CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Appunti per il corso di Algoritmi e Strutture Dati

a.a. 2019/2020

Nicoletta De Francesco, Luca Martini

Ultimo aggiornamento 3 marzo 2020

Indice

1 Complessità computazionale

1.1 Tempo di esecuzione dei programmi

Si dice *complessità* di un algoritmo la funzione che associa alla "dimensione" del problema il costo della sua risoluzione in base alla misura scelta. Possibili misure sono il tempo di esecuzione o lo spazio di memoria.

Supponiamo di voler valutare il tempo di esecuzione di un programma. Possiamo provarlo su un insieme significativo di ingressi e misurare il tempo di esecuzione. Quasi tutti gli ambienti di programmazione forniscono uno strumento di *profiling*, per mezzo del quale è possibile associare ad ogni istruzione un tempo e calcolare così il tempo totale di una esecuzione. Vedremo come sia possibile invece dare una misura dell'efficienza in termini di tempo, svincolata dalla particolare macchina sulla quale i programmi vengono eseguiti. Naturalmente, nei casi concreti di progetto di algoritmi, tale misura è solo indicativa e dovrà essere considerata insieme agli altri fattori che possono influenzare il tempo reale di esecuzione.

Ovviamente il tempo di esecuzione di un programma dipende dalla dimensione degli ingressi. La prima cosa da fare per valutare un programma è quindi individuare qual è la dimensione o misura degli ingressi. Questa misura può variare da programma a programma. Per esempio, per un programma di ordinamento la dimensione è in genere il numero di elementi da ordinare, per un programma che risolve sistemi di equazioni è il numero delle incognite, per un programma su liste la lunghezza delle liste, per un programma che lavora su matrici il numero di righe e di colonne.

Dato un programma P, chiamiamo $T_P(n)$ (o semplicemente $T(n)$ quando il programma cui ci si riferisce è chiaro dal contesto) il tempo di esecuzione di P per ingressi di dimensione n. Più precisamente, $T(n)$ indica il numero di unità di tempo che P impiega per elaborare un ingresso di dimensione n. $T(n)$ è una funzione da N in N ($N \to N$), dove con N indichiamo gli interi non negativi.

Esempio 1.1
Consideriamo la seguente funzione `max` *che cerca il massimo in un array di n interi.*

```
int max(int *a, int n) {
   int m=a[0];
   for (int i=1; i < n; i++)
      if (m < a[i]) m=a[i];
   return m;
}
```

Supponiamo per ora che il calcolatore impieghi una unità di tempo per ogni assegnamento ed ogni confronto. Il tempo per l'assegnamento `m=a[0]` *sarà quindi* 1*, mentre il tempo per il comando condizionale sarà* 1 *o* 2 *a seconda che si esegua il confronto e l'assegnamento oppure solo il confronto. Se consideriamo il caso peggiore, possiamo dire che il tempo è* 2*. Il corpo del comando ripetitivo viene eseguito* $n-1$ *volte ed ogni volta il tempo è* 2 *per il comando condizionale più* 1 *per l'incremento di* `i` *più* 1 *per il confronto. Quindi il tempo per il comando ripetitivo è* $4n-4$ *più* 1 *per l'iniziale assegnamento* `i=1` *più* 1 *per l'ultimo incremento* $i{+}{+}$*. Il tempo del comando* `return` *è 1. Il tempo dell'intero programma si ottiene sommando i tempi dei suoi comandi, quindi in conclusione abbiamo* $T_{max}(n) = 4n$*.*

La nostra assunzione che il tempo di esecuzione di un assegnamento o di un confronto sia l'unità di tempo non corrisponde generalmente alla verità. Per esempio, il tempo di valutazione dell'espressione alla destra di un assegnamento varia a seconda della complessità dell'espressione: infatti la traduzione in linguaggio macchina di assegnamenti diversi può risultare in generale in una sequenze di istruzioni macchina di diversa lunghezza. Inoltre, anche la traduzione dello stesso assegnamento può portare a diverse sequenze se si utilizzano due compilatori diversi. Il discorso si estende ulteriormente se si considerano

calcolatori diversi. Quindi, per fare un conto veramente preciso, sarebbe necessario conoscere la macchina che deve eseguire il programma e anche il funzionamento del compilatore.

La teoria della complessità computazionale dei programmi ha l'obiettivo di dare una stima approssimata del tempo di esecuzione dei programmi che ci permetta di ragionare su questi indipendentemente dalla macchina. L'approssimazione è raggiunta considerando la *"velocità di crescita"* del tempo in funzione della misura utilizzata. In altre parole, tempi di programmi diversi sono confrontati dal punto vista del loro comportamento *asintotico*. Per capire meglio questo discorso, consideriamo due programmi P e Q con tempi di esecuzione rispettivamente $T_P(n) = 2n^2$ e $T_Q(n) = 100n$. Abbiamo che che $T_P(n)$ cresce molto più rapidamente di $T_Q(n)$. In particolare, abbiamo che, per valori di n maggiori di 50, $T_P(n)$ è sempre maggiore di $T_Q(n)$. Esprimiamo questo fatto dicendo che $T_Q(n)$ ha complessità minore di $T_P(n)$. Quindi la stima della complessità di una funzione è fatta in base al suo comportamento asintotico, che ci dà una misura del crescere dei suoi valori in funzione del crescere della dimensione del problema: $T_P(n)$ cresce con il quadrato della dimensione, mentre $T_Q(n)$ cresce linearmente con essa. Possiamo quindi dire che Q è un programma più efficiente di P. Naturalmente, per valori minori di 50, P è preferibile a Q.

1.2 Complessità computazionale concreta ed espressioni $\mathcal{O}$-grande

Definizione 1.1
Consideriamo due funzioni $f, g : N \to N$. Si dice che $g(n)$ è di ordine $\mathcal{O}(f(n))$ ($g(n)$ è $\mathcal{O}(f(n))$ se esistono un intero n_0 ed una costante $c > 0$ tali che, per ogni $n \geq n_0$, $g(n) \leq cf(n)$.

In altre parole, $g(n)$ è di ordine $\mathcal{O}(f(n))$ se è uguale ad al più una costante moltiplicata per $f(n)$, fatta eccezione per alcuni valori *piccoli* di n. Quindi $f(n)$ è una limitazione superiore di $g(n)$. $\mathcal{O}(f(n))$ può essere visto come un insieme di funzioni, cui $f(n)$ appartiene: quindi possiamo anche scrivere $g(n) \in \mathcal{O}(f(n))$. Diciamo anche che $g(n)$ ha complessità $\mathcal{O}(f(n))$.

Nel caso del programma dell'esempio 1.1, abbiamo che $T_{max}(n) = 4n$ è $\mathcal{O}(f(n))$, con $f(n) = n$ per ogni n; basta considerare, per esempio, $n_0 = 0$ e $c = 4$. Da ora in poi indicheremo le funzioni con la parte destra della loro definizione: per esempio con n indichiamo la funzione $f(n) = n$ e diremo che T_{max} è $\mathcal{O}(n)$. Vale anche il contrario, cioè $n \in \mathcal{O}(4n)$, per $n_0 = c = 1$. Nel seguito elenchiamo alcune proprietà che sono utili per la determinazione della complessità delle funzioni, permettendo di semplificarne il calcolo.

Regola dei fattori costanti. Per ogni costante positiva k, $\mathcal{O}(f(n)) = \mathcal{O}(kf(n))$.

Dim. Prendiamo $g(n) \in \mathcal{O}(f(n))$; sappiamo che esistono n_0 e c tali che, per ogni $n \geq n_0$, $g(n) \leq cf(n)$; abbiamo $g(n) \leq (c/k\ k)f(n)$; se scegliamo $c' = c/k$, abbiamo $g(n) \leq c'(kf(n))$. Ora prendiamo $g(n) \in \mathcal{O}(kf(n))$; ciò vuol dire $g(n) \leq c(kf(n))$ per qualche c; abbiamo $g(n) \leq (ck)f(n)$.

Esempio 1.2
- $2n^2 \in \mathcal{O}(n^2)(n_0 = 0\ e\ c = 2)$.
- $2^{n+10} \in \mathcal{O}(2^n)(n_0 = 0\ e\ c = 2^{10}))$.
- $n^2 \in \mathcal{O}(\frac{1}{1000}n^2)(n_0 = 0\ e\ c = 1000)$.

Regola della somma.
Se $f(n) \in \mathcal{O}(g(n))$, allora $f(n) + g(n) \in \mathcal{O}(g(n))$.

Dim. Poiché $f(n) \in \mathcal{O}(g(n))$, allora esistono n_1 e c_1 tali che, per $n \geq n_1$, $f(n) \leq c_1 g(n)$. Abbiamo quindi, per $n \geq n_1$, $f(n) + g(n) \leq c_1 g(n) + g(n) = (c_1 + 1)g(n)$. Di conseguenza $f(n) + g(n) \in \mathcal{O}(g(n))$

per $n_0 = n_1$ e $c = c_1 + 1$.

Esempio 1.3

- $2n + 3n + 2$ *è* $\mathcal{O}(n)$. *Infatti, per la regola della somma,* $2n + 3n + 2$ *è* $\mathcal{O}(3n)$, *mentre* $3n$ *è* $\mathcal{O}(n)$ *per la regola dei fattori costanti.*
- $(n + 1)^2$ *è* $\mathcal{O}(n^2)$. *Infatti* $(n + 1)^2 = n^2 + 2n + 1$. *Per* $n \geq 3$, *abbiamo* $2n + 1 \leq n^2$, *quindi* $2n + 1 \in \mathcal{O}(n^2)$ *e, per la regola della somma,* $\mathcal{O}(n^2 + 2n) = \mathcal{O}(n^2)$.
- $n^2 + 2^n \in \mathcal{O}(2^n)$. *Per* $n \geq 4, n^2 \leq 2^n$.

Regola del prodotto. Se $f(n)$ è $\mathcal{O}(f1(n))$, $g(n)$ è $\mathcal{O}(g1(n))$, allora $f(n)g(n)$ è $\mathcal{O}(f1(n)g1(n))$.

Dim. Sappiamo che esistono n_1 e c_1 tali che, per ogni $n \geq n_1$, $f(n) \leq c_1 g(n)$ e che esistono n_2 e c_2 tali che, per ogni $n \geq n_2$, $f(n) \leq c_2 g(n)$.
Il teorema segue scegliendo come n_0 il maggiore fra n_1 e n_2 e come costante $c_1 c_2$.

La regola suddetta vale ovviamente se supponiamo, come sempre sarà per le funzioni che indicano la complessità, che $f(n)$ e $g(n)$ non sono mai negative.

Regola dei polinomi. Un polinomio di grado m, cioè $a_m n^m + ..a_1 n + a_0$, è $\mathcal{O}(n^m)$.

Dim. Con la regola dei fattori costanti e la regola della somma.

Regola transitiva. Se $f(n)$ è $\mathcal{O}(g(n))$ e $g(n)$ è $\mathcal{O}(h(n))$, allora $f(n)$ è $\mathcal{O}(h(n))$.

Valgono le seguenti affermazioni di facile dimostrazione.

- per ogni costante k, k è $\mathcal{O}(1)$.
- per $m \leq p$, n^m è $\mathcal{O}(n^p)$.

Da quanto visto finora, segue che esistono funzioni diverse che sono ognuna dell'ordine dell'altra. Si possono considerare come esempio le funzione $2n + 1$ e $3n$. La relazione $\mathcal{O}$ però non vale per qualsiasi coppia di funzioni. Ad esempio le due funzioni seguenti sono tra loro incommensurabili:

$$f(n) = \begin{cases} n & \text{se n è pari} \\ n^2 & \text{se n è dispari} \end{cases}$$

$$g(n) = \begin{cases} n^2 & \text{se n è pari} \\ n & \text{se n è dispari} \end{cases}$$

Se invece consideriamo le due funzioni:

$$f(n) = \begin{cases} n^2 & \text{se n è dispari} \\ n^3 & \text{se n è pari} \end{cases}$$

$$g(n) = \begin{cases} n^2 & \text{se n è primo} \\ n^3 & \text{se n è composto} \end{cases}$$

abbiamo che $f(n)$ è $\mathcal{O}(g(n))$ $(n_0 = 3, c = 1)$, ma non il contrario (i numeri dispari composti sono infiniti).

La proposizione seguente afferma che gli esponenziali a^n per $a > 1$, crescono di più di qualsiasi polinomiale e nessun esponenziale è $\mathcal{O}(n^k)$ per qualsiasi k. Per esempio, $2^n + n^3 \in \mathcal{O}(2^n)$.

Proposizione 1.1
Data $f(n) \in \mathcal{O}(n^k), f(n) \in \mathcal{O}(a^n)$ *per ogni* $k, a > 1$.

Le regole definite sopra permettono di semplificare le espressioni di complessità, togliendo fattori o termini di ordine inferiore.

1.3 Precisione e classi di complessità

Nella valutazione della complessità del tempo di esecuzione $T(n)$ di un programma, siamo interessati alla *più piccola* funzione fra quelle che approssimano $T(n)$. Per esempio, anche se il tempo della funzione `max` definita sopra è anche $\mathcal{O}(n^2)$, noi consideriamo $\mathcal{O}(n)$ come sua complessità.

D'altra parte, quando scegliamo la funzione che rappresenta la complessità del tempo di esecuzione di un programma, cerchiamo sempre una funzione che sia il più semplice possibile. Diciamo che una funzione è *semplice* se consiste di un unico termine con coefficiente 1. Per esempio, anche $f(n) = 0.01n$ fosse un limite preciso per il tempo di una funzione, si preferisce la funzione semplice $g(n) = n$ come rappresentante della sua classe di complessità.

Le più comuni classi di complessità degli algoritmi sono le seguenti (in ordine di grandezza). Si noti come, nel caso di complessità logaritmica, non sia necessario specificare la base del logaritmo: per ogni a e b, abbiamo $\mathcal{O}(\log_a n) = \mathcal{O}(\log_b n)$. Infatti $\log_a n = (\log_b n)(\log_a b)$ e, poiché $\log_a b$ è una costante, $\log_a n \in \mathcal{O}(\log_b n)$. Ogni classe $\mathcal{O}(n^k)$, per ogni $k \geq 0$, è detta di complessità *polinomiale*. Le funzioni con complessità minore di $\mathcal{O}(n)$ si dicono *sottolineari* (per esempio, oltre alle costanti, $radice(n)$ è sottolineare) mentre quelle con complessità maggiore *sopralineari*.

$\mathcal{O}(1)$	*complessità costante*
$\mathcal{O}(\log n)$	*complessità logaritmica*
$\mathcal{O}(n)$	*complessità lineare*
$\mathcal{O}(n \log n)$	*complessità* $n \log n$
$\mathcal{O}(n^2)$	*complessità quadratica*
$\mathcal{O}(n^3)$	*complessità cubica*
$\mathcal{O}(2^n)$	*complessità esponenziale*

2 Complessità dei programmi iterativi

2.1 Una regola induttiva per il calcolo della complessità

In questo capitolo daremo una regola per calcolare la complessità del tempo di esecuzione dei programmi non ricorsivi e in cui non vi siano cicli nelle chiamate di funzione (ricorsione nascosta). La regola è definita per induzione sulla sintassi dei costrutti del linguaggio: la complessità viene definita direttamente sui costrutti elementari, mentre la complessità dei costrutti composti è definita in base alla complessità dei costrutti componenti (definizione composizionale).

Nel seguito viene definita una grammatica che definisce la sintassi delle espressioni e dei comandi di un linguaggio di programmazione, che è un sottoinsieme del $C++$. I simboli nonterminali della grammatica sono le lettere maiuscole: E genera le espressioni e C i comandi; inoltre assumiamo che V generi un qualsiasi valore elementare, I un qualsiasi identificatore e O un qualsiasi operatore. Indichiamo con $I(E,..,E)$ una chiamata di funzione, mentre $I[E]$ è l'operatore di selezione di un array. Per semplicità, la notazione non è del tutto esatta nelle produzioni che riguardano le chiamate di funzione.

$$E ::= \quad V \mid E\ O\ E \mid O\ E \mid I(E,..,E) \mid I \mid I[E]$$

$$C ::= \quad I = E; \mid I[E] = E; \mid \texttt{if}\ (E)\ C \mid \texttt{if}(E)\ C\ \texttt{else}\ C \mid \texttt{for}\ (I = E; E; I = E)\ C \mid \{C..C\} \mid$$

$$\texttt{while}\ (E)\ C \mid \texttt{do}\ C\ \texttt{while}\ (E); \mid I(E,..,E); \mid \texttt{return}\ E;$$

La complessità dei costrutti è definita mediante una funzione

$\mathcal{C}$: comandi $\cup$ espressioni $\rightarrow$ classi di complessità

Dato un comando C, se $\mathcal{C}[\![C]\!] = \mathcal{O}(f(n))$, questo vuol dire che il tempo del comando è $\mathcal{O}(f(n))$. Inoltre somma e prodotto fra classi di complessità sono definite nel modo seguente:

$$\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$$

$$\mathcal{O}(f(n))\mathcal{O}(g(n)) = \mathcal{O}(f(n)g(n))$$

Nella definizione della complessità dei costrutti consideriamo il caso peggiore che si può verificare: ad esempio, per il condizionale `if (E) C1 else C2`, prendiamo il tempo massimo fra quelli occorrenti ad eseguire $C1$ e $C2$.

Complessità delle espressioni

1. $\mathcal{C}[\![V]\!] = \mathcal{C}[\![I]\!] = \mathcal{O}(1)$
   Il tempo per una espressione costante V o un identificatore I è costante (nel caso dell'identificatore, il tempo è quello necessario a recuperare il valore nella cella di memoria associata).
2. $\mathcal{C}[\![E1OE2]\!] = \mathcal{C}[\![E1]\!] + \mathcal{C}[\![E2]\!]$
   Per le espressioni composte, il tempo consiste nella somma fra i tempi occorrenti a valutare le due sottoespressioni più una costante per eseguire l'operazione. La costante è omessa poiché $\mathcal{C}[\![E1]\!] + \mathcal{C}[\![E2]\!]$ è comunque maggiore o uguale di $\mathcal{O}(1)$.
3. $\mathcal{C}[\![I[E]]\!] = \mathcal{C}[\![E]\!]$
   Il tempo per valutare una variabile indicizzata è dato dal tempo per valutare l'indice più il tempo (costante) per raggiungere la cella di memoria associata alla variabile.
4. $\mathcal{C}[\![I(E1,..,En)]\!] = \mathcal{C}[\![E1]\!] + \mathcal{C}[\![E2]\!] + .. + \mathcal{C}[\![En]\!] + \mathcal{C}[\![\{C..C\}]\!]$
   se nelle dichiarazioni compare `T I (T I1 , .., T In)`$\{C \ldots C\}$, (dove T è un tipo).

Il tempo per una chiamata di funzione è dato dal tempo per la valutazione dei parametri più il tempo necessario ad eseguire il comando corrispondente al corpo della funzione.

Complessità dei comandi

1. $\mathcal{C}[\![I = E;]\!] = \mathcal{C}[\![E]\!]$
   Il tempo di esecuzione di un assegnamento è dato dal tempo di valutazione dell'espressione alla destra dell'assegnamento più quello, omesso in quanto costante, per la modifica della opportuna cella di memoria.

2. $\mathcal{C}[\![\texttt{return } E;]\!] = \mathcal{C}[\![E]\!]$
   Il tempo di esecuzione di una istruzione `return` *è uguale al tempo per calcolare il valore dell'espressione.*

3. $\mathcal{C}[\![I[E1] = E2;]\!] = \mathcal{C}[\![E1]\!] + \mathcal{C}[\![E2]\!]$
   Il tempo di esecuzione di un assegnamento ad una variabile indicizzata è dato dal tempo di valutazione dell'indice più il tempo di valutazione dell'espressione alla destra dell'assegnamento.

