT$	$fine\, della\, computazione$

Ogni cella dei nastri di input\output o della memoria è in grado di memorizzare un intero (di lunghezza arbitraria)

14.1 Valutazione di complessità con criterio del costo costante

Il criterio del costo costante (o uniforme) ci fornisce una stima dell'efficienza del codice eseguito dalla macchina RAM. In generale ogni istruzione elementare ha complessità temporale costante (è $O(1)$). Ogni cella di memoria occupata M[x] occupa un'unità di memoria e quindi $S(n) = O(1)$.

- Tutti i cicli hanno una complessità pari al prodotto della complessità del corpo del ciclo per il numero di volte che esso viene eseguito (per indici incrementali da 0 ad q la complessità è O(q), se l'indice viene moltiplicato o diviso per un numero k ad ogni ciclo allora la complessità è $log_k(q)$). Questo vale sia per la complessità spaziale che temporale.

14.2 Il criterio logaritmico

Il criterio logaritmico viene introdotto per sopperire alle non idealità della macchina RAM: usando questo tipo di approccio si ha che il costo delle operazioni elementari è proporzionale alla grandezza degli operandi (codificati in binario), come se la macchina avesse a disposizione un bus ad 1 solo bit.

Con questo criterio tutte le istruzioni a parte JUMP ed HALT hanno complessità temporale $O(log_2(n))$ mentre le due citate sono O(1).

Per valutare il costo spaziale invece si può supporre che le celle di memoria vengano ridimensionate dinamicamente e quindi il costo totale è $\sum log_2(M_i)$ con i insieme di tutte le celle di memoria consumate.

Per i cicli bisogna considerare che se gli indici vengono incrementati di valori costanti allora la complessità del ciclo è $O(nlog_2(n))$ (ovvero n il numero di "giri" e log(n) per tener conto dell'addizione), mentre se l'indice viene moltiplicato o diviso per k allora la complessità diventa $O(log_k n \cdot log(n))$. Visto che per cambiare la base di un logaritmo è sufficiente dividere il suo valore per una costante che dipende dalle due basi (proprietà dei logaritmi) fondamentalmente la complessità diventa $O((log(n))^2)$.

15 Il teorema di correlazione polinomiale

Applicando ragionevoli criteri di costo (in generale per la macchina RAM quello logaritmico è verosimile, quello costante no) se un modello di calcolo risolve un problema con complessità (spazio\temporale) $C_1(n)$ allora esso è risolvibile da qualsiasi altro modello di calcolo con complessità $C_2(n) \leq P(C_1(n))$ con P un opportuno polinomio. In sostanza non è possibile cioè che la soluzione ottima di un problema sia $O(n)$ su una macchina e $O(2^n)$ su un'altra; al massimo è ammissibile che sia $O(n^2)$ (ma anche $O(n^{1000})$). E' possibile dimostrare che una MT è in grado di simulare una RAM con costo temporale al più di$\Theta(T^2)$ dove T è la complessità temporale di quest'ultima.

16 Gerarchie di complessità

Dal risultato del teorema di cui sopra possiamo dividere i problemi in grandi categorie (**classi di complessità**), ad esempio possiamo chiamare **P** la classe di problemi risolvibili in tempo polinomiale (non importa che siano $O(n)$ o $O(n^{1000})$ l'importante è che non siano del tipo $O(2^n)$ o, peggio ancora, $O(n^n)$).

P in generale è considerata come la classe dei problemi "trattabili" nella pratica anche se in realtà è richiesto che l'esponente dominante del polinomio non sia incredibilmente alto.

Part III
Strutture dati a algoritmi fondamentali

17 Algoritmi di ricerca ed ordinamento

17.1 Insertion sort

L'idea alla base dell'algoritmo è la stessa applicata per l'ordinamento dei libri su uno scaffale: se si vuole inserire un elemento basta scorrere da sinistra verso destra tutti gli elementi fino ad individuare la posizione corretta. A questo punto tutti gli elementi alla destra vengono spostati di una posizione più in la e il nuovo elemento (key) viene inserito nello spazio appena creato.