4. $\mathcal{C}[\![\texttt{if } (E)\ C]\!] = \mathcal{C}[\![E]\!] + \mathcal{C}[\![C]\!]$
   Il tempo di esecuzione di un condizionale è dato dal tempo di valutazione dell'espressione più quello di esecuzione del comando.

5. $\mathcal{C}[\![\texttt{if } (E)\ C1 \texttt{ else } C2]\!] = \mathcal{C}[\![E]\!] + \mathcal{C}[\![C1]\!] + \mathcal{C}[\![C2]\!]$
   Il tempo di esecuzione di un condizionale è dato dal tempo di valutazione dell'espressione più la somma fra i tempi dei comani alternativi. Se fra questi due tempi esiste una relazione, il tempo è uguale al massimo fra i due (regola della somma).

6. $\mathcal{C}[\![\texttt{for } (I = E1; E2; I = E3)\ C]\!] = \mathcal{C}[\![E1]\!] + \mathcal{C}[\![E2]\!] + (\mathcal{C}[\![C]\!] + \mathcal{C}[\![E2]\!] + \mathcal{C}[\![E3]\!])\mathcal{O}(g(n))$,
   dove $\mathcal{O}(g(n))$ è la complessità del numero di volte che il ciclo è eseguito.
   Il tempo di esecuzione di un comando ripetitivo `for` *è dato dalla somma dei tempi di valutazione delle due espressioni corrispondenti alla inizializzazione e condizione più il tempo per eseguire il comando, l'incremento e la valutazione della condizione, il tutto moltiplicato per il numero di volte in cui il ciclo è eseguito.*

7. $\mathcal{C}[\![\texttt{while}(E)C]\!] = \mathcal{C}[\![E]\!] + (\mathcal{C}[\![E]\!] + \mathcal{C}[\![C]\!])\mathcal{O}(g(n))$
   dove $\mathcal{O}(g(n))$ è la complessità del numero di volte che il ciclo è eseguito.
   Il tempo di esecuzione di un comando ripetitivo `while` *è dato dal tempo di valutazione dell'espressione più la somma del tempo di valutazione dell'espressione con quello di esecuzione del comando, moltiplicata per il numero di volte che il ciclo è eseguito. Notare che* $\mathcal{C}[\![E]\!]$ *non è omesso poiché potrebbe essere anche g(n)=0.*

8. $\mathcal{C}[\![\texttt{do } C \texttt{ while}(E);]\!] = (\mathcal{C}[\![E]\!] + \mathcal{C}[\![C]\!])\mathcal{O}(g(n))$

9. $\mathcal{C}[\![\{C1..Cn\}]\!] = \mathcal{C}[\![C1]\!] + \mathcal{C}[\![C2]\!].. + \mathcal{C}[\![Cn]\!]$
   Il tempo di esecuzione di un blocco è uguale alla somma dei tempi di esecuzione dei comandi componenti.

È importante notare come, calcolando il tempo di un comando in modo composizionale come definito sopra, in certi casi si ottenga una valutazione non precisa, anche se corretta, come si vede nell'esempio seguente.

Esempio 2.1
Consideriamo il comando `if (0)` C_1 `else` C_2 *e supponiamo* $\mathcal{C}[\![C_1]\!] = \mathcal{O}(n^2)$ *e* $\mathcal{C}[\![C_2]\!] = \mathcal{O}(n)$. *Con la valutazione composizionale abbiamo un tempo* $\mathcal{O}(n^2)$, *mentre un limite preciso è* $\mathcal{O}(n)$.

2.2 Due algoritmi di ordinamento iterativi

Il problema dell'*ordinamento* è quello di ordinare in modo crescente o decrescente, in base ad una qualche definizione di relazione d'ordine, un insieme di elementi. In questa sezione presentiamo due algoritmi di ordinamento non ricorsivi. Nelle sezioni successive torneremo sul problema e definiremo altri algoritmi più efficienti. Consideriamo il caso in cui l'insieme sia memorizzato in un array. Il primo algoritmo che consideriamo, chiamato *selection-sort* (ordinamento per selezione) è il seguente: dato un array di n elementi, si cerca l'elemento più piccolo e lo si mette al primo posto. Poi si cerca il più piccolo fra i rimanenti $n-1$ elementi e lo si mette al secondo posto, e così via. Una possibile funzione è la seguente. Essa usa una funzione che scambia due elementi di un array. Supponiamo che gli elementi dell'array siano interi.

```
void exchange(int& x, int& y) {
   int temp;
   temp=x; x=y; y=temp;
 }

void selectionSort(int *A, int n) {
   for (int i=0; i< n-1; i++) {
      int min= i;
      for (int j=i+1; j< n; j++)
         if (A[j] < A[min]) min=j;
      exchange(A[i], A[min]);
   }
}
```

Calcoliamo ora, utilizzando la regola induttiva definita nella precedente sezione, la complessità del tempo di esecuzione della procedura definita, in funzione del numero di elementi nell'array, cioè n. Si vede facilmente che il tempo di tutti gli assegnamenti del programma è $\mathcal{O}(1)$; come esempio, consideriamo `j=i+1;`:

$$\mathcal{C}[\![\texttt{j = i + 1}]\!] = \mathcal{C}[\![\texttt{i + 1}]\!] = \mathcal{C}[\![\texttt{i}]\!] + \mathcal{C}[\![\texttt{1}]\!] = \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(1)$$

dove l'ultimo passaggio è ottenuto utilizzando la regola della somma. Calcoliamo ora la complessità del comando condizionale:

$$\mathcal{C}[\![\texttt{if(A[j] < A[min])min = j}]\!] = \mathcal{C}[\![\texttt{A[j] < A[min]}]\!] + \mathcal{C}[\![\texttt{min = j}]\!] = \mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(1)$$

Consideriamo ora il `for` più interno. Il suo corpo viene eseguito $n-i-1$ volte all'i-esima iterazione. Possiamo fare qualche considerazione per esprimere il tempo in termini solo di n. Poiché `i` non diventa mai minore di 0, abbiamo che $n-1$ è un limite per $n-i$. Quindi un limite al numero di volte in cui il `for` è eseguito è $n-1$, che è $\mathcal{O}(n)$. Abbiamo:

$$\mathcal{C}[\![\texttt{for}\ldots]\!] = \ = \mathcal{C}[\![\texttt{i + 1}]\!] + (\mathcal{C}[\![\texttt{n - 1}]\!] + \mathcal{C}[\![\texttt{if}\ldots]\!] + \mathcal{C}[\![\texttt{j + +}]\!])\mathcal{O}(n) = \mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(1)\mathcal{O}(n) = \mathcal{O}(n)$$

dove l'ultimo passagio è ottenuto applicando le regole della somma e del prodotto. L'assegnamento `min=i` e la chaimata alla funzione `exchange` hanno complessità costante. Quindi il corpo del `for` più esterno ha complessità $\mathcal{O}(n)$. Poiché esso è eseguito, $n-1$ volte, il `for` ha complessità $\mathcal{O}(n^2)$. Infine, poiché il blocco più esterno è costituto dal solo comando `for`, abbiamo che $\mathcal{O}(n^2)$ è anche l'ordine di complessità del tempo di esecuzione dell'intera funzione. Nel calcolo della complessità fin qui descritto, abbiamo approssimato per eccesso il tempo del ciclo più interno supponendo che ogni sua iterazione

avesse come lunghezza la massima lunghezza fra quelle delle singole iterazioni $(n-1)$. Cerchiamo di fare un'analisi più precisa dell'algoritmo. Mentre il corpo del for più esterno è eseguito esattamente $n-1$ volte, il corpo del for più interno è eseguito un numero di volte differente ad ogni iterazione, ed esattamente $n-i-1$ volte durante l'iterazione i-esima; quindi il numero di iterazioni in realtà è funzione sia di n che di i. Una stima più precisa del tempo di esecuzione dell'algoritmo è quindi $(n-1)+(n-2)+(n-3)+..1$, cioè la somma dei primi $n-1$ numeri interi, ed è quindi uguale a $\frac{(n-1)n}{2}$. In questo modo abbiamo dato una valutazione più precisa di $T(n)$, che è comunque $\mathcal{O}(n^2)$ e quindi coincide con la valutazione composizionale data precedentemente. Si noti come $\frac{(n-1)n}{2}$ sia anche il numero di confronti effettuati fra gli elementi dell'array: durante la prima esecuzione del for più interno facciamo $n-1$ confronti, durante la seconda $n-2$, ... durante la $(n-1)$-esima 1. Per quasi tutti gli algoritmi di ordinamento, il numero di confronti dà una misura del tempo di esecuzione.

Il secondo algoritmo di ordinamento che consideriamo, *bubble-sort* (ordinamento col metodo delle bolle), consiste nello scorrere l'array da "destra" verso "sinistra", scambiando due elementi contigui se non sono nell'ordine giusto. Ogni volta un elemento risale fino che non incontra un elemento più piccolo. Alla fine della iterazione i-esima del ciclo più esterno, gli elementi sono ordinati dal primo fino all'i-esimo. La complessità della procedura è $\mathcal{O}(n^2)$. Gli scambi avvengono solo fra posizioni contigue.

```
void bubbleSort(int* A, int n) {
   for (int i=0; i < n-1; i++)
      for (int j=n-1; j >= i+1; j--)
        if (A[j] < A[j-1])
              exchange(A[j],A[j-1]);
}
```

Confrontiamo ora i due algoritmi che abbiamo definito e che hanno la stessa complessità, cioè $\mathcal{O}(n^2)$, sia nel caso medio che nel peggiore. L'algoritmo *bubble-sort* ha il vantaggio di poter essere migliorato (pur mantenendo la stessa complessità) terminando alla prima iterazione del ciclo esterno che non ha effettuato scambi. Inoltre è preferibile se è importante fare scambi fra posizioni adiacenti piuttosto che lontane, come può accadere se si lavora su memorie secondarie.

D'altra parte, se gli elementi da ordinare hanno una grossa dimensione, conviene un algoritmo che fa meno scambi, e quindi *selection-sort* è meglio di *bubble-sort*. Infatti *bubble-sort* in media fa $\mathcal{O}(n^2)$ scambi (per la precisione ne fa in media $n^2/4$), mentre *selection-sort* ne fa sempre n. Questo perchè la chiamata alla funzione `exchange` si trova nel ciclo più interno di *bubble-sort* e solo in quello più esterno di *selection-sort*. Se con $S(n)$ indichiamo il numero di scambi in funzione del numero degli elementi, abbiamo che $S_{selection-sort}(n)$ è $\mathcal{O}(n)$, mentre $S_{bubble-sort}(n)$ è $\mathcal{O}(n^2)$. Questa considerazione è importante anche perchè mostra come in certi casi possano essere definite misure diverse dal tempo di esecuzione. Una strategia che può essere utile per migliorare l'efficienza degli scambi di *bubble-sort* quando gli elementi sono strutture è quella di mantenere un array di puntatori alle strutture e scambiare soltanto gli elementi di questo array risistemando alla fine gli elementi nell'ordine giusto con tempo lineare.

2.3 Esempio di programma con complessità logaritmica

La funzione seguente, dato un numero positivo n, calcola quante volte 2 divide n, cioè restituisce l'esponente della più alta potenza di 2 per cui n è divisibile.

```
int div2(int n) {
   int i=0;
   while (n % 2 == 0) {
      n=n/2; i++;
   }
```

```
    return i;
}
```

Il blocco interno è eseguito m volte, se m è il numero di volte che 2 divide n. Quindi nel peggiore dei casi, e cioè quando $n = 2^m$, è eseguito $\log_2 n$ volte. Abbiamo quindi che il tempo $T(n)$ è $\mathcal{O}(\log n)$.

2.4 Moltiplicazione di matrici

Consideriamo per semplicità matrici quadrate. Date due matrici quadrate $n \times n$, il prodotto righe per colonne è una matrice $n \times n$ tale che l'elemento $c_{ij}, i, j = 1, .., n$ è definito nel modo seguente:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$$

La complessità del tempo in questo caso si calcola in funzione di n, cioè del numero di righe e di colonne. Poiché si devono calcolare almeno n^2 prodotti fra elementi distinti, il numero di operazioni da elementari da effettuare ha un limite inferiore proporzionale a n^2 (per una trattazione dei limiti inferiori si veda la sezione 10). La moltiplicazione classica tra matrici, che calcola direttamente la formula definita sopra, può essere descritta dal seguente algoritmo, dove A e B sono due matrici $n \times n$ di cui si deve calcolare il prodotto e C è la matrice risultato:

```
void matrixMult(int A [n] [n], int B [n] [n], int C [n][n]) {
    for (int i=0; i <= n-1; i++)
        for (int j=0; j <= n-1, j++) {
            C[i][j]=0;
            for (int k=0; k <= n-1; k++)
                C[i][j]+=A[i][k]*B[k][j];
        }
}
```

La funzione `matrixMult` ha complessità $\mathcal{O}(n^3)$.

3 Programmi ricorsivi

Consideriamo la seguente definizione induttiva (o ricorsiva) della funzione fattoriale f(n)=n!:

$0! = 1$

$n! = n * (n-1)!$ se $n > 0$

Il seguente programma per il calcolo della funzione fattoriale ricalca la definizione data sopra:

```
int fact(int x) {
  if (x == 0) return 1;
  else return x*fact(x-1);
}
```

Ad esempio, il calcolo di `fact(3)` può essere rappresentato nel modo seguente:

```
fact(3)=3*fact(2)=3*2*fact(1)=3*2*1*fact(0)=3*2*1*1=6
```

Un altro esempio di funzione ricorsiva è la seguente, che decide se un numero naturale è pari non utilizzando la divisione:

```
int pari(int x) {
   if (x == 0) return 1;
   if (x == 1) return 0;
   return pari(x-2);
}
```

Una funzione ricorsiva è composta da alcuni casi di *base* in cui il risultato è dato immediatamente, per esempio il caso `x=0` per la funzione `fatt` o i casi `x=0` e `x=1` per la funzione `pari` e dai i casi ricorsivi, in cui la funzione richiama se stessa. Una metodologia per programmare una funzione ricorsiva è la seguente:

1. individuare i casi base in cui la funzione é definita immeditamente;
2. effettuare le chiamate ricorsive su un insieme *più piccolo* di dati, cioé un insieme più vicino ai casi base;
3. fare in modo che alla fine di ogni sequenza di chiamate ricorsive, si ricada nei casi base.

Le condizioni 2 e 3 assicurano che il calcolo di una funzione ricorsiva termini sempre. Altri esempi di funzioni ricorsive sono le seguenti, che calcolano rispettivamente il prodotto di due numeri naturali utilizzando la somma e il massimo comun divisore di due numeri naturali:

```
int mult(int x, int y) {
   if (x == 0) return 0;
   return y+mult(x-1,y);
}
```

```
int mcd(int x, int y) {
   if (x == y)  return x;
   if (x < y) return mcd(x, y-x);
   return mcd(x-y, y);
}
```

Esempi di funzioni che non rispettano la metodologia sopra descritta sono le seguenti:

```
int pari_errata(int x) {
   if (x == 0) return 1;
   return pari_errata(n-2);
}
```

```
int mcd_errata(int x, int y) {
   if (x == y)  return x;
   if (x < y) return mcd_errata(x, y-x);
   return mcd(x, y);
}
```

`pari_errata` non rispetta la regola 3 della metodologia perchè per i numeri dispari non si ricade sempre nell'unico caso base (`x=0`), mentre `mcd_errata` non rispetta la regola 2 della metodologia perchè con l'ultima istruzione non si richiama su un dato più piccolo, ma sugli stessi valori sia di `x` che di `y`.

Il metodo più naturale per provare proprietà delle funzioni ricorsive è l'induzione (vedi Appendice A). Per esempio, la correttezza del programma `fatt`, cioé `fatt(n)`=n! può essere dimostrata per induzione naturale nel modo seguente.

- Base. $n = 0$. In questo caso abbiamo `fatt(0)` $= 1 = 0!$.
- Induzione.
  Ipotesi: `fatt(n)`=n!.
  Tesi: `fatt(n+1)`=(n+1)!
  Dim.

$$\begin{array}{lll} \texttt{fatt}(n+1) & = (n+1) * \texttt{fatt}(n+1-1) & \text{per definizione di } \texttt{fatt} \text{ e poiché } (n+1) > 0 \\ & = (n+1) * \texttt{fatt}(n) & \\ & = (n+1) * n! & \text{per ipotesi induttiva} \\ & = (n+1)! & \text{per definizione di fattoriale.} \end{array}$$

La programmazione ricorsiva ha un'applicazione naturale nella manipolazione di tipi di dato definiti ricorsivamente. Un esempio di tale tipo tipo è la *lista di elementi*, che può essere definita come segue:

- la lista vuota è una lista;
- un elemento seguito da una lista è una lista.

Una buona metodologia per scrivere funzioni che lavorano su liste è quella di ricalcare la definizione di lista, cioè considerare come caso base la lista vuota e come caso ricorsivo il caso di un elemento seguito da una lista. Per esempio, se la lista è rappresentata con elementi della struttura:

```
struct Elem {
   int   inf;
   Elem* next;
};
```

e la lista vuota è rappresentata con `NULL`, le funzioni seguenti sono definite con la metodologia suddetta: la prima calcola la lunghezza di una lista, la seconda il numero di volte che un elemento compare in una lista (il test `(p == NULL)` può essere sostituito da `(!p)`.

```
int length(Elem* p) {
   if (p == NULL) return 0;
   return 1+length(p->next);
}
```

```
int howMany(Elem* p, int x) {
   if (p == NULL) return 0;
   return (p->inf == x)+howMany(p->next, x);
}
```

Naturalmente non sempre i casi non ricorsivi di una funzione che opera su liste coincidono esattamente con il caso della lista vuota, ma ci possono essere degli altri argomenti su cui la funzione dà immediatemente il risultato. In genere i casi non ricorsivi sono un soprainsieme dei casi base del tipo di dato. Nella seguente funzione che cerca se un elemento compare nella lista, i casi non ricorsivi sono due: lista vuota e lista il cui primo elemento è quello cercato.

```
int belongs(Elem* l, int x) {
   if (l == NULL) return 0;
   if (l->inf == x) return 1;
   return belongs(l->next, x);
}
```

La correttezza delle funzioni che lavorano su liste può essere in genere dimostrata con l'induzione naturale o completa sulla lunghezza della lista.

Esempio 3.1 (Esempi di funzioni su liste)

- *La funzione seguente cancella tutti gli elementi uguali ad x che compaiono in una lista. Si noti come il parametro* `l` *sia passato per riferimento.*

```
void deletex(Elem* & l, int x) {
   if (l == NULL) return;
   if (l->inf == x) {
      Elem *l1=l; l=l->next; delete l1; deletex(l,x);
   }
   else deletex(l->next, x);
}
```

- *Le seguenti funzioni calcolano la concatenazione tra due liste* `l1` *e* `l2`*; sono entrambe definite ricorrendo sulla prima lista; la prima restituisce la lista come risultato, mentre la seconda modifica il parametro che è il puntatore di inizio della prima lista.*

```
Elem* append(Elem* l1, Elem* l2) {
   if (l1 == NULL) return l2;
   l1->next=append(l1->next, l2); return l1;
}
```

```
void append(Elem* & l1, Elem* l2) {
   if (l1 == NULL) l1=l2;
   else append(l1->next, l2);
}
```

- *La funzione seguente elimina l'ultimo elemento di una lista:*

```
void tailDelete(Elem* & l) {
   if (l == NULL) return;
   if (l->next == NULL) { delete l; l=NULL; }
   else tailDelete(l->next);
}
```

- *La funzione seguente aggiunge un elemento in fondo ad una lista:*

```
void tailInsert(Elem* & l, int x) {
   if (l == NULL) { l=new Elem; l->inf=x; l->next=NULL; }
   else tailInsert(l->next,x);
}
```

4 Complessità dei programmi ricorsivi

Riconsideriamo la funzione che calcola il fattoriale di un numero naturale.

```
int fact (int n) {
   if (n == 0)  return 1;
   else return n*fact(n-1);
}
```

Per calcolare l'ordine del tempo di esecuzione $T(n)$ (o di una qualsiasi altra misura che vogliamo considerare) di una funzione ricorsiva, non possiamo applicare direttamente il ragionamento che abbiamo adottato per il caso di programmi iterativi. Cercheremo quindi di definire $T(n)$ in modo induttivo: data una funzione ricorsiva, la studiamo in due casi: il caso in cui la funzione non effettua nessuna chiamata ricorsiva (caso base della ricorsione) e quelli in cui la effetttua. Nel primo caso facciamo la valutazione della complessità del tempo di esecuzione direttamente come per le funzioni iterative. Nel secondo caso definiamo la funzione incognita $T(n)$ in funzione del tempo $T(k)$ di ogni chiamata ricorsiva. Ovviamente, perchè la definizione induttiva sia corretta, è necessario assumere $k \leq n$. Non possiamo però semplificare completamente le espressioni $\mathcal{O}$-grande così ottenute, perchè esse in generale contengono funzioni concrete oltre a classi di funzioni. Quindi è necessario a questo punto rimpiazzare le classi di funzioni che compaiono in una espressione con funzioni concrete, anche se espresse mediante costanti simboliche; per esempio, al posto di $\mathcal{O}(1)$ possiamo considerare una costante c e al posto di $\mathcal{O}(n)$ una funzione cn. In generale, al posto di $\mathcal{O}(f(n))$, possiamo considerare $cf(n)$, dove c è una costante. Questa definizione induttiva di $T(n)$ si chiama *relazione di ricorrenza*. Essa deve essere risolta per trovare la funzione incognita $T(n)$. Nel caso di `fatt`, consideriamo come base il valore 0:

$$T(0) \in \mathcal{C}[\![\texttt{n == 0}]\!] + \mathcal{C}[\![\texttt{return 1;}]\!] = \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(1)$$

Per $T(n)$, se $n > 0$, abbiamo un tempo $\mathcal{O}(1)$ complessivo per il test, la chiamata ricorsiva e la moltiplicazione più il tempo per l'esecuzione della funzione applicata a $n-1$. Quindi

$$T(n) = \mathcal{O}(1) + T(n-1)$$

A questo punto rimpiazziamo gli $\mathcal{O}(1)$ di $T(1)$ e $T(n)$ con simboli generici di costante, diversi fra loro perchè corrispondono a pezzi di programma diversi, e abbiamo la seguente relazione di ricorrenza:

$$\begin{array}{ll} T(0) & = a \\ T(n) & = b + T(n-1) \quad n > 0 \end{array}$$

Proviamo a calcolare i valori di $T(n)$:

$$\begin{array}{lll} T(1) & = b + T(0) & = b + a \\ T(2) & = b + T(1) & = 2b + a \end{array}$$

In generale, per $i \geq 0$, avremo $T(i) = ib + a$. Dimostriamo questo risultato con l'induzione naturale.

- Base. $i = 0$. Abbiamo $T(0) = 0b + a = a$.
- Induzione.
  Ipotesi. $T(i) = ib + a$
  Tesi. $T(i+1) = (i+1)b + a$
  Dim.

$$\begin{array}{lll} T(i+1) & = b + T(i) & \textit{per definizione di } T \\ & = b + ib + a & \textit{per ipotesi induttiva} \\ & = (i+1)b + a & \end{array}$$

Quindi, poiché $T(n) = nb + a$, abbiamo $T(n) \in \mathcal{O}(n)$.

4.1 Una versione ricorsiva di *selection-sort*

Consideriamo come secondo esempio la seguente "versione ricorsiva" dell'ordinamento per selezione:

```
void r_selectionSort (int* A, int n, int i=0) {
   if (i < n-1) {
      int min= i;
      for (int j=i+1; j<n; j++)
         if (A[j] < A[min]) min=j;
      exchange(A[i] ,A[min] );
      r_selectionSort (A, n, i+1);
   }
}
```

Se l'array ha un solo elemento ($i = n - 1$), abbiamo un tempo costante per il test `i<n-1`. Se ci sono n elementi, abbiamo un tempo lineare per il `for` più un tempo costante per la chiamata di `exchange` più il tempo della chiamata ricorsiva della funzione su di una porzione di array di $n - 1$ elementi. Quindi la relazione di ricorrenza è:

$$\begin{aligned} T(1) &= a \\ T(n) &= bn + T(n-1) \quad n > 1 \end{aligned}$$

Facendo qualche prova abbiamo:

$$\begin{aligned} T(1) &= a \\ T(2) &= 2b + a \\ T(3) &= 3b + 2b + a \end{aligned}$$

Abbiamo la seguente soluzione, che dimostra che $T(n)$ è $\mathcal{O}(n^2)$, come nel caso di `selection-sort`:

$$T(n) = \frac{bn(n+1)}{2} - b + a$$

4.2 L'algoritmo di ordinamento *mergeSort*

L'algoritmo di ordinamento *mergeSort* (ordinamento per fusione), oltre che su array, è applicato spesso alle liste. Consideriamo perciò la sua versione "su lista". L'algoritmo è definito induttivamente nel modo seguente. Data una lista, se la sua lunghezza è 0 o 1 allora *mergeSort* restituisce la lista inalterata. Altrimenti, si divide la lista in due sottoliste uguali (o quasi uguali se il numero di elementi è dispari). Queste vengono ordinate ricorsivamente con *mergeSort*. Alla fine le liste risultanti sono fuse in un'unica lista ordinata. La funzione `mergeSort` utilizza due funzioni, una (`split`) per dividere la lista in due liste, e l'altra (`merge`) per fare la fusione ordinata di due liste.

```
void mergeSort(Elem* &list1) {
     if (list1 == NULL || list1->next == NULL) return;
     Elem* list2=NULL;
     split(list1, list2);
     mergeSort(list1);
     mergeSort(list2);
     merge(list1,list2);
}
```

La funzione `split` divide la lista in ingresso in due sottoliste di uguale dimensione. Una possibile implementazione di `split` è la seguente: data una lista `list1`, `split` costruisce una seconda lista `list2` contenente gli elementi di `list1` con posizione pari e inoltre modifica `list1` in modo che questa alla fine contenga soltanto gli elementi che erano di posizione dispari.

```
void split(Elem* & list1, Elem* & list2) {
    if (list1 == NULL || list1->next == NULL) return;
    Elem *l = list1->next;
    list1->next = l->next;
    l->next = list2; list2 = l;
    split(list1->next, list2);
}
```

La complessità di `split`, prendendo come dimensione il numero di elementi della lista, è $\mathcal{O}(n)$. Per la precisione, il ciclo viene eseguito un numero di volte uguale a $\frac{n}{2}$.

La funzione `merge` riceve in ingresso due liste già ordinate e le fonde in modo da ottenere una lista ordinata, scorrendo contemporaneamente le due liste. Il risultato va nel primo parametro (`list1`). Con più precisione, date le due liste `list1` e , `list2`, se una delle due liste è vuota, allora `merge` restituisce l'altra. Altrimenti, se il primo elemento di `list1` è minore o uguale del primo elemento di `list2` , `merge` restituisce la lista che ha come primo elemento il primo elemento di `list1` e come resto il risultato di merge applicato a `list1` senza il primo elemento e a `list2`. Altrimenti `merge` restituisce la lista che ha come primo elemento il primo elemento di `list1` e come resto il risultato di merge applicato a `list1` e a `list2` senza il primo elemento.

```
void merge(Elem* &list1, Elem* list2) {
    if (list1 == NULL) { list1=list2; return;}
    if (list2 == NULL) return;
    if (list1->inf <= list2->inf)
        merge(list1->next, list2);
    else { merge(list2->next,list1);list1=list2; }
}
```

Dimostriamo che `merge` è corretta, cioè che,

- *Asserto A*: date due liste ordinate `list1` e `list2`, `merge` restituisce la lista ordinata che contiene tutti e soli gli elementi di `list1` e di `list2`.

Facciamo una dimostrazione per induzione naturale sulla somma del numero di elementi di `list1` più il numero di elementi di `list2`. Data una list `l`, indichiamo con |`l`| il numero dei suoi elementi.

- Base. |`list1`| + |`list2`| = 0. A è verificato.
- Induzione. Ipotesi: A è vero per liste tali che |`list1`| + |`list2`| = n.
  Tesi: A è vero per liste tali che |`list1`| + |`list2`| = $n + 1$.
  Dim. Consideriamo due liste `list1` e `list2` tali che |`list1`| + |`list2`| = $n + 1$. Se la prima o la seconda lista sono vuote, allora viene restituita l'altra lista che verifica ovviamente A. Supponiamo che il primo elemento di `list1` (d) sia minore o uguale del primo elemento di `list2` (l'altro caso è analogo); se chiamiamo $l1$ la lista uguale a `list1` senza il primo elemento (cioè la lista puntata da `list1->next`), allora `merge` è applicata ricorsivamente a $l1$ e a `list2`. Poiché $|l1| + |$`list2`$| = n$, per ipotesi induttiva, abbiamo che $l2$=`merge(l1, list2)` verifica A, cioè $l2$ è la lista ordinata che contiene tutti e soli gli elementi di $l1$ e di `list2`. Il risultato della chiamata di merge è la lista che contiene d come primo elemento e $l2$ come resto. Sia l tale lista. Poiché d è l'elemento minore fra

quelli di `list1` e `list2`, segue che l verifica A.

Vediamo ora la complessità di `merge` considerando come dimensione la somma del numero di elementi di `list1` con il numero di elementi di `list2`. La relazione di ricorrenza è :

$$\begin{array}{ll} T(0) = T(1) & = a \\ T(n) & = b + T(n-1) \quad n > 1 \end{array}$$

che è la stessa di `fatt`. Quindi $T_{\texttt{merge}}(n) \in \mathcal{O}(n)$. Torniamo ora a `mergeSort`. Si può dimostrare per induzione sul numero di elementi della lista e tenendo conto della correttezza di `split` e `merge`, che anche `mergeSort` è corretta. cioè che, data una lista, `mergeSort` restituisce la lista ordinata che contiene gli elementi della lista data.

Per il calcolo della complessità del tempo di esecuzione di `mergeSort`, consideriamo inizialmente liste con numero di elementi pari ad una potenza di due. Se c'è un solo elemento, abbiamo un tempo costante per i due test; se ci sono n elementi, abbiamo:

$$T(n) = \quad \mathcal{O}(1) + \mathcal{C}[\![split(list1, list2)]\!] + \mathcal{C}[\![merge(MergeSort(list1), MergeSort(list2))]\!]$$

dove $\mathcal{O}(1)$ è il tempo dei test. Poiché abbiamo visto che `split` e `merge` hanno complessità lineare con il numero complessivo di elementi delle loro liste in ingresso e le liste cui si applica ricorsivamente *mergeSort* hanno la metà della lunghezza di quella iniziale, abbiamo:

$$\begin{array}{ll} T(1) & = \mathcal{O}(1) \\ T(n) & = \mathcal{O}(n) + 2T(\frac{n}{2}) + \mathcal{O}(n) \quad n > 1 \end{array}$$

Da cui la relazione di ricorrenza ($\mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n)$):

$$\begin{array}{ll} T(1) & = a \\ T(n) & = 2T(\frac{n}{2}) + bn \quad n > 1 \end{array}$$

Proviamo a calcolare i valori di $T(n)$:

$$\begin{array}{lll} T(1) & = a & \\ T(2) & = 2T(1) + 2b & = 2a + 2b \\ T(4) & = 2T(2) + 4b & = 4a + 8b \\ T(8) & = 2T(4) + 8b & = 8a + 24b \end{array}$$

La soluzione della relazione, per n potenza di due,è :

$$T(n) = an + bn(\log_2 n)$$

Dimostriamo questo risultato per induzione, cioè $T(2^i) = a2^i + bi2^i$.

- Base. $i = 0$. $T(1) = a$.
- Induzione.
  Ipotesi. $T(2^i) = a2^i + bi2^i$. Tesi. $T(2^{i+1}) = a2^{i+1} + b(i+1)2^{i+1}$.

$$\begin{array}{lll} T(2^{i+1}) & = 2T(i) + b2^{i+1} & \text{per la relazione di ricorrenza} \\ & = 2(a2^i + bi2^i) + b2^{i+1} & \text{per ipotesi induttiva} \\ & = a2^{i+1} + b(i+1)2^{i+1} & \end{array}$$

Poiché la funzione $T(n)$ è crescente, anche per valori di n che non sono potenze di 2, abbiamo che $T(n)$ è $\mathcal{O}(n \log n)$. Notiamo inoltre che il tempo di esecuzione di `mergeSort` non dipende dalla disposizione iniziale dei dati nella lista da ordinare, e quindi $\mathcal{O}(n \log n)$ è la complessità del tempo sia nel caso migliore che nel medio che nel peggiore. Può essere definito un algoritmo di ordinamento per fusione con complessità $\mathcal{O}(n \log n)$ anche per ordinare gli elementi di un array: l'analogo della funzione di `split` è ottenuto semplicemente dividendo l'array in due metà, mentre l'analogo di `merge` va definito opportunamente. In conclusione l'algoritmo `mergeSort` è migliore degli algoritmi quadratici visti precedentemente.

4.3 L'algoritmo di ordinamento *quickSort*

L'algoritmo di ordinamento *quickSort* (ordinamento veloce) è definito nel modo seguente. Dato un array con indici da inf a *sup*, si sceglie un elemento dell'array, detto *perno*. Quindi si modifica l'array in modo che la prima parte dell'array, e cioè la porzione da inf a $k-1$ per qualche k, contenga elementi minori o uguali al perno e la seconda parte, e cioè la porzione da k a *sup*, contenga elementi maggiori o uguali al perno. Si applica quindi ricorsivamente *quickSort* alle due parti dell'array così ottenute. Il caso base della ricorsione è quando l'array ha un solo elemento.

L'algoritmo può essere descritto mediante una funzione `quickSort` descritta come segue. Si sceglie come perno l'elemento centrale. Una volta scelto il perno, si effettua la suddivisione dell'array con il comando `while`: si parte dai due estremi dell'array con due indici, il cursore sinistro `s` e il cursore destro `d`, posti inizialmente rispettivamente uguali a `inf` e `sup`, e si fanno scorrere l'uno da sinistra a destra e l'altro da destra a sinistra finchè non si incontrano, o meglio `s` "scavalca" `d`. Durante lo scorrimento, si cerca di scambiare ogni elemento maggiore del perno, trovato partendo dall'estremo "sinistro", con uno minore, raggiunto partendo da "destra". Notare che il perno può cambiare posizione durante gli scambi. I due ultimi comandi alternativi scartano i casi particolari in cui `inf >= d` (che può accadere se il perno si trova alla fine degli scambi nella prima posizione dell'array) o `s >= sup` (che può accadere se il perno si trova alla fine degli scambi nell'ultima posizione dell'array). Entrambe le condizioni sono false se `inf=sup`.

```
void quickSort(int A[], int inf=0, int sup=n) {
    int perno = A[(inf + sup) / 2], h = inf, k = sup;
    while (h <= k) {
        while (A[h] < perno) h++;
        while (A[k] > perno) k--;
        if (h > k) break;
        exchange(A[h], A[k]);
        h++; k--;
    };
    if (inf < k)
        quickSort(A, inf, k);
    if (h < sup)
        quickSort(A, h, sup);
}
```

Per esempio, la chiamata a `quicksort` su un array contenente i valori `[3,5,2,1,1]` genera le seguenti chiamate ricorsive:

```
quickSort(A,0,4) con A=[3,5,2,1,1]
quickSort(A,0,1) con A=[1,1,2,5,3]
quickSort(A,3,4) con A=[1,1,2,5,3]
```

Se l'array contiene un solo elemento, dobbiamo considerare solo un tempo costante per alcuni assegnamenti e test. Se gli elementi sono $n > 1$, abbiamo un tempo costante per gli assegnamenti e i test più un tempo $\mathcal{O}(n)$ per il `while` più i tempi per le chiamata ricorsive di `quickSort` su una porzione di array di k elementi e una di $n - k$ elementi. La relazione di ricorrenza è quindi

$$\begin{aligned} T(1) &= a \\ T(n) &= bn + T(k) + T(n-k) \quad n > 1 \end{aligned}$$

Il tempo $T(n)$ è diverso a seconda del valore di k ad ogni coppia di chiamate ricorsive. Il caso migliore è quando k è sempre uguale circa a $\frac{n}{2}$, per cui la relazione di ricorrenza è uguale a quella di *mergeSort* e quindi $T(n)$ è $\mathcal{O}(n \log n)$. In questo caso il perno è sempre maggiore della metà degli elementi. Il caso peggiore è quando, ogni volta, o $k = 1$ o $n - k = 1$, per cui abbiamo la stessa relazione di ricorrenza della funzione `r_selectionSort`, con tempo $\mathcal{O}(n^2)$. In questo caso il perno è sempre il minimo o il massimo dell'array. Negli altri casi abbiamo un tempo compreso fra $\mathcal{O}(n \log n)$ e $\mathcal{O}(n^2)$. Si può dimostrare che il tempo medio, cioè la media di tutti i tempi se si considerano equiprobabili tutte le configurazioni iniziali dell'array, è $\mathcal{O}(n \log n)$.

La scelta del perno come elemento centrale dell'array non è necessaria: un qualsiasi elemento può essere preso come perno. Un possibile miglioramento dell'algoritmo è quello che si ottiene scegliendo di volta in volta un perno "migliore", per esempio prendendo più elementi a caso e considerando quello mediano fra di essi. Anche con questi miglioramenti, però, il tempo rimane sempre dello stesso ordine, cioè i miglioramenti non modificano la complessità del tempo di esecuzione. Naturalmente questo non vuol dire che gli algoritmi non debbano essere migliorati, perchè è anche importante ridurre il tempo, pur mantenendo l'ordine di complessità.

4.4 Confronto fra algoritmi di ordinamento

In generale la scelta di un algoritmo può dipendere da tanti fattori, come ad esempio il fatto che lavoriamo sulla memoria principale o su quella esterna o se consideriamo parametri come la semplicità, la comprensibilità, la portabilità di un algoritmo. Inoltre spesso un parametro di cui tenere conto è l'occupazione di memoria. Per orientarci nella scelta fra un algoritmo e un altro, bisogna anche nei casi concreti valutare anche l'entità delle "costanti nascoste" nelle espressioni di complessità. Questo vuol anche dire che nella scelta di un algoritmo bisogna tener conto anche di valori del tempo di esecuzione ottenuti sperimentalmente: per esempio, valutazioni sperimentali hanno provato che per valori all'incirca minori di 100 conviene usare per l'ordinamento gli algoritmi quadratici che sono più semplici e non hanno chiamate ricorsive e che *quickSort* è più efficiente di *mergeSort* su array.

Gli algoritmi che abbiamo visto lavorano sulla memoria principale (ordinamento interno). Quando bisogna ordinare file residenti su memorie secondarie, i parametri di giudizio variano: quello che conta è il numero di trasferimenti di blocchi da memoria secondaria a principale e viceversa e il tempo di elaborazione può non essere un elemento critico.