Lo pseudocodice è il seguente (A è il vettore da ordinare):

INSERTION-SORT(A)	*costo*	*numero di volte*
1 **for** *j* := 2 **to** *A.length*	c_1	n
2 *key* := *A[j]*	c_2	n - 1
3 //Inserisce *A[j]* nella sequenza *A[1..j-1]*	0	n - 1
4 *i* := *j* - 1	c_4	n - 1
5 **while** *i* > 0 and *A[i]* > *key*	c_5	$\sum_{j=2}^{n} t_j$
6 *A[i + 1]* := *A[i]*	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 *i* := *i* - 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 *A[i + 1]* := *key*	c_8	n - 1

Nel caso pessimo (array ordinato in ordine decrescente) la complessità temporale di insertion-sort è $\Theta(n^2)$.

Nel caso ottimo (array ordinato) la complessità temporale è $\Theta(n)$.

La complessità spaziale è $\Theta(n)$ (nessuna memoria aggiuntiva è richiesta).

17.2 Merge sort

L'algoritmo è un classico divide et impera: se l'array ha meno di due elementi è ordinato, altrimenti si divide l'array in due sottoarray e si riapplica l'algoritmo. A questo punto si fondono i due sottoarray preservando l'ordine tra gli elementi dei due e si risale lungo l'albero delle ricorsioni.

```
MERGE-SORT(A, p, r)
1  if p < r
2    q := ⌊(p + r)/2⌋
3    MERGE-SORT(A, p, q)
4    MERGE-SORT(A, q+1, r)
5    MERGE(A, p, q, r)
```

```
MERGE (A, p, q, r)
1  n_1 := q − p + 1
2  n_2 := r − q
3  crea (alloca) 2 nuovi array L[1..n_1+1] e R[1..n_2+1]
4  for i := 1 to n_1
5    L[i] := A[p + i - 1]
6  for j := 1 to n_2
7    R[j] := A[q + j]
8  L[n_1 + 1] := ∞
9  R[n_2 + 1] := ∞
10 i := 1
11 j := 1
12 for k := p to r
13   if L[i] ≤ R[j]
14     A[k] := L[i]
15     i := i + 1
16   else A[k] := R[j]
17     j := j + 1
```

La complessità della routine Merge è $\Theta(n)$, mentre la routine Merge-Sort crea due sottoproblemi di dimensioni dimezzate ogni volta. La complessità dell'algoritmo è $T(n) = 2T(n/2) + \Theta(n)$ se n>=2 o $\Theta(1)$ se n<2. Usando il master theorem concludiamo che la complessità temporale totale è $\Theta(nlog(n))$. Rispetto alla insertion sort merge sort crea due sottoarray di lunghezza $\Theta(n)$ ad ogni sottoproblema quindi la complessità spaziale totale è $\Theta(nlog(n))$.

17.3 Heapsort

L'heapsort è efficiente quanto mergesort, tuttavia è un algoritmo che ha complessità spaziale $\Theta(n)$ non richiedendo memoria aggiuntiva.

L'idea di base è che ogni array può essere visto come un albero binario in cui per ogni nodo di indice "i" il figlio sinistro è l'elemento 2i-esimo, mentre quello destro è l'elemento 2i+1-esimo. Il padre è l'elemento i/2-esimo.

Un heap binario è un albero binario quasi completo, ovvero che ha tutti i livelli occupati tranne al più l'ultimo che può essere anche incompleto fino ad

un certo punto (partendo da sinistra).

Un heap è max-heap se per ogni nodo x dell'albero il valore del nodo padre è sempre maggiore o uguale a quello memorizzato in x. Dalla proprietà seguente segue che in un max-heap il nodo più grande è la radice dell'albero.

L'algoritmo seguente prende un heap in cui i figli della radice sono max heap e se la radice risulta più piccola dei propri figli, allora risolve il problema (su tutto l'albero ricorsivamente):

```
MAX-HEAPIFY(A, i)
1  l := LEFT(i)
2  r := RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4    max := l
5  else max := i
6  if r ≤ A.heap-size and A[r] > A[max]
7    max := r
8  if max ≠ i then
9    swap A[i] ↔ A[max]
10   MAX-HEAPIFY(A, max)
```

La complessità temporale di maxheapify è proporzionale all'altezza dell'albero $T(n) = O(log(n))$.