5 Classificazione di alcune relazioni di ricorrenza

5.1 Metodo del *divide et impera*

Quasi tutti gli algoritmi ricorsivi che abbiamo visto sono basati su di una tecnica di progetto di algoritmi chiamata *divide et impera*, cioè per risolvere un problema per un certo insieme di dati, lo si affronta applicando l'algoritmo ricorsivamente su sottoinsiemi dei dati del problema (*partizioni*) e ricomponendo poi i risulati (lavoro di *combinazione*). La struttura di un tipico algoritmo di questo tipo è la seguenete, dove m è la dimensione dei dati del problema che permette una sua risoluzione immediata senza chiamate

ricorsive, b è il numero dei sottoinsiemi in cui i dati sono divisi, e $S_{i_1} \dots S_{i_a}, (1 \leq i_1, .., i_a \leq b)$ sono i sottoinsiemi cui l'algoritmo è applicato ricorsivamente. Con $|S|$ indichiamo la cardinalità di S, cioè il numero dei suoi elementi.

```
void dividetimpera(S){
    if (|S| <= m)
        // ¡risolvi direttamente il problema¿
    else {
        // ¡dividi S in b sottoinsiemi S1 ... Sb¿
        dividetimpera(S1);
        //...
        dividetimpera(Sb);
        // ¡combina i risultati ottenuti¿
    }
}
```

L'efficienza dell'algoritmo dipende in genere dall'uniformità della dimensione dei sottoinsiemi in cui l'insieme di partenza è diviso, nel senso che l'efficienza è maggiore quanto più i sottoinsiemi hanno dimensioni che differiscono meno fra loro. Se i sottoinsiemi sono delle stesse dimensioni (le partizioni sono bilanciate), la relazione di ricorrenza sarà del tipo seguente, dove d è una costante che rappresenta il tempo per risolvere il problema quando la dimensione dei dati è minore o uguale di m, b è il numero di sottoinsiemi in cui l'insieme è diviso , a indica su quanti di questi sottoinsiemi la procedura è chiamata ricorsivamente e $c(n)$ rappresenta il tempo di combinazione dei risultati:

$$\begin{array}{ll} T(n) = d & n \leq m \\ T(n) = aT(\frac{n}{b}) + c(n) & n \geq m \end{array}$$

Consideriamo prima il caso in cui $c(n)$ è costante (combinazione costante): questo vuol dire che il tempo per mettere insieme i risulati è indipendente da n. Se $m = 1$, a relazione ha la forma:

$$\begin{array}{ll} T(n) = d & n \leq 1 \\ T(n) = aT(\frac{n}{b}) + c & n > 1 \end{array}$$

Risolviamo per tentativi con le potenze di b:

$$\begin{array}{lcl} T(b^0) & = & T(1) = d \\ T(b^1) & = & ad + c \\ T(b^2) & = & a^2 d + ac + c \\ .. & & \\ T(b^i) & = & a^i d + (a^{i-1} + a^{i-2} + ..a + 1)c \end{array}$$

Quindi, se n è una potenza di b,

$T(n) = da^{\log_b n} + c \sum_{i=0}^{\log_b n - 1} a^i$

Se $a = 1$, abbiamo $T(n) = d + c \log_b n$ e quindi $T(n) \in \mathcal{O}(\log n)$. Se $a > 1$, poiché

$$\log_b n = (\log_a n)(\log_b a) \quad e \quad \sum_{j=0}^{h-1} x^j = \frac{x^h - 1}{x - 1}$$

abbiamo

$$
\begin{array}{ll}
T(n) = d & se\ n \leq m \\
T(n) = aT(\frac{n}{b}) + hn^k & se\ n > m \\
 & \\
 & \\
T(n) \in \mathcal{O}(n^k) & se\ a < b^k \\
T(n) \in \mathcal{O}(n^k \log n) & se\ a = b^k \\
T(n) \in \mathcal{O}(n^{\log_b a}) & se\ a > b^k
\end{array}
$$

Figura 1: Relazioni del metodo divide et impera

$$
T(n) = d(a^{\log_a n})^{\log_b a} + c\frac{a^{\log_b n} - 1}{a - 1} = dn^{\log_b a} + c\frac{n^{\log_b a} - 1}{a - 1}
$$

e quindi la relazione di ricorrenza ha soluzione $\mathcal{O}(n^{\log_b a})$. Riassumendo,

$$
\begin{array}{ll}
T(n) \in \mathcal{O}(\log n) & se\ a = 1 \\
T(n) \in \mathcal{O}(n^{\log_b a}) & se\ a > 1
\end{array}
$$

Per esempio, per $a = 1$ e $b = 2$ abbiamo la relazione di ricorrenza della ricerca binaria (vedi cap 11), mentre per $a = b = 2$ abbiamo la visita su alberi binari bilanciati (vedi cap. 6). In questo caso, cioè quando $a = b$, la complessità è lineare, mentre per $a < b$ la complessità è sottolineare e per $b < a$ è sopralineare.

Ora consideriamo il caso in cui il lavoro di combinazione è lineare, cioè $c(n) = hn$ per qualche costante h:

$$
\begin{array}{ll}
T(n) = d & n \leq m \\
T(n) = aT(\frac{n}{b}) + hn & n > m
\end{array}
$$

Si può dimostrare il seguente risultato:

$$
\begin{array}{ll}
T(n) \in \mathcal{O}(n) & se\ a < b \\
T(n) \in \mathcal{O}(n \log n) & se\ a = b \\
T(n) \in \mathcal{O}(n^{\log_b a}) & se\ a > b
\end{array}
$$

Per $a = b = 2$ abbiamo la relazione di ricorrenza di `MergeSort`. Si noti che per $a < b$ il lavoro di combinazione lineare ha la prevalenza sul tempo di risoluzione dei sottoproblemi, mentre per $a > b$ questo è dominante: in questo caso la soluzione è uguale a quella del caso di tempo di combinazione costante. I due casi considerati sono casi particolari del caso più generale in cui $c(n)$ è polinomiale, mostrato in figura 5.1.

5.2 Un algoritmo di moltiplicazione

Finora abbiamo fatto l'assunzione che il tempo necessario ad eseguire le operazioni aritmetiche come somma, sottrazione e moltiplicazione sia costante. In realtà spesso queste operazioni, una volta che il programma in linguaggio ad alto livello viene tradotto in linguaggio macchina, non vengono eseguite in unico ciclo di clock, ma vengono tradotte in una sequenza di istruzioni macchina. Questo succede se i numeri interessati non sono rappresentabili con il numero di bit della parola di memoria. Inoltre può succedere che, anche anche lavorando in linguaggio ad alto livello, sia necessario eseguire operazioni con numeri maggiori del massimo numero definibile: in questo caso è necessario programmare una operazione aritmetica come sequenza di operazioni elementari. Se consideriamo numeri di lunghezza non limitata,

non è quindi corretta l'assunzione che il tempo di una operazione sia costante, ma si deve calcolare la complessità dell'algoritmo che implementa l'operazione in funzione del numero di cifre degli argomenti.

Per quanto riguarda l'addizione, se i numeri sono lunghi n cifre, la complessità è $\mathcal{O}(n)$ e l'algoritmo è quello classico di somma cifra per cifra. Per la moltiplicazione, l'algoritmo classico ha complessità $\mathcal{O}(n^2)$: infatti si eseguono n^2 operazioni di moltiplicazione tra singole cifre e una somma di n prodotti parziali, ciascuno di $2n$ cifre al più. Vediamo ora un algoritmo più veloce. Siano dati due numeri naturali A e B con n cifre e supponiamo per semplicità che n sia una potenza di 2. Chiamiamo A_{sin} e A_{des} rispettivamente le prime e ultime $\frac{n}{2}$ cifre di A e con e B_{sin} e B_{des} le prime e ultime $\frac{n}{2}$ cifre di B, cioè :

$$\begin{aligned} A &= A_{sin}10^{\frac{n}{2}} + A_{des} \\ B &= B_{sin}10^{\frac{n}{2}} + B_{des} \end{aligned}$$

Abbiamo:

$$AB = A_{sin}B_{sin}10^n + (A_{sin}B_{des} + A_{des}B_{sin})10^{\frac{n}{2}} + A_{des}B_{des}$$

Poiché

$$(A_{sin} + A_{des})(B_{sin} + B_{des}) = A_{sin}B_{sin} + A_{sin}B_{des} + A_{des}B_{sin} + A_{des}B_{des}$$

abbiamo:

$$AB = A_{sin}B_{sin}10^n + ((A_{sin} + A_{des})(B_{sin} + B_{des}) - A_{sin}B_{sin} - A_{des}B_{des})10^{\frac{n}{2}} + A_{des}B_{des}$$

Quindi abbiamo ridotto la moltiplicazione fra A e B a tre moltiplicazioni fra numeri di $\frac{n}{2}$ cifre più alcune addizioni, sottrazioni e traslazioni (per moltiplicare per $10^{\frac{n}{2}}$ e 10^n). Un algoritmo può essere il seguente, dove n indica il numero di cifre da moltiplicare (l'algoritmo è descritto parzialmente in modo informale):

```
int mult(int* A, int* B, int n) {
   if (n==1) return A[0]*B[0];
   int * Asin =new int [n/2];
   int * Ades =new int [n/2];
   int * Bsin =new int [n/2];
   int * Bsin =new int [n/2];
   int * Asd =new int [n/2+1];
   int * Bsd =new int [n/2+1];

   // ¡riempi Ades e Bdes con le parti meno significative di A e B, risp.¿
   // ¡riempi Asin e Bsin con le parti più significative di A e B, risp.¿
   // ¡riempi Asd con Asin+Ades e cosí per Bsd¿

   int x= mult(Asin, Bsin, n/2);
   int y= mult(Ades, Bdes, n/2);
   int z= mult(Asd, Bsd, n/2+1) - x - y;
   return x*pow(10,n) + z*pow(10,n/2) + y;
}
```

La relazione di ricorrenza per il tempo di esecuzione dell'algoritmo è la seguente:

$$\begin{aligned} T(1) &= d \\ T(n) &= 3T(n/2) + cn \end{aligned}$$

che implica che $T(n)$ ha complessità $\mathcal{O}(n^{\log_2 3})$, cioè circa $\mathcal{O}(n^{1.59})$, che è un risultato migliore di $\mathcal{O}(n^2)$. Esistono anche algoritmi ancora più efficienti per la moltiplicazione.

5.3 Relazioni di ricorrenza lineari

Sia data la relazione di ricorrenza:

$$T(0) = d$$
$$T(n) = aT(n-1) + g(n) \quad n > 0$$

Distinguiamo due casi: $a = 1$ e $a > 1$ e consideriamo prima il caso $a = 1$, cioè :

$$T(1) = d$$
$$T(n) = T(n-1) + g(n) \quad n > 0$$

Si vede facilmente che

$$T(n) = d + g(1) + .. + g(n) \tag{1}$$

Vediamo la soluzione per alcune funzioni $g(n)$. Se $g(n)$ è costante, cioè

$$T(0) = d$$
$$T(n) = T(n-1) + b \quad n > 0$$

abbiamo la relazione di ricorrenza della funzione `fatt` che calcola il fattoriale di un numero, con soluzione $T(n) = d + bn$, che quindi è $\mathcal{O}(n)$. Si può dimostrare che, per $g(n) = bn^k$, $T(n)$ è $\mathcal{O}(n^{k+1})$. Infatti, consideriamo

$$T(0) = d$$
$$T(n) = T(n-1) + bn^k \quad n > 0$$

Per la (1) abbiamo, per $n \geq 1$:

$$T(n) = d+g(1)+..+g(n) = d+b1^k+..bn^k = d+b(1^k+..+n^k) \leq d+b(n^k+..+n^k) = d+bnn^k \in \mathcal{O}(n^{k+1})$$

Nel caso di $a > 1$, si può dimostrare che $T(n) \in \mathcal{O}(a^n)$. Questo vuol dire che la complessità in questi casi è esponenziale. Le relazioni considerate sono casi particolari di relazioni di ricorrenza generali, dette lineari di ordine k:

$$T(n) = d \qquad n \leq k$$
$$T(n) = a_1T(n-1) + a_2T(n-2) + .. + a_kT(n-k) + b \quad n > k$$

Le relazioni di ricorrenza lineari hanno in genere soluzioni di ordine esponenziale, tranne che quando esiste al più un solo $a_i \neq 0$ e inoltre $a_i = 1$. C'è una grande differenza fra un tempo polinomiale e uno esponenziale, per cui in generale gli algoritmi esponenziali sono da scartare, se ovviamente se ne riesce a definirne uno polinomiale per risolvere il problema, come nel caso del prossimo esempio. Purtroppo, per molti problemi finora non si è riusciti a dare un algoritmo di risoluzione non esponenziale, anche se non esiste a tutt'oggi una dimostrazione formale del fatto che non si possa riuscirvi (vedi cap. 14).

Esempio 5.1
Definiamo una funzione ricorsiva che calcola l'ennesimo numero di Fibonacci, dove i numeri di Fibonacci sono definiti induttivamente nel modo seguente:

$$f_0 = 0, f_1 = 1,$$
$$f_n = f_{n-1} + f_{n-2} \quad \textit{per } n > 1$$

La funzione `fibonacci` *restituisce l'n-esimo termine della serie di Fibonacci.*

```
int fibonacci(int n) {
```

```
    if (n <= 1)  return n;
    return (fibonacci(n-1)+fibonacci(n-2));
}
```

La relazione di ricorrenza è :

$$\begin{aligned} T(n) &= d && n \leq 1 \\ T(n) &= b + T(n-1) + T(n-2) && n > 2 \end{aligned}$$

con soluzione esponenziale. In questo caso, esiste un algoritmo iterativo che risolve lo stesso problema in tempo lineare:

```
int fibonacci(int n) {
    int k; int j=0; int f=1;
    for (int i=1; i<=n; i++) { k=j; j=f; f=k+j; }
    return j;
}
```

Un algoritmo ricorsivo con complessità lineare, che però non ricalca la definizione induttiva della serie di Fibonacci, è il seguente:

```
int fibonacci(int n, int a=0, int b=1) {
    if (n == 0) return a;
    return fibonacci(n-1, b, a+b);
}
```

Esempio 5.2
Tutte le funzioni ricorsive su liste definite nella sezione 3 hanno la stessa relazione di ricorrenza del fattoriale e quindi sono lineari nella lunghezza della lista. La seguente funzione calcola l'inversa di una lista utilizzando `append`.

```
void reverse(Elem* &l) {
    if (! l) return;
    Elem* p=l; l=l->next; p->next=NULL; append(reverse(l), p);
}
```

La relazione di ricorrenza di `reverse` *è*

$$\begin{aligned} T(0) &= a \\ T(n) &= bn + T(n-1) \quad n > 1 \end{aligned}$$

e quindi la complessità è $\mathcal{O}(n^2)$. Una funzione più efficiente è la seguente, che ha complessità lineare.

```
void quickReverse(Elem* &l1, Elem* l2=NULL) {
    if (! l1) l1=l2;
    else {
        Elem *p=l1; l1=l1->next; p->next=l2;
        quickReverse(l1, p);
    }
}
```

6 Alberi binari

6.1 Definizione e visite

Un *albero binario* è un insieme di nodi; se l'insieme non è vuoto, un nodo è chiamato *radice* e i restanti sono suddivisi in due sottoinsiemi disgiunti che sono a loro volta alberi binari e sono detti *sottoalberi sinistro e destro* della radice. Gli alberi binari possono essere definiti induttivamente nel modo seguente:

Definizione 6.1 (albero binario)

- *l'insieme vuoto di nodi è un albero binario;*
- *un nodo p più due alberi B_s e B_d forma un albero, di cui p è radice, B_s è il* sottoalbero sinistro *e B_d il* sottoalbero destro *di p.*

Dato un albero binario non vuoto con radice p e sottoalberi B_s, e B_d,

- p è *padre* della radice di B_s e della radice di B_d;
- la radice di B_s (B_d) è il *figlio sinistro (destro)* di p;
- p è *antecedente* di ogni nodo dell'albero, tranne p stesso;
- un nodo con entrambi i sottoalberi vuoti è una *foglia*;
- ogni nodo dell'albero, tranne p, è *discendente* di p;
- il *livello di un nodo* è il numero dei suoi antecedenti;
- il *livello dell'albero* è il massimo fra i livelli dei suoi nodi;

Un albero può essere rappresentato con un grafo avente per nodi i nodi dell'albero e un arco che congiunge ogni figlio al relativo padre, orientato dal padre al figlio. In genere si adotta la convenzione che i figli sinistro e destro di ogni nodo sono congiunti al padre mediante un arco inclinato rispettivamente verso sinistra e destra. La figura 2 mostra un albero binario. Quindi il livello di un nodo è dato dalla lunghezza del cammino fra la radice e il nodo; per esempio, il livello della radice è 0 (assumiamo che il livello di un albero vuoto sia -1). Il livello dell'albero è il più lungo cammino fra la radice e una foglia. Un albero binario *etichettato* è un albero binario in cui ad ogni nodo è associato un nome, o etichetta. D'ora in poi considereremo soltanto alberi binari etichettati.

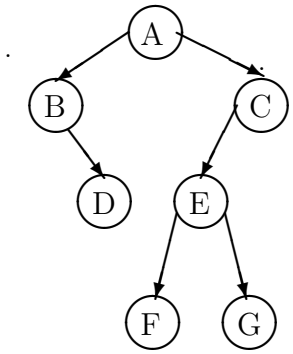


Figura 2: Esempio di albero binario

La definizione induttiva degli alberi binari data sopra permette di fare prove di proprietà degli alberi binari e di correttezza degli algoritmi che lavorano su alberi binari con l'*induzione strutturale*, che è un

caso particolare di induzione ben fondata (vedi **Appendice 1**) . L'insieme degli alberi binari può essere visto come un insieme ben fondato prendendo come relazione di ordinamento la relazione di sottoalbero (sottostruttura) fra due alberi. Il minimo è l'albero vuoto ed è quindi la base dell'induzione. Il passo induttivo suppone vera la proprietà su B_s, e B_d, e dimostra che è vera per un nodo con sottoalberi sinistro e destro rispettivamente B_s e B_d.

Esempio 6.1 (Induzione strutturale su alberi binari)
Dimostriamo che in un albero binario il numero dei sottoalberi vuoti è uguale al numero dei nodi più 1. Dato un albero B, indichiamo con $N(B)$ il numero dei suoi nodi e con con $V(B)$ il numero di sottoalberi vuoti di B. Vogliamo dimostrare che, per ogni B, $V(B) = N(B) + 1$.

- *Base. Se l'albero è vuoto $N(B) = 0$, $V(B) = 1$ e la proprietà è verificata.*
- *Induzione. Consideriamo un albero B con radice p e sottoalberi sinistro e destro B_s e B_d. Ipotesi: $V(B_s) = N(B_s) + 1$ e $V(B_d) = N(B_d) + 1$.*
  Tesi: $V(B) = N(B) + 1$.
  Dim. .
  $$\begin{aligned} V(B) &= V(B_s) + V(B_d) && \text{perchè } B \text{ non è vuoto} \\ &= N(B_s) + 1 + N(B_d) + 1 && \text{per ipotesi induttiva} \\ &= N(B) + 1 && \text{perchè } N(B) = N(B_s) + N(B_d) + 1 \end{aligned}$$

Le operazioni più comuni sugli alberi sono quelle di linearizzazione, ricerca, inserimento, e cancellazione di nodi. Una linearizzazione di un albero è una sequenza contenente i nomi dei suoi nodi. Esistono diversi tipi di linearizzazione. Le più comuni linearizzazioni, dette *visite*, degli alberi binari sono tre: *ordine anticipato (preorder)*, *ordine differito (postorder)* e *ordine simmetrico (inorder)*, definite nel modo seguente:

ordine anticipato:

```
{
   se l'albero binario non e' vuoto {
      esamina la radice;
      visita il sottoalbero sinistro;
      visita il sottoalbero destro;
   }
}
```

ordine differito:

```
{
   se l'albero binario non e' vuoto {
      visita il sottoalbero sinistro;
      visita il sottoalbero destro;
      esamina la radice;
   }
}
```

ordine simmetrico:

```
{
   se l'albero binario non e' vuoto {
      visita il sottoalbero sinistro;
      esamina la radice;
      visita il sottoalbero destro;
   }
}
```

L'albero binario di figura 2 è linearizzato secondo i tre ordini nel modo seguente:

- ordine anticipato: $ABDCEFG$
- ordine differito: $DBFGECA$
- ordine simmetrico: $BDAFEGC$

Per memorizzare un albero binario può essere utilizzata una lista multipla: ogni nodo è rappresentato con una struttura con un campo informazione e due puntatori, di cui il primo punta al sottoalbero sinistro, mentre il secondo punta al sottoalbero destro. L'albero vuoto corrisponde al valore `NULL`:

```
struct Node {
   LabelType label;
   Node* left;
   Node* right;
};
```

Il risultato della memorizzazione dell'albero binario di figura 2 è mostrata in figura 3.