A questo punto dato un array da ordinare lo trasformiamo in un max-heap mediante la seguente (si parte dalla metà dell'array perchè la parte destra contiene tutti i nodi foglia che sono già max-heap per definizione):

```
BUILD-MAX-HEAP(A)
1 A.heap-size := A.length
2 for i := A.length/2 downto 1
3   MAX-HEAPIFY(A, i)
```

La complessità temporale della precedente è $T(n) = O(nlog(n))$ ma da una più profonda analisi otteniamo in realtà che la precedente è in realtà $O(n)$.

A questo punto possiamo ordinare l'array mediante la seguente

```
HEAPSORT(A)
1 BUILD-MAX-HEAP(A)
2 for i := A.length downto 2
3   swap A[1] ↔ A[i]
4   A.heap-size := A.heap-size - 1
5   MAX-HEAPIFY(A,1)
```

L'idea alla base è quella di richiamare max heapify su tutto l'array, quindi giacchè sappiamo che la radice è l'elemento più grande lo mettiamo alla fine dell'array e ripetiamo l'algoritmo su n-1 elementi (tanto l'ultimo è già ordinato). La complessità temporale è pertanto data da $O(n)$ cicli moltiplicati per $O(log(n))$ complessità di max-heapify. La complessità temporale totale è quindi $O(nlog(n))$ (scartiamo il contributo additivo della funzione build max-heap).

17.4 Quicksort

Il seguente è un altro esempio di algoritmo divide et impera che ordina senza aver bisogno di memoria aggiuntiva.

L'idea alla base è quella di organizzare l'array in modo che tutti gli elementi in posizione $<$ i (A[i] è detto pivot) siano inferiori ad A[i], mentre quelli in posizione $\geq$i siano maggiori di A[i].

A questo punto dopo la riorganizzazione si richiama l'algoritmo sui sottovettori [1,i-1] e [i+1,n] e così via (in ogni passo l'elemento A[i] risulta sempre nella posizione corretta).

```
QUICKSORT(A, p, r)
1 if p < r
2   q := PARTITION(A, p, r)
3   QUICKSORT(A, p, q-1)
4   QUICKSORT(A, q+1, r)
PARTITION(A, p, r)
1 x := A[r]
2 i := p - 1
3 for j := p to r - 1
4   if A[j] ≤ x
5     i := i + 1
6     swap A[i] ↔ A[j]
7 swap A[i+1] ↔ A[r]
8 return i + 1
```

La complessità di Partition è $T(n) = \Theta(n)$, quella dell'intero algoritmo (nel caso medio) è $\Theta(nlog(n))$.

Il caso pessimo si ha se si sceglie un pivot tale per cui un sottoarray sia vuoto e l'altro sia di lunghezza n-1. In tal caso la complessità temporale diventa $T(n) = \Theta(n^2)$.

17.5 Counting sort

Il counting sort è un algoritmo di ordinamento non basato sui confronti particolarmente efficiente se i valori di input appartengono ad un dominio finito noto.

L'idea di base è che se nell'array ci sono m valori più piccoli di un certo elemento, allora quest'ultimo occuperà la posizione m+1.

```
COUNTING-SORT (A, B, k)
1  for i := 0 to k
2    C[i] := 0
3  for j := 1 to A.length
4    C[A[j]] := C[A[j]] + 1
5  //C[i] ora contiene il numero di elementi uguali a i
6  for i := 1 to k
7    C[i] := C[i] + C[i - 1]
8  //C[i] ora contiene il numero di elementi ≤ i
9  for j := A.length downto 1
10   B[C[A[j]]] := A[j]
11   C[A[j]] := C[A[j]] - 1
```

L'array C è un array di supporto tale per cui l'elemento i-esimo indica quanti elementi ci sono prima dell'elemento di valore i all'interno dell'array.

La complessità temporale dell'algoritmo è $T(n) = O(n+k) = O(n)$ dove n è il numero di elementi da ordinare e k è la grandezza del loro dominio. In realtà l'algoritmo è parecchio poco efficiente se k non è $O(n)$.

18 Strutture dati elementari: rappresentazione, algoritmi di ricerca e gestione

Le strutture dati non sono altro che oggetti usati per contenerne altri di varia natura. A questi elementi generalmente viene associata una chiave (possibilmente univoca) e di solito hanno dei dati satelliti (accessibili attraverso puntatori).