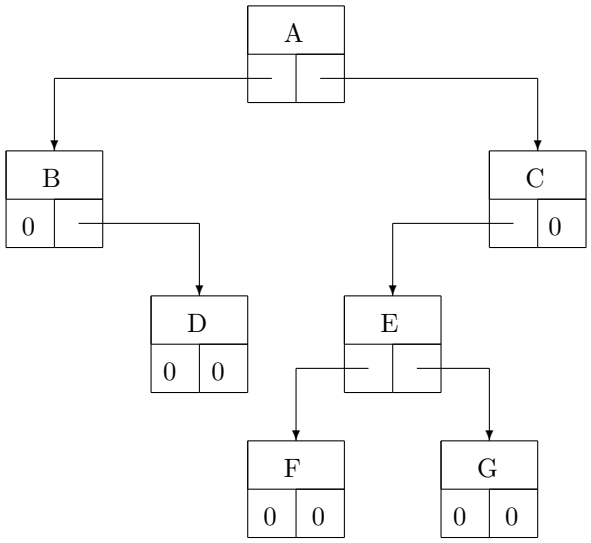


Figura 3: Memorizzazione dell'albero in Figura 2

In certi casi particolari, come vedremo, un albero binario può essere memorizzato anche in un array. Con il tipo di memorizzazione con lista multipla, le procedure di visita per alberi binari possono essere definite nel modo seguente:

```
void preOrder(Node* tree) {
   if (tree) {
```

```
        <esamina tree->label>
        preOrder(tree->left);
        preOrder(tree->right);
    }
}
```

```
void postOrder(Node* tree) {
    if (tree) {
        postOrder(tree->left);
        postOrder(tree->right);
        <esamina tree->label>;
    }
}
```

```
void inOrder(Node* tree) {
    if (tree) {
        inOrder(tree->left);
        <esamina tree->label>;
        inOrder(tree->right);
    }
}
```

Per quanto riguarda il tempo di esecuzione, in genere la sua complessità è valutata in funzione del numero n di nodi dell'albero. Supponiamo per semplicità che il tempo per l'esame del nodo sia costante. La relazione di ricorrenza per le tre visite è la seguente, dove con n_s e n_d indichiamo rispettivamente il numero di nodi del sottoalbero sinistro e destro della radice:

$$\begin{aligned} T(0) &= a \\ T(n) &= b + T(n_s) + T(n_d) \text{ con } n_s + n_d = n - 1 \quad n > 0 \end{aligned}$$

Si noti che con a abbiamo indicato il tempo per il test `(tree)` e con b il tempo per l'esame del nodo.

Dimostriamo con l'induzione strutturale the $T(n) = bn + a(n+1)$.
Base. Se l'abero è vuoto, $T(0) = a$ e la formula è verificata.
Induzione. Ipotesi: $T(n_s) = bn_s + a(n_s + 1)$ e $T(n_d) = bn_d + a(n_d + 1)$
Tesi: $T(n) = bn + a(n+1)$.
Dim.

$$\begin{aligned} T(n) &= b + T(n_s) + T(n_d) && \text{per la relazione di ricorrenza} \\ &= b + bn_s + a(n_s + 1) + bn_d + a(n_d + 1) && \text{per ipotesi induttiva} \\ &= b(1 + n_s + n_d) + a(n_s + n_d + 2) \\ &= bn + a(n+1) && \text{perchè } n_s + n_d + 1 = n \end{aligned}$$

Abbiamo quindi che $T(n)$ è lineare nel numero dei nodi. Diamo ora alcune definizioni che ci saranno utili nel seguito.

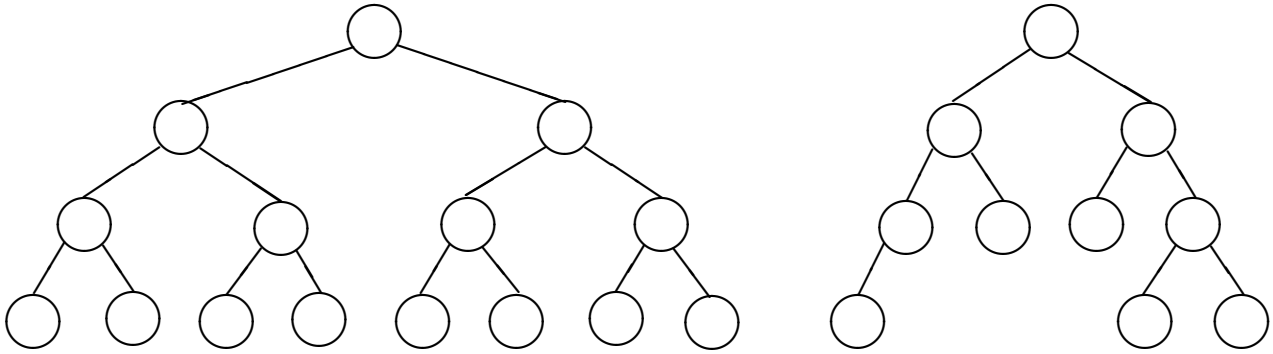
Definizione 6.2 (albero binario bilanciato)

- *Un albero binario* bilanciato *è un albero binario tale che i nodi di tutti i livelli tranne quelli dell'ultimo hanno due figli.*

- *Un albero binario* quasi bilanciato *è un albero binario che fino al penultimo livello è un albero bilanciato.*

Definizione 6.3 (albero binario pieno)
Un albero binario pienamente binario *è un albero binario in cui tutti i nodi tranne le foglie hanno due figli.*

In un albero binario bilanciato tutte le foglie sono all'ultimo livello e il sottoalbero sinistro e destro di ogni nodo contengono lo stesso numero di nodi. In un albero binario quasi bilanciato le lunghezze di due cammini dalla radice a due foglie differiscono di al più 1: le foglie si trovano tutte all'ultimo e penultimo livello. Un albero bilanciato è un caso particolare di albero quasi bilanciato. Le figure 4a), 4b) e 5 mostrano rispettivamente un albero binario bilanciato, un albero binario quasi bilanciato e un albero pienamente binario. Valgono i seguenti teoremi.



(a) (b)

Figura 4: Un albero bilanciato e uno quasi bilanciato

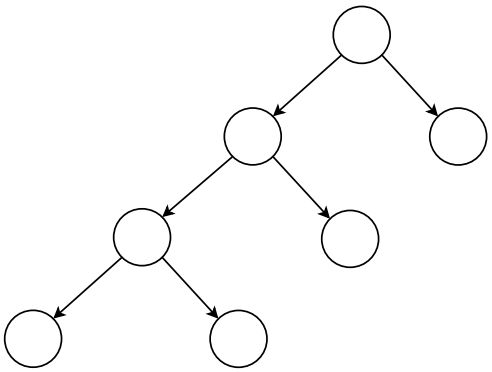


Figura 5: Un albero pienamente binario

Proposizione 6.1
Un albero binario bilanciato non vuoto di livello k ha $2^{(k+1)} - 1$ nodi e 2^k foglie.

Proposizione 6.2
Un albero binario quasi bilanciato non vuoto di livello k ha un numero di nodi compreso fra 2^k *e* $(2^{(k+1)}-1)$ *e un numero di foglie compreso fra* $2^{(k-1)}$ *e* 2^k.

Proposizione 6.3
In un albero binario pienamente binario il numero di nodi interni è uguale al numero di foglie meno 1.

6.2 Esempi di programmi su alberi binari

Essendo gli alberi binari una struttura dati ricorsiva, analogamente alle liste, è conveniente progettare i programmi che lavorano su alberi binari facendo corrispondere il più possibile la struttura del programma ricorsivo a quella della definizione, come è stato fatto per le liste: i casi non ricorsivi comprenderanno l'albero binario vuoto, gli altri un nodo con due sottoalberi. Le seguenti funzioni contano rispettivamente il numero di nodi e il numero di foglie in unalbero binario:

```
int nodes (Node* tree) {
   if (!tree) return 0;
   return 1+nodes(tree->left)+nodes(tree->right);
}
```

```
int leaves (Node* tree) {
   if (!tree) return 0;
   if ( !tree->left && !tree->right ) return 1;
   return leaves(tree->left)+leaves(tree->right);
}
```

La funzione seguente cerca un'etichetta in un albero binario e restituisce il puntatore al nodo che la contiene:

```
Node* findNode (LabelType n, Node *tree) {
   if (!tree) return NULL;
   if (tree->label == n) return tree;
   Node* a=findNode(n, tree->left);
   if (a) return a;
   else return findNode(n, tree->right);
}
```

La funzione seguente cancella un albero binario restituendo un puntatore nullo:

```
void delTree(Node* &tree) {
   if (tree) {
      delTree(tree->left);
      delTree(tree->right);
      delete tree; tree=NULL;
   }
}
```

La funzione seguente inserisce un nuovo nodo in un albero binario. La funzione restituisce 1 se l'inserimento ha avuto successo e 0 altrimenti. La funzione inserisce una foglia con etichetta `son` come figlio del nodo con etichetta `father`, in modo che sia il figlio sinistro o destro a seconda che c='l' o c='r'; se `father` non compare nell'albero o ha gia' un figlio in quella posizione, non modifica l'albero. Se l'albero e' vuoto, inserisce `son` come radice.

```
int insert(Node*& root, LabelType son, LabelType father, char c) {
   if (!root) {
      root=new Node;
      root->label=son; root->left = root->right = NULL;
      return 1;
   }
   Node* a=findNode(father,root);
   if (!a) return 0;
   if (c=='l' && !a->left) {
      a->left=new Node;
      a->left->label=son; a->left->left = a->left->right = NULL;
      return 1;
   }
   if (c=='r' && !a->right) {
      a->right=new Node;
      a->right->label=son; a->right->left = a->right->right = NULL;
      return 1;
   }
   return 0;
}
```

In **Appendice 3** si riporta una classe C++ che implementa un albero binario e che, nei suoi metodi, utilizza le funzioni presentate in questa sezione.

Come altro esempio di programma ricorsivo su alberi binari, consideriamo il problema di contare il numero di nodi che hanno un ugual numero di discendenti nel sottoalbero sinistro e destro. La seguente funzione restituisce tale numero. Si noti che, per avere una complessità O(n), la funzione restituisce il numero di nodi richiesto ma anche, utilizzando il parametro `nodes` passato per riferimento, conta il numero di nodi complessivo dell'albero.

```
int counter(Node* tree, int & nodes) {
   if (!tree) { nodes =0; return 0; }
   int nodes_s, nodes _d;
   int c_s=counter(tree->left, nodes_s);
   int c_d=counter(tree->right, nodes_d);
   nodes = nodes_s+ nodes_d+1;
   return c_s+c_d+(nodes_s == nodes_d);
}
```

7 Alberi generici

7.1 Definizione e visite

Un *albero generico* è un insieme non vuoto di nodi, di cui uno è chiamato radice, e i restanti sono suddivisi in sottoinsiemi disgiunti che sono a loro volta alberi e sono detti sottoalberi della radice:

Definizione 7.1 (albero generico)

- *Un nodo p è un albero ed è anche la radice dell'albero;*
- *un nodo p più un insieme di alberi* $\{A_1, \ldots, A_n\}, n \geq 1$ *è un albero, di cui p è radice, mentre* $A_1, \ldots, A_n$ *sono detti primo, secondo ,.. n-esimo sottoalbero di p.*

Dato un albero con radice p e sottoalberi $A_1, \ldots, A_n$,

- p è *padre* della radice di ogni A_i;
- la radice di ogni A_i è l'i-esimo *figlio* di p;
- un nodo che non ha sottoalberi è una *foglia*;

Si noti che in un albero generico l'ordine tra i figli di ciascun nodo è significativo. Le definizioni di antecedente, discendente e livello sono uguali a quelle per gli alberi binari. Da ora in poi considereremo solo alberi generici etichettati. Un esempio di albero generico è mostrato in figura 6.

Anche le proprietà degli alberi generici e dei programmi che li manipolano possono essere dimostrati con l'induzione strutturale:i casi base sono gli alberi composti da un solo nodo; Il passo induttivo suppone vera la proprietà sugli alberi $A_1, \ldots, A_n$ e la dimostra vera sull'albero composto da un nodo e $A_1, \ldots, A_n$ come sottoalberi.

Esempio 7.1 (induzione strutturale su alberi generici)
Dimostriamo che in un albero il numero dei nodi è uguale ad uno più la somma dei gradi *dei nodi, dove il grado di un nodo è il numero dei suoi figli. Dato un albero A, indichiamo con $N(A)$ il numero dei suoi nodi, con $G(A)$ la somma dei gradi dei suoi nodi e con $G(p)$ il grado di un nodo p. Vogliamo dimostrare che, per ogni A, $N(A) = G(A) + 1$.*

- *Base. Se l'albero è formato da un solo nodo, questo ha grado* 0 *e quindi la proprietà è verificata.*
- *Induzione. Consideriamo un albero A con radice p e sottoalberi della radice $A_1, \ldots, A_n$. Ipotesi: Per ogni $1 \leq i \leq n$, $N(A_i) = G(A_i) + 1$.*
  Tesi: $N(A) = G(A) + 1$.
  Dim.
  $$\begin{aligned} N(A) &= N(A_1) + \ldots + N(A_n) + 1 \\ &= G(A_1) + 1 + \ldots + G(A_n) + 1 + 1 && \textit{per ipotesi induttiva} \\ &= G(A_1) + \ldots + G(A_n) + n + 1 \\ &= G(A_1) + \ldots + G(A_n) + G(p) + 1 && \textit{perchè } G(p) = n \\ &= G(A) + 1 && \textit{per definizione di grado} \end{aligned}$$

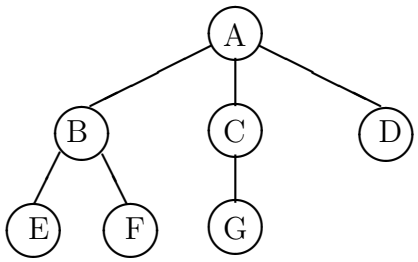


Figura 6: Esempio di albero

È importante notare come un albero binario non sia un caso particolare di albero generico, ma un tipo di dato strutturalmente diverso. Questo, per esempio, vuol dire che con due nodi A e B possono essere costruiti due alberi generici differenti, mostrati in figura 7.a), mentre possono essere costruiti quattro

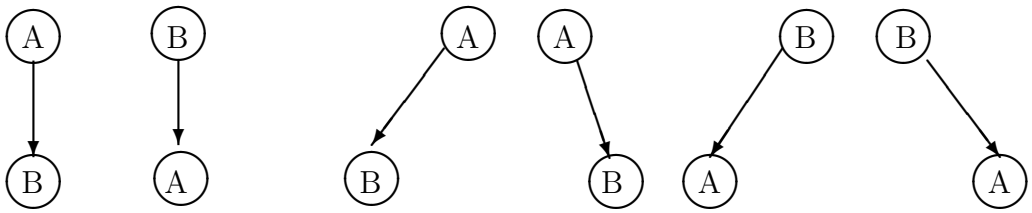


Figura 7: Alberi ordinati e binari con due nodi

alberi binari differenti, come mostrato in figura 7.b).

Le linearizzazioni di un albero generico sono quella anticipata e quella differita: La linearizzazione secondo l'ordine anticipato è ottenuta secondo il seguente algoritmo di *visita*:

```
{
   esamina la radice;
   se la radice ha sottoalberi {
      visita il primo sottoalbero;
      ...
      visita l'ennesimo sottoalbero;
   }
}
```

mentre quella secondo l'ordine *differito* è ottenuta nel modo seguente:

```
{
   se la radice ha sottoalberi {
      visita il primo sottoalbero;
      ...
      visita l'ennesimo sottoalbero;
   }
   esamina la radice;
}
```

Le linearizzazioni dell'albero in figura 6 rispettivamente secondo l'ordine anticipato e differito sono $ABEFCGD$ e $EFBGCDA$. Per memorizzare un albero si possono utilizzare strutture con puntatori (liste multiple). Un primo tipo di memorizzazione è quella in cui ogni nodo è memorizzato con una struttura che ha un campo che contiene l'etichetta del nodo e k altri campi di tipo puntatore a nodo, se k è il numero massimo di figli di un nodo, o un array di puntatori a nodo.

Un altro metodo di memorizzazione, che prescinde dal numero di figli dei nodi è quello detto *figlio-fratello*, in cui un nodo è una struttura con tre campi: il primo campo contiene il nome del nodo, il secondo il puntatore al primo figlio del nodo, se esiste, e il terzo il puntatore al primo fratello nell'ordinamento, se esiste. Se il figlio o il fratello non esistono, il puntatore relativo è vuoto. Allora può essere utilizzata la stessa truttura definita per gli alberi binari. La memorizzazione dell'albero di figura 6 con i due metodi è mostrata in figura 8 e 9 rispettivamente.

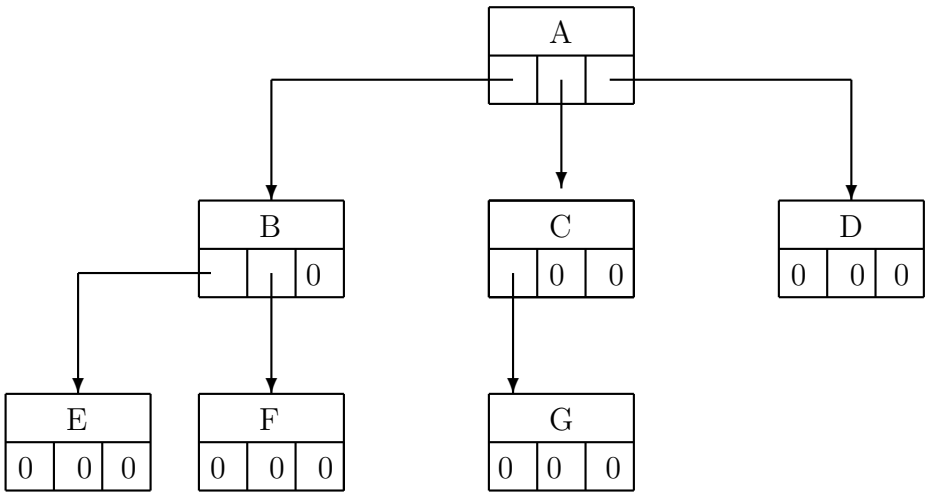


Figura 8: La memorizzazione M1 dell'albero in figura 6

Riconsideriamo l'albero ordinato di figura 6 e la sua memorizzazione figlio-fratello, mostrata in figura 9. È facile rendersi conto che, applicando `preorder` otteniamo la linearizzazione dell'albero secondo l'ordine anticipato, anche se la funzione non è definita con gli stessi casi della definizione di albero generico (`preorder` ha come caso base l'albero vuoto, che non è un albero generico, mentre il caso base della definizione è l'albero composto da un solo nodo). Per ottenere la linearizzazione differita dell'albero, è facile rendersi conto che possiamo utilizzare la visita simmetrica. Infatti, adottando la memorizzazione figlio-fratello, è come se avessimo "trasformato" l'albero in un albero binario, con la proprietà che

- la visita differita dell'albero corrisponde alla visita simmetrica del trasformato.

Quindi il tempo delle visite in un albero generico è lineare nel numero dei nodi. Per quanto riguarda la ricerca, l'inserimento e la cancellazione di un nodo, il tempo è comunque lineare. Infatti queste operazioni possono essere programmate mantenendo la struttura delle visite.

Supponiamo ora di voler calcolare la complessità delle visite prendendo come dimensione i livelli dell'albero. Abbiamo

$$\begin{aligned} T(0) &= a \\ T(n) &= 2T(n-1)+b \quad n>1 \end{aligned}$$

Infatti se l'albero ha livello 0, cioè è composto dalla sola radice, il tempo è costante: se l'albero ha n livelli, il tempo è un tempo costante per la visita della radice più il tempo di visita di due sottoalberi, ognuno con al massimo $n-1$ livelli. Se risolviamo la relazione, vediamo che $T(n)$ è $\mathcal{O}(2^n)$, che è un tempo esponenziale (la relazione di ricorrenza è lineare).

7.2 Esempi di programmi su alberi generici

Consideriamo la memorizzazione figlio-fratello. La funzione seguente conta le foglie di un albero generico. Si noti la differenza con la funzione, definita nella sezione 6.2 che conta le foglie di un albero binario: mentre in un albero binario una foglia ha entrambi i sottoalberi vuoti, in un albero generico, se un nodo ha il campo `left` a NULL, allora non ha nessun figlio, cioè è una foglia.

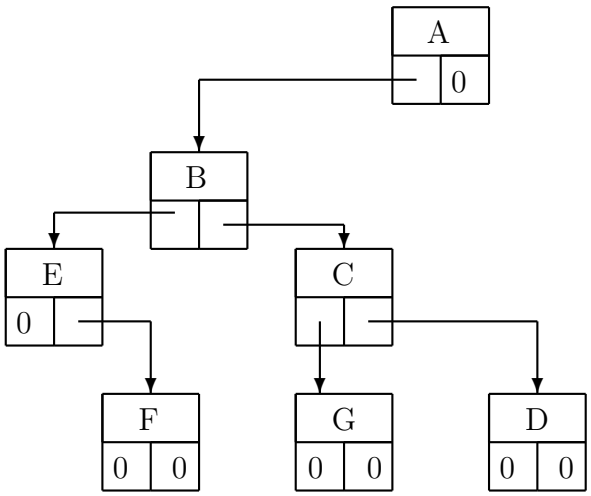


Figura 9: La memorizzazione M2 dell'albero in figura 6

```
int leaves(Node* tree) {
   if (!tree) return 0;
   if (!tree->left) return 1+ leaves(tree->right);
   return leaves(tree->left)+ leaves(tree->right);
}
```

La seguente funzione `insert` aggiunge un figlio come ultimo figlio di un nodo; la funzione chiama la funzione `findNode` definita nella sezione 6.2 e la funzione `addSon` che inserisce un nodo con etichetta `son` in fondo alla lista dei figli di `father`. La funzione `insert`, se l'albero è vuoto, inserisce il nodo come radice.

```
void addSon(LabelType x, Node* &tree) {
   if (!tree) {
      tree=new Node;
      tree->label=x; tree->left = tree->right = NULL;
   }
   else addSon(x, tree->right);
}

int insert(LabelType son, LabelType father, Node* &root) {
   if (!root) {
      root=new Node;
      root->label=son; root->left = root->right = NULL;
      return 1;
   }
   Node* a=findNode(father, root);
   if (!a) return 0;
   addSon(son, a->left);
   return 1;
}
```

8 Alberi binari di ricerca

Definizione 8.1 (albero binario di ricerca)
Un albero binario di ricerca è un albero binario etichettato tale che sulle etichette dei nodi è stabilita una relazione di ordinamento e per ogni nodo p vale la seguente proprietà:

- *tutti i nodi del sottoalbero sinistro di p hanno etichette minori di quella di p e tutti i nodi del sottoalbero destro di p hanno etichette maggiori di quella di p.*

Da questa proprietà segue che i nodi di un albero binario di ricerca hanno tutti etichette diverse. Un esempio di albero binario di ricerca avente per etichette numeri interi è mostrato in figura 10.

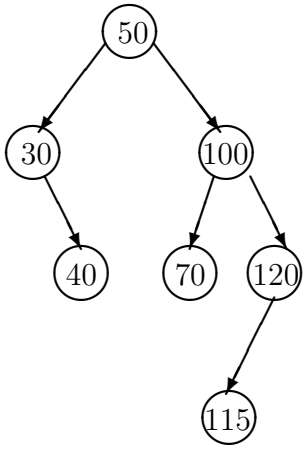


Figura 10: Esempio di albero binario di ricerca

Le operazioni su di un albero binario di ricerca sono, data una etichetta, la ricerca, l'inserimento e la cancellazione di un nodo avente quella etichetta. Queste operazioni sono più efficienti su di un albero binario di ricerca che su di un albero binario, a causa del fatto che è possibile sfruttare la proprietà descritta sopra che lega le etichette dei nodi. Infatti, per cercare un elemento in un albero binario di ricerca, possiamo seguire il seguente algoritmo: se l'albero è vuoto, allora l'elemento ovviamente non appartiene all'albero. Se c'è almeno un nodo, confrontiamo l'elemento cercato con (l'etichetta di) quel nodo. Se sono uguali, abbiamo finito. Altrimenti cerchiamo ricorsivamente nel sottoalbero sinistro o destro a seconda che l'elemento sia rispettivamente minore o maggiore del nodo. Siamo sicuri che possiamo scartare l'altro sottoalbero, perchè questo contiene soltanto elementi rispettivamente maggiori o minori della radice. Alla fine, o troviamo il nodo, o un albero vuoto. La ricerca può essere definita nel modo seguente (restituisce il puntatore al nodo che ha come etichetta l'elemento cercato):

```
Node* findNode (LabelType n, Node* tree) {
   if (!tree) return 0;
   if (n == tree->label) return tree;
   if (n < tree->label) return findNode(n, tree->left);
   return findNode(n, tree->right);
}
```

Dimostriamo con l'induzione strutturale che la funzione ricerca è corretta, cioè termina sempre e dà come risultato 1 se il nodo cercato si trova nell'albero e 0 altrimenti.
Base Se l'albero è vuoto l'elemento non può essere nell'albero e la funzione infatti restituisce 0.

Induzione. Ipotesi: La funzione è corretta se applicata agli alberi B_s e B_d.
Tesi: La funzione è corretta se applicata all'albero di radice p con sottoalberi sinistro e destro B_s e B_d.
Dim. Ci sono tre casi da considerare: (1) l'elemento è uguale alla radice; (2) l'elemento è più piccolo della radice; (3) l'elemento è più grande della radice. Nel primo caso la funzione restituisce correttamente il puntatore alla radice. Nel secondo caso, data la proprietà degli alberi binari di ricerca, l'elemento è nell'albero se e solo se è nel sottolabero sinistro. La funzione in questo caso fa una chiamata ricorsiva sul sottolabero sinistro che dà il risultato corretto per ipotesi induttiva. Il terzo caso è analogo, tranne che si considera il sottoalbero destro.

Per inserire un elemento il procedimento è analogo: partendo dalla radice, si scende nell'albero scegliendo di andare a destra o a sinistra a seconda che l'elemento da inserire sia più grande o più piccolo del nodo che stiamo esaminando. Non appena si incontra un albero vuoto, si può inserire l'elemento: se l'albero vuoto cui siamo arrivati è il sottoalbero sinistro di un nodo p, allora l'elemento può essere inserito come figlio sinistro di p; se invece è il sottoalbero destro di un nodo q, allora l'elemento può essere inserito come figlio destro di q. Notare che la seguente funzione non effettua alcun inserimento se l'elemento è già presente nell'albero.

```
void insertNode (LabelType n, Node* &tree) {
   if (!tree) {
      tree=new Node;
      tree->label=n; tree->left = tree->right = NULL;
      return;
   }
   if (n < tree->label) insertNode (n, tree->left);
   if (n > tree->label) insertNode (n, tree->right);
}
```

Per quanto riguarda la cancellazione, le cose sono un po' più complicate. Infatti è necessario fare qualche operazione in più per mantenere la proprietà dell'albero binario di ricerca. Un possibile algoritmo è il seguente: prima si cerca il nodo da cancellare effettuando una ricerca come negli algoritmi precedenti. Se il nodo viene trovato, sia esso p, possono verificarsi due situazioni diverse. Se p ha un sottoalbero vuoto, il padre di p viene connesso all'unico sottoalbero non vuoto di p , cioè l'unico sottoalbero di p diventa sottoalbero sinistro o destro del padre di p , a seconda che p sia figlio sinistro o destro. Se p ha entrambi i sottoalberi non vuoti, una possibile scelta è quella di cercare il nodo con etichetta minore nel sottoalbero destro di p, cancellarlo e mettere la sua etichetta come etichetta di p. Il procedimento è corretto perchè questa etichetta è senz'altro maggiore di tutte quelle nel sottoalbero sinistro di p (infatti è maggiore dell'etichetta di p) e minore di tutte quelle del sottoalbero destro. L'algoritmo può essere definito nel modo seguente, dove `deleteNode` è una procedura che effettua la cancellazione e `deleteMin`, chiamata da `deleteNode`, esegue la ricerca della minima etichetta in un albero binario di ricerca e la sua cancellazione, restituendo in uscita il suo valore.

```
void deleteMin (Node* &tree, LabelType &m) {
   if (tree->left) deleteMin(tree->left, m);
   else {
      m=tree->label;
      Node* a=tree;
      tree=tree->right;
      delete a;
   }
}
```

```
void deleteNode(LabelType n, Node* &tree) {
   if (tree)
      if (n < tree->label) deleteNode(n, tree->left);
      else if (n > tree->label) deleteNode(n, tree->right);
      else if (!tree->left) {
         Node* a=tree; tree=tree->right; delete a;
      }
      else if (!tree->right) {
         Node* a=tree; tree=tree->left; delete a;
      }
      else deleteMin(tree->right, tree->label);
}
```

La relazione di ricorrenza per le operazioni di ricerca e inserimento sull'albero binario di ricerca è :

$$\begin{aligned} T(0) &= a \\ T(n) &= b + T(k) \text{ con } k < n \end{aligned}$$

In generale la soluzione dipende dal livello del nodo che si vuole cercare, inserire o cancellare. Il caso peggiore è quando k è sempre uguale a $n-1$, per cui abbiamo la relazione

$$\begin{aligned} T(0) &= a \\ T(n) &= b + T(n-1) \end{aligned}$$

e $T(n)$ ha complessità lineare ($T(n) = a + bn$): questo si verifica se l'albero è "degenere", cioè è costituto da un solo cammino, per esempio che va sempre a destra o sempre a sinistra o a destra e sinistra alternativamente. Il caso migliore è quando l'albero è bilanciato o quasi bilanciato, per cui k è sempre uguale a circa $\frac{n}{2}$, nel qual caso la relazione di ricorrenza è :

$$\begin{aligned} T(0) &= a \\ T(n) &= b + T(\tfrac{n}{2}) \end{aligned}$$

per cui $T(n) \in \mathcal{O}(\log n)$. In generale, il tempo di esecuzione sarà compreso fra $\mathcal{O}(n)$ e $\mathcal{O}(\log n)$, mentre il tempo medio è $\mathcal{O}(\log n)$. Per la cancellazione, abbiamo un tempo $\mathcal{O}(\log n)$ per trovare il nodo da cancellare, più un tempo $\mathcal{O}(\log n)$ per trovare il nodo da sostituire al posto di quello cancellato. Quindi il tempo è comunque logaritmico.

Come ultima osservazione, si noti che vale la seguente proprietà, dimostrabile con induzione strutturale:

- visitando un albero binario di ricerca in ordine simmetrico, si ottiene la sequenza ordinata in ordine crescente delle etichette dei nodi.

Esempio 8.1
In figura 11 è mostrato un albero binario di ricerca, e come esso si modifica aggiungendo un nodo e successivamente togliendone un altro, secondo gli algoritmi presentati.

In **Appendice 4** è riportata la descrizione di una libreria di classi C++, organizzate gerarchicamente, che implementano i vari tipi di albero trattati. L'analisi di tale libreria evidenzia le analogie e le differenze presenti tra i vari tipi di albero.

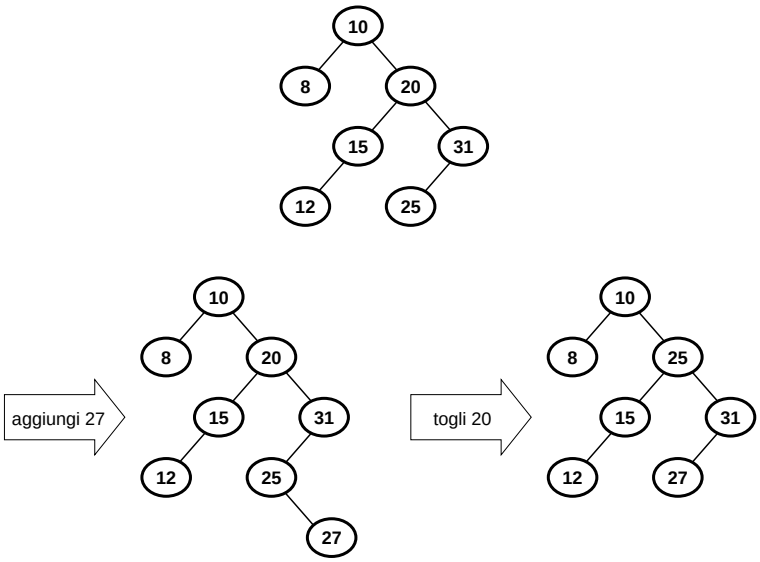


Figura 11: *Inserimento ed estrazione in un albero binario di ricerca.*

9 Heap

9.1 Il tipo di dato

Definizione 9.1 (heap)
Uno heap *è un albero binario quasi bilanciato etichettato che gode delle seguenti proprietà:*

- *i nodi dell'ultimo livello sono addossati a sinistra;*
- *in ogni sottoalbero l'etichetta della radice è maggiore o uguale a quella di tutti i discendenti.*

La figura 12 mostra un esempio di heap.

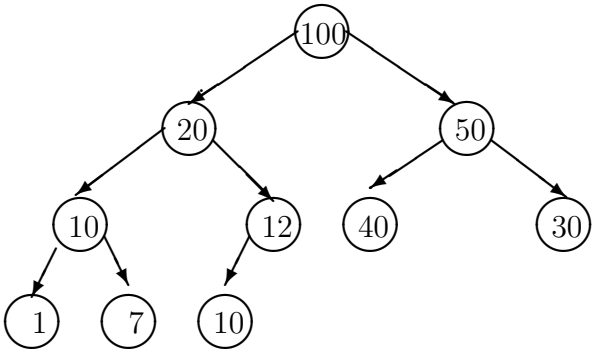


Figura 12: Esempio di heap

Un conseguenza della prima proprietà è che uno heap può essere linearizzato senza perdere informazione sulla struttura dell'albero: la linearizzazione deve semplicemente elencare l'albero per livelli. Nella linearizzazionbe suddetta vale la seguente proprietà (dimostrabile mediante induzione strutturale):

- il nodo di posizione i ha il figlio sinistro, se esiste, in posizione $2i+1$ e il figlio destro, se esiste, in posizione $2i+2$

Di conseguenza il nodo in posizione i, con $i > 0$, ha il padre in posizione $(i-1)/2$ (quoziente della divisione intera). Ad esempio, la linearizzazione per livelli dello heap di figura 12 è la seguente:

100 20 50 10 12 40 30 1 7 10.

La proprietà suddetta permette la memorizzazione di uno heap mediante un array, risparmiando così la memoria necessaria per i puntatori di una memorizzazione a lista multipla. Bisogna anche tener conto del numero di elementi dello heap, che può variare nel tempo e che coincide con l'indice dell'ultimo elemento occupato dell'array. Uno heap di interi può essere definito con la classe seguente, dove `h` è l'array che memorizza lo heap e `last` è l'indice dell'ultimo elemento occupato.

```
class Heap {
   int * h;
   int last;
   void up(int);
   void down(int);
   void exchange(int i, int j){
      int k=h[i];
      h[i]=h[j];
      h[j]=k;
   }
public:
   Heap(int);
   ~Heap();
   void insert(int);
   int extract();
};
```

Le operazioni di uno heap sono l'*inserimento* di un nuovo elemento e l'*estrazione dell'elemento massimo*. La realizzazione della classe `Heap` puo essere la seguente:

```
// costruttore
Heap::Heap(int n){ h=new int[n]; last=-1;}

// distruttore
Heap::~Heap() {delete [] h;}

// inserimento
void Heap::up(int i) {
    if (i > 0)
       if (h[i] > h[(i-1)/ 2]) {
          exchange(i,(i-1)/2);
          up((i-1)/2);
       }
 }
void Heap::insert (int x) {
    h[++last]=x;
    up(last);
 }
```

```
// estrazione

void Heap::down(int i) {
    int son=2*i+1;                                  // son=figlio sinistro di i
    if (son == last) {                              // se i ha un solo figlio
       if (h[son] > h[i]) exchange(i,last);         // padre e figlio si scambiano
                                                    // se il figlio e' maggiore
    }
    else if (son < last) {                  // se i ha entrambi i figli
       if (h[son] < h[son+1]) son++;        // il padre si scambia col
                                            // maggiore fra i due
       if (h[son] > h[i]) {                 // se questo e' maggiore del padre
          exchange(i,son);
          down(son);
       }
    }
 }
int Heap::extract() {
    int r=h[0];
    h[0]=h[last--];
    down(0);
    return r;
 }
```

La funzione `insert` prima memorizza l'elemento nuovo nella prima posizione libera dell'array (cioé $last + 1$) e quindi chiama la procedura `up` che risistema l'array in modo da mentenere le proprietà dello heap, facendo risalire x con scambi figlio-padre fino a trovare un padre maggiore del figlio o fino ad aver raggiunto la radice.

L'altra operazione definita sullo heap è l'estrazione del massimo elemento. Il massimo elemento è la radice e cioè il primo elemento dell'array. La funzione `extract` realizza il seguente algoritmo: la radice viene conservata per essere poi restituita, l'ultimo elemento viene sostituito alla radice, l'indice dell'ultimo elemento (`last`) viene decrementato e viene chiamata una funzione `down` che fa scendere l'etichetta della nuova radice con scambi successivi padre-figlio per mantenere la proprietà dello heap. Infine viene restituito il valore della vecchia radice.

La complessità di `insert` ed `extract` è $\mathcal{O}(\log n)$. Per rendercene conto, consideriamo prima la funzione `insert`. Poiché le prime due istruzioni hanno tempo costante, dobbiamo considerare la chiamata a `up` e quindi la complessità di `up` . La sua relazione di ricorrenza è la seguente, dove b è il tempo costante per la valutazione delle varie espressioni e per lo scambio:

$$\begin{aligned} T(1) &= a \\ T(n) &= T(\tfrac{n}{2}) + b \end{aligned}$$

Ogni volta che `up` fa uno scambio fra figlio e padre, si risale di un livello e i livelli di uno heap di n nodi sono $\log n$, poiché uno heap è un albero (quasi) bilanciato. Con ragionamenti simili si giunge alla conclusione che anche `extract` è dello stesso ordine di complessità.