Queste strutture dati ammettono fondamentalmente due tipi di operazioni: quelle che modificano la collezione e quelle che la interrogano.

Tra le strutture più famose annoveriamo le **pile**, le **code**, le **liste concatenate**, i **dizionari** e le **hashtable**.

I dizionari sono delle strutture dati che ammettono le sole operazioni di INSERT, SEARCH e DELETE: non ammettono chiavi duplicate e non esiste un ordine tra i vari elementi.

19 Tabelle hash: funzioni di hash, indirizzamento aperto

Una tabella di hash è un caso speciale di dizionario la quale usa una memoria proporzionale (T) al numero di chiavi effettivamente memorizzate al suo interno. Il vantaggio rispetto alle altre strutture dati è che in generale tutte le operazioni sono $O(1)$.

Una **funzione di hash h(k)** è una funzione che restituisce un indirizzo all'interno della tabella T sfruttando una chiave in ingresso k.

In generale se si vogliono memorizzare tante chiavi ci saranno valori k_1e k_2 con $k_1 \neq k_2$ tali per cui $h(k_1) = h(k_2)$. In sostanza in certi casi la funzione di hash restituisce lo stesso indirizzo per chiavi differenti: questa eventualità è chiamata **collisione** ed è un evento scontato quando il numero di chiavi inizia ad essere molto grande (è statisticamente certo quando tutta la hashtable è saturata).

Per risolvere questi problemi si adottano diverse strategie:

- **Chaining**: in questo caso ogni elemento della hashtable è una lista concatenata: gli elementi vengono semplicemente aggiunti alla lista ogni volta che ne viene richiesto l'inserimento risolvendo difatti il problema della collisione. Definiamo **fattore di carico** $\alpha = \frac{n}{m}$ dove n è il numero di elementi memorizzati ed m il numero di slot disponibili nella hashtable. Supponendo che la probabilità di collidere sia $\frac{1}{m}$ allora la lunghezza media di una lista è pari al numero di inserimenti n moltiplicata per la problabilità di collisione: $L = n \cdot \frac{1}{m} = \alpha$. Quindi il tempo medio per cercare una chiave k nella hastable con chaining è $\Theta(1 + \alpha)$.

- **Indirizzamento aperto:** in questo caso non si usa memoria aggiuntiva: se una chiave collide si cerca uno slot vuoto disponibile mediante diverse tecniche di **ispezione**:

  - **Ispezione lineare**: si individua il primo slot libero dopo quello individuato dalla funzione di hash. Soffre di problemi di addensamento in quanto vi sono molte celle consecutive occupate che peggiorano le prestazioni in fase di ricerca.
  - **Ispezione quadratica**: in questo caso si procede per tentativi controllando gli slot di indice $(h(k) + c_1 i + c_2 i^2) mod\, m$ dove i è il numero di tentativi fatti e c_1, c_2 scelte in modo che la sequenza di tentativi copra (prima o poi) tutta la tabella (altrimenti si rischia il loop). Anche in questo caso chiavi con la stessa posizione danno luogo alla stessa sequenza di ispezione.
  - **Hashing doppio**: in questo caso gli indirizzi testati sono della forma $(h(k) + ih'(k)) mod\, m$ con h'(k) altra funzione di hash e i tentativo di inserimento. In questo caso è bene che $h'(k)$ sia sempre primo rispetto a m (m è una potenza di 2 e h'(k) restituisce sempre valori dispari, m è un numero primo e h'(k) restituisce sempre valori minori di m, ecc.)

L'indirizzamento aperto soffre di un "grosso" problema riguardante la cancellazione: non è possibile cancellare un elemento senza "rompere" la sequenza di ispezioni e ciò porterebbe alla perdita di tutte le chiavi memorizzate dopo l'oggetto che si vuole cancellare (e che condividono lo stesso hash). In questo caso è impossibile cancellare fisicamente un dato, tuttavia lo si può fare logicamente (facendo assumere al dato un valore DELETED) in modo da non rompere

la sequenza: il problema che sorge è che la struttura si appesantisce di parecchio dopo molte operazioni (di cancellazione soprattuto) anche se continua a funzionare correttamente.