Lo heap è particolarmente indicato per l'implementazione del tipo di dato astratto *coda con priorità*, che è una coda in cui gli elementi contengono, oltre all'informazione, un intero che ne definisce la priorità e, in caso di estrazione, l'elemento da estrarre deve essere quello con maggiore priorità.

9.2 L'algoritmo di ordinamento *heap-sort*

L'algoritmo di ordinamento $heap - sort$, dato un array A di elementi da ordinare, prima costruisce uno heap contenente gli elementi di A e poi applica n volte una funzione simile alla estrazione dallo heap, che però, invece di riscrivere l'ultimo elemento nella prima posizione dell'array, scambia la radice con l'ultimo elemento. Il programma è il seguente, dove la riorganizzazione di un array in modo che questo rappresenti uno heap è eseguita dalla funzione `buildHeap`. Le funzioni `down` ed `extract` sono ridefinite per essere chiamate da `heapSort`; in particolare, `extract` non ritorna alcun valore.

```
void down(int * h, int i, int last) {
    int son=2*i+1;
    if (son == last) {
       if (h[son] > h[i]) exchange(h, i,last);
    }
    else if (son < last) {
       if (h[son] < h[son+1]) son++;
       if (h[son] > h[i]) {
          exchange(h, i,son);
          down(h, son, last);
       }
    }
}

void extract(int* h, int & last) {
    exchange(h, 0,last--);
    down(h, 0, last);
}

void buildHeap(int* A, int n) {
  for (int i=n/2; i>=0; i--) down(A,i,n);
}

void heapSort(int* A, int n) {
  buildHeap(A,n-1);
  int i=n-1;
  while (i > 0) extract(A,i);
}
```

`buildHeap` chiama `down` soltanto sui primi $\frac{n}{2}$ elementi dell'array, poiché gli elementi che si trovano nella seconda metà sono foglie. Notiamo che sul secondo quarto dell'array, cioè sugli elementi da `[(last / 4)+1]` a `[last / 2]`, `down` viene chiamata al massimo due volte, la chiamata da `buildHeap` e un'altra ricorsiva, se l'elemento è minore del maggiore tra i suoi figli; altre chiamate non ci sono, poiché i figli sono foglie. In maniera analoga possiamo renderci conto che sul secondo ottavo dell'array le chiamate a `down` sono al massimo tre, sul secondo sedicesimo quattro e così via. Quindi abbiamo:

$$\begin{aligned} T(n) &= 2\tfrac{n}{4} + 3\tfrac{n}{8} + 4\tfrac{n}{16} .. + \log n \\ &= n(\tfrac{2}{4} + \tfrac{3}{8} + \tfrac{4}{16} + .. \tfrac{\log n}{n}) \\ &= n \textstyle\sum_{i=2}^{\log_2 n} \frac{i}{2^i} \end{aligned}$$

Abbiamo che $\sum_{i=1}^{\infty} \frac{i}{2^i}$ tende a 2, quindi $T(n) \leq 2n \in \mathcal{O}(n)$ e quindi `buildHeap` ha complessità lineare. Per quanto riguarda la complessità di `heapSort`, abbiamo che la chiamata a `buildHeap` ha complessità

$\mathcal{O}(n)$, l'assegnamento $\mathcal{O}(1)$ e il ciclo del `while` viene eseguito n volte, se n è il numero di elementi dell'array, mentre ogni chiamata a `down` ha complessità logaritmica. Quindi abbiamo $\mathcal{O}(n) + \mathcal{O}(1) + \mathcal{O}(n \log n) = \mathcal{O}(n \log n)$. Ne segue che *heap-sort* ha la stessa complessità di *mergesort* e, come vedremo, è un algoritmo ottimo.

10 Limiti inferiori

10.1 Le notazioni Ω-grande e Θ-grande

Mentre la notazione $\mathcal{O}$-grande fornisce un limite superiore al comportamento asintotico di una funzione, la notazione che ora introduciamo fornisce un limite inferiore.

Definizione 10.1
Consideriamo una funzione $g : N \to N$. Si dice che $g(n)$ è di ordine $\Omega(f(n))$ ($g(n)$ è $\Omega(f(n))$) se esistono un intero n_0 ed una costante $c > 0$ tali che, per ogni $n \geq n_0, g(n) \geq cf(n)$.

In altre parole, $g(n)$ è di ordine $\Omega(f(n))$ se $f(n)$ cresce meno o nello stesso modo di $g(n)$, fatta eventualmente eccezione per alcuni valori "piccoli" di n. Analogamente al caso della notazione $\mathcal{O}$-grande, $\Omega(f(n))$ può essere visto come l'insieme delle funzioni che non crescono meno di $f(n)$. Per esempio, n^3 è $\Omega(n^2)$. Per le espressioni Ω-grande si possono definire delle regole di semplificazione analoghe a quelle per la semplificazione delle espressioni $\mathcal{O}$-grande.

La notazione Θ mette in relazione due funzioni che crescono nello stesso modo:

Definizione 10.2
Una funzione $g(n)$ è di ordine $\Theta(f(n))$ ($g(n)$ è $\Theta(f(n))$) se $g(n)$ è $\mathcal{O}(f(n))$ e $g(n)$ è $\Omega(f(n))$.

La notazione Ω può essere utilizzata anche per i *problemi*. Dato un certo problema, risolto con un certo algoritmo di complessità $\mathcal{O}(f(n))$, ci si può chiedere se ci sia un algoritmo migliore per risolverlo. Se si riesce a dimostrare che non si può fare di meglio, diciamo che il problema in questione è $\Omega(f(n))$. Il limite inferiore può essere trovato con argomentazioni intuitive: per esempio, la visita di un albero di n nodi, dovendo raggiungere tutti i nodi, è limitata inferiormente dal loro numero e quindi è $\Omega(n)$. Una tecnica per trovare i limiti inferiori è descritta nel paragrafo seguente.

10.2 Alberi di decisione

Gli alberi di decisione sono una tecnica per trovare limiti inferiori a problemi basati su confronti, quando il tempo è proporzionale al numero di confronti che vengono effettuati durante il calcolo. Un *albero di decisione* è un albero binario che corrisponde ad un algoritmo nel modo seguente. Ogni foglia dell'albero rappresenta una soluzione del problema per un particolare assetto dei dati iniziali. Ogni percorso dalla radice ad una foglia rappresenta un'esecuzione dell'algoritmo per giungere alla soluzione relativa alla foglia: più precisamente rappresenta la relativa sequenza di di confronti. Per esempio, l'albero di decisione corrispondente all'algoritmo *selection-sort* quando l'array da ordinare contiene tre elementi è mostrato in figura 13.

Il caso peggiore di un algoritmo è quello che corrisponde al cammino più lungo dalla radice ad una foglia cioè alla disposizione iniziale dei dati corrispondente alla foglia più lontana dalla radice. Il caso migliore corrisponde al cammino più corto dalla radice ad una foglia. Il caso medio è la media della lunghezza dei cammini dalla radice ad una foglia.

Ogni algoritmo che risolve un problema che ha s soluzioni ha un albero di decisione corrispondente con almeno s foglie. Fra tutti gli alberi di decisione per un particolare problema cha s soluzioni, l'albero di decisione che minimizza la lunghezza massima dei percorsi (cioè che ha il minimo cammino più lungo) fornisce un limite inferiore al numero di confronti che un algoritmo che risolve il problema deve fare nel

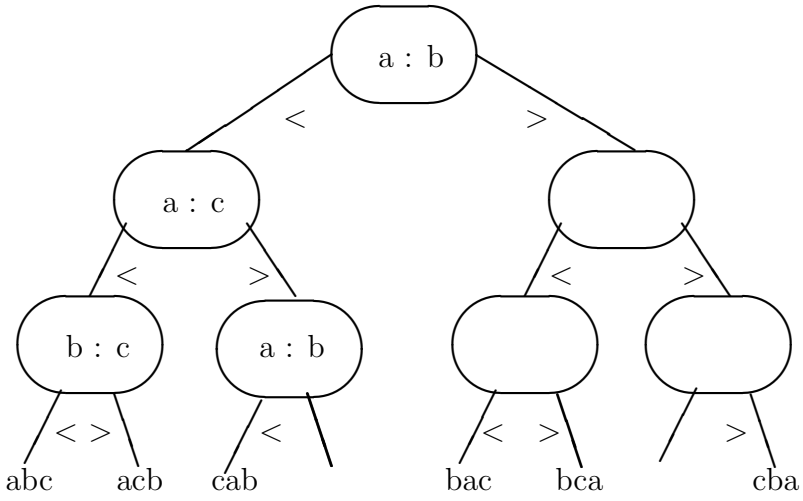


Figura 13: Albero di decisione di Selection-Sort

caso peggiore. L'albero di decisione che minimizza la lunghezza media dei percorsi fornisce un limite inferiore al numero di confronti che un algoritmo che risolve il problema deve fare nel caso medio. Il seguente teorema ci aiuta a stabilire un limite inferiore alla lunghezza del cammino massimo di ogni albero.

Teorema 10.1
Un albero binario di livello k ha al massimo 2^k foglie.
Dim. L'albero che ha il massimo numero di foglie è quello bilanciato, che ha 2^k foglie per il teorema 6.1.

Come conseguenza, ogni albero binario con s foglie ha livello maggiore o uguale $\log_2 s$. Quindi ogni albero binario con s foglie deve avere almeno un cammino di lunghezza maggiore o uguale a $\log_2 s$. Allora $\log_2 s$ è un limite inferiore alla lunghezza massima dei cammini per tutti gli alberi con almeno s foglie. Si può anche dimostrare che $\log_2 s$ è un limite inferiore alla lunghezza media dei cammini dalla radice ad una foglia per tutti gli alberi con almeno s foglie.

Quindi, dato un problema, se indichiamo con $s(n)$ il numero di soluzioni come funzione della dimensione dei dati n, abbiamo che il numero massimo e il numero medio di decisioni sono limitati inferiormente da $\log_2 s(n)$.

Utilizziamo questo risultato per trovare un limite inferiore al tempo degli algoritmi di ordinamento di n elementi basati su confronti. In questo caso le possibili disposizioni degli elementi sono $s(n) = n!$. Quindi ogni albero di decisione deve avere almeno $n!$ foglie e ogni albero deve avere almeno un cammino di lunghezza $\log(n!)$. Si può dimostrare che $n!$ è quasi uguale a $(\frac{n}{e})^n$. Abbiamo $\log(n!) = \log((\frac{n}{e})^n) = n\log(\frac{n}{e}) = n\log n - n\log e$. Quindi la complessità dell'ordinamento è limitata inferiormente da $n\log n$ e cioè è $\Omega(n\log n)$. Di conseguenza gli algoritmi *mergesort*, *quicksort* e *heapsort* sono ottimi.

Ribadiamo che il ragionamento con gli alberi di decisione riguarda soltanto i casi in cui gli algoritmi sono basati su confronti e non abbiamo nessuna informazione sugli elementi da trattare. Per esempio, l'ordinamento di n elementi che comprendono soltanto valori compresi tra 1 e n è di ordine lineare (vedi prossima sezione).

È importante anche notare che non è detto che sia possibile trovare per ogni problema un algoritmo con tempo uguale al limite inferiore trovato con gli alberi di decisione: per esempio, il problema di cercare un elemento in un insieme di n elementi è $\Omega(\log n)$. Infatti le possibili soluzioni sono che l'elemento cercato sia il primo, il secondo, ecc. e quindi sono n. Tuttavia anche l'algoritmo migliore, se non abbiamo

informazioni sugli elementi, fa necessariamente almeno n confronti. Se invece gli elementi sono ordinati, è possibile, come vedremo, raggiungere il limite inferiore.

Gli alberi binari bilanciati corrispondono ad algoritmi ottimi sia per il caso peggiore che per il caso medio. Se infatti consideriamo un albero binario di decisione quasi bilanciato B con s foglie, abbiamo che B ha livello $\log_2 s$ e quindi la lunghezza media dei cammini dalla radice ad una foglia è $\log_2 s$. Quindi possiamo affermare che un albero binario quasi bilanciato minimizza sia la lunghezza massima che la lunghezza media dei cammini dalla radice alle foglie rispetto a tutti gli alberi binari con lo stesso numero di foglie. Per renderci conto della minimizzazione della lunghezza media, consideriamo i due alberi in figura 14.a) e b): vediamo che l'albero bilanciato ha una lunghezza media di 3 mentre l'altro di quasi 4. Quindi sono da preferire gli algoritmi con albero di decisione (quasi) bilanciato.

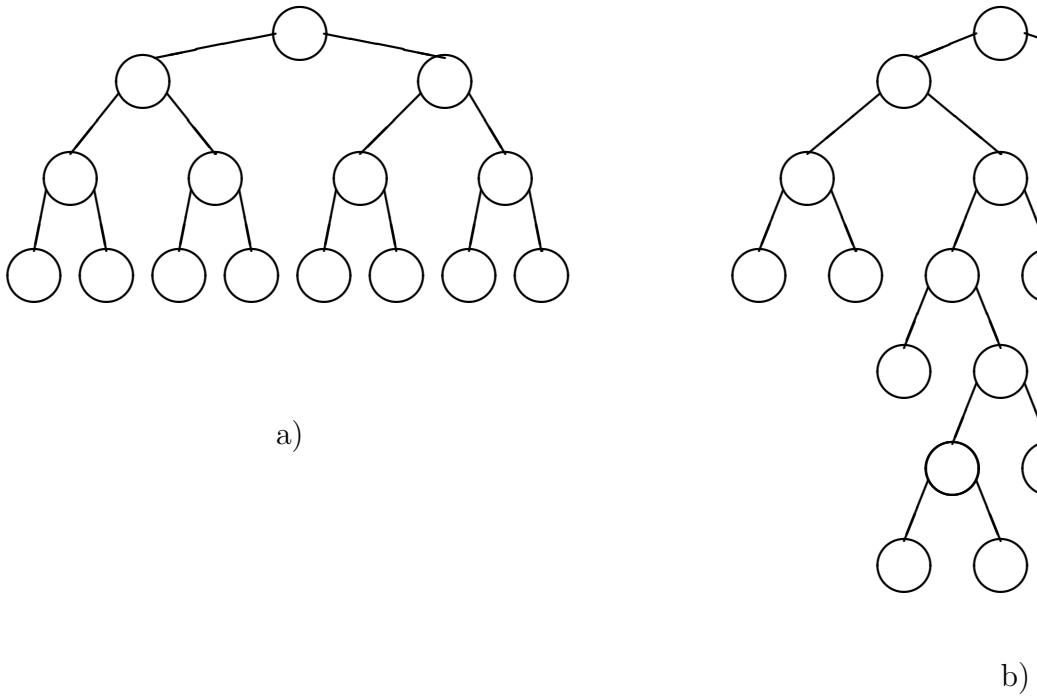


Figura 14:

11 Ricerca in un insieme

11.1 Insiemi semplici

In questa sezione descriveremo alcuni algoritmi per la ricerca di un elemento in un insieme. Consideriamo prima il caso in cui l'insieme sia rappresentato con una lista. Una semplice funzione ricorsiva che cerca l'elemento di informazione `x` in una lista di interi è la funzione `belongs` definita nella sezione 3. è semplice rendersi conto che la complessità della funzione è lineare. Se l'insieme è invece rappresentato con un albero binario di ricerca, la ricerca ha complessità logaritmica (vedi il capitolo relativo). Naturalmente, l'albero è più complesso da gestire nel caso di inserzioni nell'insieme. Consideriamo ora il caso in cui l'insieme sia memorizzato in un array. Il primo algoritmo (ricerca lineare o *completa*) esamina tutti gli elementi dell'array, come nel caso delle liste. La sua complessità è lineare. Una possibile funzione che esegue la ricerca lineare è la seguente:

```
int linearSearch (InfoType *A , int n, InfoType x) {
   for (int i=0;i<n;i++) if (A[i]==x) return 1;
   return 0;
}
```

Se gli elementi dell'insieme sono mantenuti ordinati, è possibile una ricerca più efficiente, detta ricerca *binaria* o *logaritmica*, simile a quella utilizzata per l'albero binario di ricerca, che consiste nell'esaminare l'elemento centrale dell'array e, se questo è diverso dall'elemento da ricercare, andare a cercare nella prima metà dell'array, mentre, se è maggiore, nella seconda metà. L'algoritmo può essere definito nel modo seguente:

```
 int binSearch (InfoType *A ,InfoType x, int i=0, int j=n-1) {
    if (i > j) return 0;
    int k=(i+j)/2;
    if (x == A[k]) return 1;
    if (x < A[k]) return binSearch(A, x, i, k-1);
    else return binSearch(A, x, k+1, j)
 }
```

È facile dimostrare per induzione la correttezza del programma. La relativa relazione di ricorrenza è :

$$\begin{aligned} T(0) &= a \\ T(n) &= T(\tfrac{n}{2}) + b \quad n > 0 \end{aligned}$$

e quindi $T(n)$ è $\mathcal{O}(\log n)$. Per quanto visto nel capitolo sui limiti inferiori, abbiamo che il problema della ricerca con algoritmi basati su confronti è $\Omega(\log n)$. Quindi la ricerca binaria raggiunge tale limite inferiore. Naturalmente può non essere conveniente per implementare un insieme se ci sono frequenti inserzioni e cancellazioni, poiché l'array deve rimanere ordinato.

11.2 Metodo hash

Un metodo di ricerca in un insieme, non completamente basato su confronti, è il metodo detto *hash* (to hash = rimescolare). Questo metodo consiste nel definire una funzione hash h che associa ad ogni elemento dell'insieme una posizione nell'array (indirizzo hash), cioè h: `InfoType` $\rightarrow$ indici. Questo vuol dire che, se l'elemento x è presente nell'insieme, esso deve trovarsi nella posizione corrispondente all'indice $h(x)$. Una funzione hash genera tutti e soli gli indici dell'array, quindi h deve essere surgettiva. Quando h è anche iniettiva, la posizione degli elementi è essere sempre diversa per elementi diversi. Se h è iniettiva, abbiamo il cosiddetto *accesso diretto*:

```
bool hashSearch (InfoType *A , int n, InfoType x) {
   int i=h(x); if (A[i]==x) return true;
   else return false;
}
```

La complessità della ricerca è ovviamente $\mathcal{O}(1)$, se il tempo per il calcolo della funzione hash è costante. Naturalmente l'inserimento di un elemento x deve essere effettuato in posizione $h(x)$ e la cancellazione di un elemento x inserirà un particolare valore in $h(x)$, che indica che l'array non contiene elementi in quella posizione. Un esempio di funzione hash iniettiva, se gli elementi sono interi, è $h(x) = x$. Se non abbiamo alcuna informazione sulla struttura degli elementi dell'insieme, per progettare un accesso diretto bisogna prevedere tutte i possibili elementi e questo, in generale, porta ad una dimensione dell'array che è di molto superiore al numero massimo n di elementi dell'insieme. Per esempio, se volessimo utilizzare una ricerca ad accesso diretto per un insieme di parole italiane di lunghezza inferiore o uguale a 10, dovremmo

associare ad ognuna delle possibili sequenze di caratteri di lunghezza inferiore o uguale a 10 un indice diverso, e le sequenze di caratteri di lunghezza inferiore o uguale a 10 sono $26^{10} + 26^9 + \ldots 26$, mentre il numero di parole italiane formate al massimo da 10 caratteri è di molto inferiore.