19.1 Calcolo della funzione di hash

Una buona funzione di hash deve essere una funzione in grado di riempire in maniera uniforme la hashtable evitando le collisioni: se le chiavi hanno valori prossimi tra loro la funzione di hash deve essere in grado di separarle. In generale per generare una funzione di hash buona dovrei conoscere a priori la distribuzione statistica delle chiavi, e ciò non è sempre possibile!

- **Metodo della divisione**: $h(k) = k \, mod \, m$: la funzione è particolarmente veloce tuttavia è bene evitare certi valori per m, come ad esempio le potenze del 2, questo perchè se no h(k) dipenderebbe solo dai bit meno significativi di k. In generale è bene prendere m numero primo non vicino ad una potenza di 2 (come ad esempio m=701).
- **Metodo della moltiplicazione**: $h(k) = floor(m(kA \, mod \, 1))$ con $A \in \mathbb{R}, 0 < A < 1$. In questo caso il valore di m non è critico, quindi in generale si prendono potenze del 2 che sono molto veloci da calcolare. Un buon valore per A è il seguente $A = \frac{\sqrt{5}-1}{2}$.

20 Alberi e loro gestione

20.1 Alberi binari di ricerca

Un albero binario di ricerca o BST è un albero binario in cui per ogni nodo x vale la proprietà che tutti i nodi del sottoalbero sinistro siano minori o uguali di x e tutti quelli del sottoalbero destro siano maggiori.

20.2 Algoritmi di visita e gestione

E' possibile attraversare un albero secondo diversi modi:

- **Attraversamento inordine o simmetrico**: si visita (restituisce) prima il sottoalbero sinistro, poi la radice e poi quello destro.
- **Attraversamento preordine**: si visita prima la radice poi il sottoalbero sinistro e poi quello destro
- **Attraversamento postordine**: si visita prima il sottoalbero sinistro, poi quello destro e poi la radice.

In un BST l'elemento **minimo** è quello più a sinistra, mentre quello **massimo** è quello più a destra.

Sia dato un elemento x, il suo **successore** è l'elemento minimo del sottoalbero destro, il suo **predecessore** è l'elemento massimo del sottoalbero sinistro.

Pseudocodice per l'inserimento:

```
TREE-INSERT(T, z)
1   y := NIL
2   x := T.root
3   while x ≠ NIL
4     y := x
5     if z.key < x.key
6       x := x.left
7     else x := x.right
8   z.p := y
9   if y = NIL
10    T.root := z   //l'albero T e' vuoto
11  elsif z.key < y.key
12    y.left := z
13  else y.right := z
```

Pseudocodice per la cancellazione (prestare attenzione al caso in cui si cancella un nodo che ha due figli!!!):

```
TREE-DELETE(T, z)
1   if z.left = NIL or z.right = NIL
2     y := z
3   else y := TREE-SUCCESSOR(z)
4   if y.left ≠ NIL
5     x := y.left
6   else x := y.right
7   if x ≠ NIL
8     x.p := y.p
9   if y.p = NIL
10    T.root := x
11  elsif y = y.p.left
12    y.p.left := x
13  else y.p.right := x
14  if y ≠ z
15    z.key := y.key
16  return y
```

20.3 Alberi rosso-neri

Gli alberi rosso-neri sono BST che hanno la capacità di "autobilanciarsi", cioè l'altezza di una foglia (numero di nodi tra la foglia e la radice) però al più essere doppia rispetto all'altezza di un'altra.

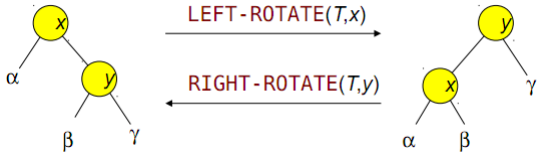
Gli alberi RB (red-black) godono delle seguenti proprietà:

- Ogni nodo o è rosso o è nero
- La radice è sempre nera
- Le foglie sono NIL (cioè nulle) e sono tutte nere

- I figli di un nodo rosso sono entrambi neri
- Per ogni nodo x tutti i cammini da x alle foglie discendenti contengono lo stesso numero di nodi neri
- bh(x) è l'**altezza nera** del nodo x: indica quanti nodi neri ci sono tra x e la radice (i rossi non vengono conteggiati)
- Per ragioni di comodita (ed efficienza) le foglie nere sono dei puntatori ad un nodo T.NIL.

Gli alberi rosso-neri si comportano (e si codificano) esattamente come dei BST per le operazioni di SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR e tutte queste operazioni hanno complessità $O(log(n))$. I meccanismi di INSERT e DELETE risultano anch'essi $O(log(n))$ tuttavia la loro codifica è più complessa perchè rischiano di violare le proprietà degli alberi rossoneri.

Le funzioni INSERT e DELETE si basano su un meccanismo fondamentale detto di "rotazione":




```
LEFT-ROTATE(T,x)
1  y := x.right
2  x.right := y.left  //il sottoalbero sinistro di y
                      //diventa quello destro di x
3  if y.left ≠ T.nil
4    y.left.p := x
5  y.p := x.p         //attacca il padre di x a y
6  if x.p = T.nil
7    T.root := y
8  elsif x = x.p.left
9    x.p.left := y
10 else x.p.right := y
11 y.left := x        //mette x a sinistra di y
12 x.p := y
```

L'operazione di rotazione preserva le proprietà dei BST!
La routine per l'inserimento dei nodi è la seguente:

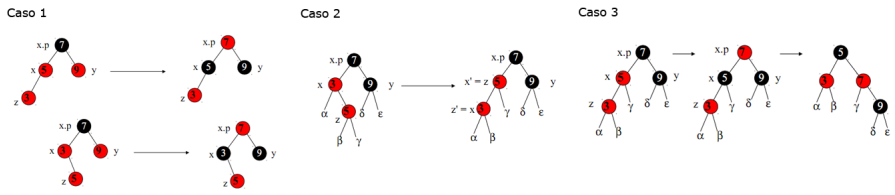
```
RB-INSERT(T, z)
1   y := T.nil       // y padre del nodo considerato
2   x := T.root      // nodo considerato
3   while x ≠ T.nil
4     y := x
5     if z.key < x.key
6       x := x.left
7     else x := x.right
8   z.p := y
9   if y = T.nil
10     T.root := z   //l'albero T e' vuoto
11  elsif z.key < y.key
12     y.left := z
13  else y.right := z
14  z.left := T.nil
15  z.right := T.nil
16  z.color := RED
17  RB-INSERT-FIXUP(T, z)
```

Fondamentalmente la routine si comporta come un inserimento normale in un BST solo che il nodo appena inserito è sempre rosso (per evitare di cambiare l'altezza nera) e i suoi figli puntano a T.NIL (e non a NULL).

Alla fine della procedura, tuttavia, è possibile che siano state violate delle proprietà degli alberi (ad esempio è stato aggiunto come figlio di un nodo rosso, i quali per definizione devono avere figli neri ecc). Per aggiustare tutte le possibili violazioni si richiama la routine RB-INSERT-FIXUP:

```
RB-INSERT-FIXUP(T, z)
1  if z = T.root
2    T.root.color = BLACK
3  else x := z.p              // x e' il padre di z
4    if x.color = RED
5      if x = x.p.left        // se x e' figlio sin.
6        y := x.p.right       // y e' fratello di x
7        if y.color = RED
8          x.color := BLACK          // Caso 1
9          y.color := BLACK          // Caso 1
10         x.p.color := RED          // Caso 1
11         RB-INSERT-FIXUP(T,x.p)    // Caso 1
12       else if z = x.right
13             z := x                // Caso 2
14             LEFT-ROTATE(T, z)     // Caso 2
15             x := z.p              // Caso 2
16         x.color := BLACK          // Caso 3
17         x.p.color := RED          // Caso 3
18         RIGHT-ROTATE(T, x.p)      // Caso 3
19     else (come 6-18, scambiando "right"↔"left")
```

I casi di violazione a cui ci si riferisce nel codice sono i seguenti:



La routine per la cancellazione è la seguente:

```
RB-DELETE(T, z)
1   if z.left = T.nil or z.right = T.nil
2     y := z
3   else y := TREE-SUCCESSOR(z)
4   if y.left ≠ T.nil
5     x := y.left
6   else x := y.right
7   x.p := y.p
8   if y.p = T.nil
9     T.root := x
10  elsif y = y.p.left
11    y.p.left := x
12  else y.p.right := x
13  if y ≠ z
14    z.key := y.key
15  if y.color = BLACK
16    RB-DELETE-FIXUP(T,x)
17  return y
```

Come per la INSERT esiste una routine in grado di aggiustare l'albero qualora qualche proprietà sia stata violata:

```
RB-DELETE-FIXUP(T, x)
1  if x.color = RED or x.p = T.nil
2    x.color := BLACK              // Caso 0
3  elsif x = x.p.left              // x e' figlio sinistro
4    w := x.p.right                // w e' fratello di x
5    if w.color = RED
6      w.color := BLACK            // Caso 1
7      x.p.color := RED            // Caso 1
8      LEFT-ROTATE(T,x.p)          // Caso 1
9      w := x.p.right              // Caso 1
10   if w.left.color = BLACK and w.right.color = BLACK
11     w.color := RED              // Caso 2
12     RB-DELETE-FIXUP(T,x.p)      // Caso 2
13   else if w.right.color = BLACK
14           w.left.color := BLACK // Caso 3
15           w.color := RED        // Caso 3
16           ROTATE-RIGHT(T,w)     // Caso 3
17           w := x.p.right        // Caso 3
18     w.color := x.p.color        // Caso 4
19     x.p.color := BLACK          // Caso 4
20     w.right.color := BLACK      // Caso 4
21     ROTATE-LEFT(T,x.p)          // Caso 4
19 else (come 4-21, scambiando "right"↔"left")
```

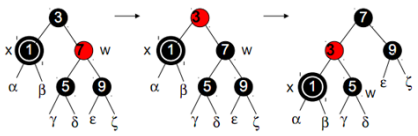
Idea: il nodo x passato come argomento si porta dietro un "nero in più", che, per fare quadrare i conti, può essere eliminato solo a certe condizioni

I casi riportati si riferiscono ai seguenti:

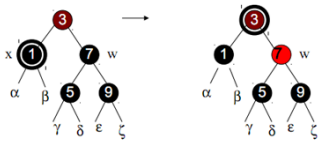
- Caso 0: x è un nodo rosso o la radice: viene colorato di nero
- Caso 1: x è un nodo nero, il suo fratello destro w è rosso e quindi il padre è nero (può ricadere nel caso 2, 3 o 4).
- Caso 2: x è nero, suo fratello destro w è nero ed entrambi i loro figli sono neri. Se arriviamo al caso 2 a partire dall'1 allora il genitore di x è rosso.
- Caso 3: x è nero, suo fratello destro w è nero con figlio sinistro rosso e figlio destro nero. Ricade nel caso 4.
- Caso 4: x è nero, suo fratello destro w è nero con figlio destro rosso.

Caso 1

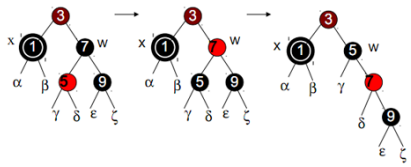
Caso 2



Caso 3



Caso 4



21 Grafi, loro rappresentazione e gestione

Un grafo è una coppia $G = (V, E)$ dove V è l'insieme dei nodi o vertici ed E è l'insieme degli archi o lati.

Riconosciamo i grafi orientati (in cui gli archi hanno anche un verso di percorrenza) e non orientati.

Per rappresentare i grafi in memoria si ricorre generalmente a due tecniche:

- **Rappresentazione con liste di adiacenza**: In questo caso viene creato un array di liste contenente un numero di elementi pari ai vertici. In ogni elemento della i-esima lista è memorizzato un numero j ad indicare che vi è un arco che parte dal vertice i-esimo ed arriva a quello j-esimo (nei grafi non orientati non fa differenza). Rappresentare un grafo con questa tecnica ha una complessità spaziale pari a $\Theta(|V| + |E|)$.

- **Rappresentazione con matrice di adiacenza**: in questo caso viene memorizzata una matrice $m \times m$ dove m è il numero dei vertici. La cella (i,j) indica che che vi è un arco che parte dall'elemento i-esimo ed arriva al j-esimo. Rappresentare un grafo con questa tecnica ha una complessità spaziale pari a $\Theta(|V|^2)$.

Se un grafo è "sparso" (ha pochi archi) la prima tecnica risulta molto più efficiente della seconda, se invece il grafo è "denso" è meglio usare una matrice per evitare di appesantire l'accesso ai nodi.

21.1 Visita in ampiezza (Breadth-first search, BFS)

In questo caso vogliamo sapere quali nodi del grafo sono raggiungibili da un nodo di partenza s.

L'idea alla base del BFS è quella di visitare prima tutti i nodi a distanza 1 da s, quindi passare a quelli che hanno distanza 2, poi 3 e così via. Mentre visitiamo i nodi li coloriamo in modo da tener traccia della progressione dell'algoritmo: un nodo è **bianco** se deve essere ancora visitato (tutti i nodi all'inizio sono bianchi), un nodo è **grigio** se lo abbiamo già visitato ma dobbiamo finire di visitare quelli ad esso adiacenti (il nodo sorgente è grigio all'inizio), un nodo è **nero** se lo abbiamo visitato assiame a tutti i suoi nodi adiacenti.

L'algoritmo parte con tutti i nodi bianci ad eccezione della sorgente, quindi mantiene traccia dei nodi adiacenti usando una coda (politica FIFO) che all'inizio contiene solo s. Ad ogni iterazione eliminiamo un elemento dalla coda e ne visitiamo i nodi adiacenti che sono ancora bianchi (e vengono aggiunti alla coda una volta che sono stati colorati di grigio):

```
BFS(G, s)
1  for each u ∈ G.V − {s}
2    u.color := WHITE
3   u.dist := ∞
4  s.color := GREY
5  s.dist := 0
6  Q := ∅
7  ENQUEUE(Q, s)
8  while Q ≠ ∅
9    u := DEQUEUE(Q)
10   for each v ∈ u.Adj
11      if v.color = WHITE
12        v.color := GRAY
13        v.dist := u.dist +1
14        ENQUEUE(Q, v)
15   u.color := BLACK
```

La complessità temporale di BFS è $T(n) = O(|V| + |E|)$. L'algoritmo memorizza la distanza del nodo dalla sorgente s.

21.2 Visita in profondità (Depth-first search, DFS)

L'algoritmo visita i nodi usando una politica LIFO: ogniqualvolta visitiamo un nodo visitiamo immediatamente anche quelli adiacenti e così via in maniera ricorsiva. In realtà anche se molto simile al BFS, il DFS serve per visitare tutti i nodi del grafo e non solo quelli raggiungibili da una determinata sorgente.

```
DFS(G)
1 for each u ∈ G.V
2   u.color := WHITE
3 time := 0
4 for each u ∈ G.V
5   if u.color = WHITE
6     DFS-VISIT(u)
```

```
DFS-VISIT(u)
1 u.color := GRAY
2 time := time + 1
3 u.d := time
4 for each v ∈ u.Adj
5   if v.color = WHITE
6     DFS-VISIT(v)
7 u.color := BLACK
8 u.f := time := time + 1
```

La complessità temporale di DFS è $T(n) = O(|V| + |E|)$

22 Ordinamento Topologico

Supponiamo di avere un grafo orientato aciclico (DAG: directed acyclic graph) che rappresenta le precedenze tra oggetti (eventi, ecc.). Un ordinamento topologico di un DAG è un ordinamento lineare tale per cui se c'è un arco da i a j, allora l'elemento i precede l'elemento j nell'ordinamento (ha una priorità più alta).

Questo tipo di ordinamento rispetta le precedenze tra eventi (se per esempio il DAG fornisce indicazioni su come assemblare un oggetto, l'ordinamento

topologico ci fornisce un possibile ordine da eseguire per poterlo assemblare correttamente).

```
TOPOLOGICAL-SORT(G)
1 L := ∅
2 for each u ∈ G.V
3   u.color := WHITE
4 for each u ∈ G.V
5   if u.color = WHITE
6     TOPSORT-VISIT(L, u)
7 return L
```

```
TOPSORT-VISIT(L, u)
1 u.color := GRAY
2 for each v ∈ u.Adj
3   if v.color = WHITE
4     TOPSORT-VISIT(L, v)
5 crea l'elemento di lista x
6 x.key := u
7 LIST-INSERT(L, x)
8 u.color := BLACK
```

Il tempo di esecuzione di TOPSORT è $\Theta(|V| + |E|)$.