Se ci sono esigenze di ottimizzazione dell'occupazione di memoria tali da non permettere l'uso di una funzione hash iniettiva, è necessario rilasciare il vincolo della iniettività. Se la dimensione dell'array è M e gli elementi sono interi, una possibile funzione hash (hash *modulare*) è : `h(x)= x % M`. Con questa organizzazione viene perduto il vantaggio della complessità costante della ricerca ed è necessario, una volta individuato l'indirizzo hash dell'elemento, fare delle ulteriori operazioni per trovarlo effettivamente. Si dice che due elementi $x1$ e $x2$ *collidono* se hanno lo stesso indirizzo hash, cioè se $h(x1) = h(x2)$ e il fenomeno si chiama *collisione*. La gestione delle collisione può essere fatta in due modi.

Supponiamo inizialmente che gli elementi, una volta inseriti nella struttura dati, non possano più essere cancellati. La prima soluzione utilizza l'array nel modo seguente: se l'elemento cercato non si trova al suo indirizzo hash, lo si cerca in posizioni successive dell'array. Con questo tipo di organizzazione la dimensione dell'array deve essere maggiore o uguale ad n. Questo metodo viene chiamato ad indirizzamento aperto. Uno speciale simbolo deve contraddistinguere le posizioni vuote dell'array (ad esempio -1 può essere usato se consideriamo i numeri naturali).

```
bool hashSearch (int *A , int n, int x) {
   int i=h(x);
   for (int j=0; j<n; j++) {
     int pos = (i+j)%n;
     if (A[pos]==-1) return false;
     if (A[pos]==x) return true;
   }
   return false;
}
```

La ricerca dell'elemento quando esso non è presente all'indirizzo hash si chiama metodo di *scansione*. Il più semplice metodo di scansione è quello adottato nella soluzione proposta sopra, cioe quello di esaminare tutte le posizioni successive all'indirizzo hash. Con questo tipo di organizzazione, una volta individuato l'indirizzo hash di un elemento x, è possibile che nella ricerca si debbano attraversare posizioni dell'array occupate, oltre che da elementi che collidono con x, anche da elementi con indirizzo hash diverso da $h(x)$. Questo fenomeno prende il nome di *agglomerato*.

L'inserimento è simile, in quanto continua a scorrere le posizioni successive dell'array, alla ricerca di una cella vuota. L'inserimento fallisce se nell'array non è presente alcuna cella vuota.

```
  int hashInsert(int *A , int n, int x) {
   int i=h(x);
   for (int j=0; j<n; j++) {
     int pos = (i+j)%n;
     if (A[pos]==-1) {
       A[pos] = x;
       return 1;
     }
   }
   return 0;
}
```

Se ammettiamo anche cancellazioni nell'array, l'elemento da cancellare non può essere rimpiazzato con uno vuoto. Consideriamo infatti un'array di dimensione 10 che dovrà contenere numeri naturali, con funzione hash `h(x)= x % 100`. Supponiamo che nell'array siano stati inseriti, nell'ordine, i numeri 94, 52, e 674. La situazione sarà quella di Figura 15(a). Attorno alla posizione 4 si è formato un agglomerato.

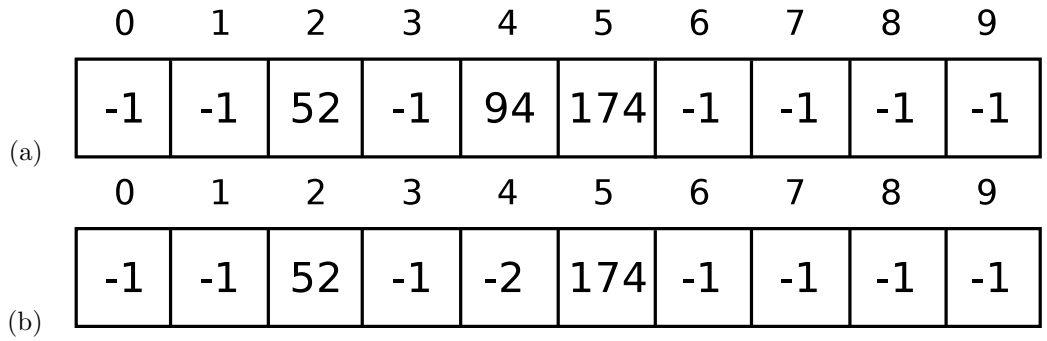


Figura 15: Numeri naturali memorizzati in un array con il metodo hash ad indirizzamento aperto e legge di scansione lineare. Il valore -1 indica una cella vuota, il -2 una cella cancellata.

Se adesso cancellassimo l'elemento 94, sostituendolo con una cella vuota, l'algoritmo di ricerca visto in precedenza darebbe un risultato errato per la ricerca dell'elemento 674. Allora, invece di sostituire l'elemento cancellato con il simbolo che indica una cella vuota, è necessario usare un altro simbolo per indicare che la cella è disponibile per un nuovo inserimento, ma la ricerca deve continuare. Ad esempio, in questo caso potremmo utilizzare il valore -2. La procedura `hashInsert` va modificata in modo da considerare come disponibile per un inserimento una cella precedentemente cancellata.

```
int hashInsert(int *A , int n, int x) {
  int i=h(x);
  for (int j=0; j<n; j++) {
    int pos = (i+j)%n;
    if ((A[pos]==-1)|| (A[pos]==-2)) {
      A[pos] = x;
      return 1;
    }
  }
  return 0;
}
```

Le cancellazioni degradano le prestazioni della ricerca in quanto le celle marcate con -2 vanno comunque esaminate. In generale, la legge di scansione può essere vista come una funzione s di due argomenti: il primo rappresenta la chiave da cercare, mentre mentre il secondo rappresenta il numero di elementi esaminati. Quindi $s(k, j)$ restituisce l'indirizzo del j-esimo elemento esaminato a partire dall'indirizzo $h(k)$, dove h è la funzione di hash usata. Le leggi di scansione lineare hanno la seguente forma:

$$s_{\text{lin}}(k, j) = (h(k) + cj) \bmod M$$

dove M è la dimensione dell'array e c una costante diversa da zero. Nell'esempio precedente abbiamo posto $c = 1$. Le leggi di *scansione quadratica* hanno la seguente forma:

$$s_{\text{q}}(k, j) = \lfloor h(k) + c_1 j + c_2 j^2 \rfloor \bmod M$$

dove $c_2 \neq 0$ e $\lfloor x \rfloor$ è la parte intera di x. Nel progetto di una legge di scansione è necessario controllare che la scansione visiti tutte le possibili celle vuote dell'array per ogni $i, j \in [0, m)$, per evitare che l'inserimento fallisca anche in presenza di array non pieno. Se la dimensione della tabella è una potenza di due $c_1 = c_2 = \frac{1}{2}$ visita tutte le possibili locazioni vuote per $i, j \in [0, m)$. Se M non è una potenza di due, la maggior parte delle scelte di c_1, c_2 porta ad un fattore di carico massimo del 50%. Leggi di scansione quadratica sono soggette ad agglomerati per tutte le chiavi che hanno la stessa hash. Questi

sono denominati agglomerati di secondo livello. Un'altro tipo di leggi di scansione, denominato ad *hashing doppio*, prevede di utilizzare un coefficiente che dipende dalla chiave cercata tramite una funzione hash. Ovvero:

$$s(k, j) = h(k) + h'(k)j \bmod M$$

dove h' è un'altra funzione hash. Leggi di scansione quadratica e ad hashing doppio cercano di evitare la formazione di agglomerati aumentando le prestazioni della ricerca cercando di disperdere il più possibile i dati sulla tabella. Questo può non essere sempre un vantaggio perché si perde località nei dati.

Il tempo di ricerca per tabelle hash con questa organizzazione dipende principalmente dal rapporto $\alpha = \frac{n}{M}$ (che deve essere sempre minore o uguale ad 1). Infatti, più è grande il numero di elementi dell'array, meno collisioni e agglomerati si possono presentare. In generale, il numero medio di accessi per trovare un elemento è al più $\frac{1}{1-\alpha}$. D'altra parte, a parità di rapporto $\frac{n}{M}$, la legge di scansione influenza il tempo medio di ricerca. Di seguito sono riportati i valori sperimentali del tempo medio di ricerca, per chiavi presenti nell'array, in funzione di α, nei casi di scansione lineare e quadratica:

$\frac{n}{M}$	**scansione lineare**	**scansione quadratica**
10%	1.06	1.06
50%	1.50	1.44
70%	2.16	1.84
80%	3.02	2.20
90%	5.64	2.87

I dati sopra riportati sono stati ricavati utilizzando le leggi di scansioni viste in precedenza su una tabella con M pari a 2^{16}, hashing modulare, e chiavi ricavate da una distribuzione uniforme.

Un problema con il metodo ad indirizzamento aperto è legato al fatto che, dopo molti inserimenti e cancellazioni, gli agglomerati possono aumentare notevolmente ed è necessaria una risistemazione dell'array per mantenere alta la velocità di ricerca.

Un modo alternativo, che non ha il problema degli agglomerati, è il metodo di *concatenazione*, in cui ogni elemento dell'array è un puntatore ad una lista che contiene tutti gli elementi che collidono. In questo tipo di memorizzazione la dimensione dell'array (M) può essere anche minore della dimensione dell'insieme (n). La ricerca consiste nell'accedere direttamente ad un elemento dell'array con la funzione hash, e poi nello scorrere la lista associata fino a trovare l'elemento cercato. Il tempo di accesso all'array è $\mathcal{O}(1)$, mentre per la scansione lineare della lista il tempo dipende dal rapporto $\frac{n}{M}$, cioè diminuisce con il diminuire di questo rapporto. Più precisamente, il tempo medio è $\mathcal{O}(\frac{n}{M})$, poiché ogni lista associata ad un indice ha lunghezza media $\frac{n}{M}$. Questo vuol dire che per $n = M$ il tempo è $\mathcal{O}(1)$, mentre per valori di M maggiori di n non si hanno sostanziali miglioramenti. Per l'inserimento il tempo è $\mathcal{O}(1)$: infatti possiamo pensare di inserire l'elemento x come primo elemento della lista associata ad $h(x)$. Per la cancellazione il tempo è uguale a quello della ricerca.

Il problema con il metodo di concatenazione può essere l'occupazione di memoria. Infatti viene utilizzata memoria per memorizzare i puntatori e questo ha un peso maggiore sull'occupazione totale se gli elementi sono di piccole dimensioni. Un secondo spreco di memoria è dovuto al fatto che alcune posizioni dell'array possono rimanere inutilizzate, se il corrispondente indice non viene mai generato.

11.2.1 La scelta della funzione hash

Un parametro che influenza la complessità della ricerca con il metodo hash è l'*uniformità* della funzione hash. Una funzione hash è *uniforme* se genera tutti gli indici dell'array con la stessa probabilità. Naturalmente, maggiore è l'uniformità della funzione, minore è la probabilità di avere collisioni e quindi agglomerati. Mostriamo adesso alcuni esempi di possibili funzioni hash. Supponiamo che le chiavi siano numeri in virgola mobile compresi fra 0 ed 1 (estremi esclusi). Una semplice funzione hash potrebbe essere la seguente:

```
int hash(float x) = { return (int)(x*M);}
```

dove M è la dimensione dell'array utilizzato. Analogamente, se le chiavi sono numeri interi codificati su w bit, è sufficiente dividerli per 2^w per ottenere un numero in virgola mobile compreso fra 0 e 1.

Questa funzione restituisce infatti un numero intero compreso fra 0 e $M-1$. Tuttavia, non è una buona funzione hash perché, se le chiavi non sono distribuite uniformemente nello spazio $(0,1)$ (come accade frequentemente nelle applicazioni reali) il rischio di collisioni è piuttosto alto in quanto solo i bit più significativi della chiave vengono utilizzati per generare l'indirizzo risultato.

Un'alternativa consiste nell'usare come funzione hash il resto della divisione per M, che è comunque un intero compreso fra 0 e $M-1$. Avendo a disposizione numeri in virgola mobile, si possono interpretarli come numeri interi prima di effettuare il modulo, ad esempio:

```
int hash(float x) = { return (*((int*)&x))%M;}
```

Tuttavia, anche questa funzione hash, per particolari valori di M, può portare a molte collisioni. Per esempio, se $M = 2^h$, questa funzione hash fa collidere tutte le chiavi che hanno gli h bit meno significativi identici. Se il numero M scelto è primo rispetto al numero di bit su cui è rappresentata la chiave, allora la funzione hash utilizza tutti i bit della chiave per generare il risultato, diminuendo così la probabilità di collisioni, anche per dati non equiprobabili. In generale, per diminuire le collisioni si sceglie come M un numero primo vicino ad una potenza di due. Nel grafico di Figura 16 sono evidenziati il numero di collisioni per le semplici funzion hash presentate in precedenza, quando la chiave è un `float` compreso fra 0 ed 1, il valore di M è fissato a 31, e le chiavi sono 990 numeri estratti da una distribuzione gaussiana di media 0.5, varianza 0.2. Come si può notare, la funzione hash modulare riesce a disperdere meglio le chiavi, che invece con la funzione prodotto seguono ancora l'andamento della gaussiana. In letteratura sono presenti funzioni hash più efficaci di un semplice prodotto o modulo [1], anche se a prezzo di complessità maggiori. Anche il tempo necessario per calcolare la funzione hash va infatti preso in considerazione.

11.3 Dizionari

Il metodo hash è utilizzato molto nelle basi di dati. Qui vediamo un esempio del suo utilizzo, insieme a quello di altre strutture dati viste finora, nella implementazione del tipo di dato astratto *dizionario* (detto spesso anche tabella). Una dizionario è un insieme di elementi, che sono coppie composte di due parti: *chiave* e *informazione*. Inoltre non esistono due elementi con la stessa chiave e informazione diversa. In generale tra gli elementi del dizionario non esiste alcuna relazione e non è stabilito un ordinamento per posizione o altro. L'operazione fondamentale su di un dizionario è la ricerca della informazione associata ad una certa chiave. Altre operazioni sono l'inserimento e la cancellazione di elementi. Un dizionario può essere una classe derivata dalla classe template astratta `Dictionary` riportata di seguito.

```
template <typename Key, typename Value>
class Dictionary {
public:
  virtual bool is_present(Key) const = 0;
  virtual std::pair<bool, Value> find(Key) = 0;
  virtual bool insert(Key, Value) = 0;
  virtual bool erase(Key) = 0;
};
```

Il metodo `is_present` restituisce `true` se un elemento con una data chiave è presente nel dizionario, `false` altrimenti. Il metodo `find` restituisce una coppia, il cui primo valore indica se nel dizionario è presente un'elemento con una data chiave, e in caso affermativo il secondo valore della coppia contiene il

[1] Si veda per esempio la hash FNV, http://isthe.com/chongo/tech/comp/fnv/

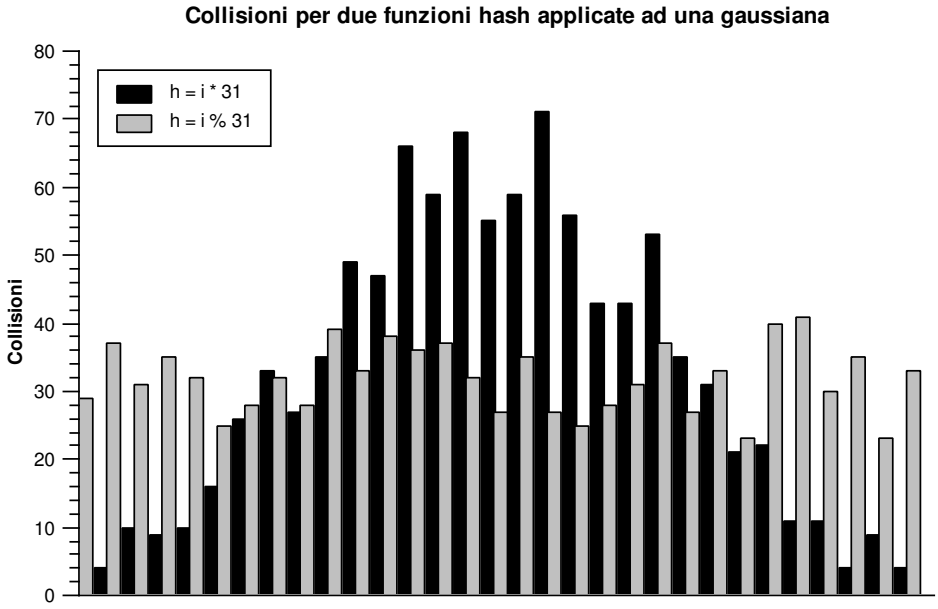


Figura 16: Le collisioni per le funzioni hash modulo (in grigio) e prodotto (in nero) quando applicate ad un insieme di numeri generati da una distribuzione gaussiana

valore associato alla chiave cercata. Il metodo `insert` inserisce una coppia chiave-valore nel dizionario, restituendo `true` se l'inserimento ha successo, `false` altrimenti. Il metodo `erase` cancella l'elemento con una data chiave dal dizionario, restituendo `true`. Se non è presente alcun elemento con tale chiave restituisce `false` lasciando il dizionario inalterato.

Vediamo adesso una serie di implementazioni possibili per il dizionario. Iniziamo da una lista semplice di coppie chiave-valore:

```
template <typename Key, typename Value>
class DictionaryList : public Dictionary<Key,Value> {
      struct Elem{
         Key key;      // chiave
         Value info;    // informazione
         Elem *next;
      };
      Elem * t;
public:
      DictionaryList(int) { t=NULL;};
      bool is_present(Key) const {...};
      std::pair<bool, Value> find(Key) {...};
      bool insert(Key, Value) {...};
      bool erase(Key) {...};
      ~DictionaryList() {...};
};
```

l'unica ricerca possibile è quella che scorre la lista, con tempo $\mathcal{O}(n)$. Per l'inserimento abbiamo un tempo $\mathcal{O}(n)$ (dobbiamo verificare che l'elemento non sia già presente), per la cancellazione $\mathcal{O}(n)$. Se memorizziamo il dizionario con un albero binario di ricerca,

```
template <typename Key, typename Value>
class DictionaryBST : public Dictionary<Key,Value> {
    struct Node{
            Key key;
            Value info;
            Node *left;
            Node *right;
      };
      Node * t;
public:
      DictionaryBST(int) { t=NULL;};
      bool is_present(Key) const {...};
      std::pair<bool, Value> find(Key) {...};
      bool insert(Key, Value) {...};
      bool erase(Key) {...};
      ~DictionaryBST() {...};
};
```

le operazioni di ricerca, inserimento e cancellazione hanno in media complessità logaritmica. Le procedure possono essere ottenute con lievi modifiche da quelle per la ricerca in insieme semplice.

Consideriamo ora la memorizzazione in array. Se n è il numero massimo di elementi della tabella, una possibile memorizzazione è con un array di dimensione $M = n$:

```
template <typename Key, typename Value>
class DictionaryArray : public Dictionary<Key,Value> {
      struct Telem{
            Key key;
            Value info;
      };
      Telem * t;
      int M;
public:
      DictionaryArray(int n) : M(m) { t = new Telem[n]};
      bool is_present(Key) const {...};
      std::pair<bool, Value> find(Key) {...};
      bool insert(Key, Value) {...};
      bool erase(Key) {...};
      ~DictionaryArray() { delete[] t;};
};
```

Per cercare un elemento si può effettuare una ricerca lineare con complessità $\mathcal{O}(n)$ oppure, se gli elementi sono mantenuti ordinati, si può usare la ricerca binaria con complessità $\mathcal{O}(\log n)$. Nel seguito, parleremo di *tabella a ricerca completa* se l'array non è ordinato e la ricerca è quindi lineare, e di *tabella a ricerca binaria* nel caso di ricerca binaria. Con una tabella a ricerca binaria c'è il problema di mantenere le chiavi ordinate, quindi la complessità dell'inserimento è lineare, mentre la complessità dell'inserimento nel caso di elementi non ordinati è costante. Nel caso di cancellazione, per la tabella con ricerca binaria abbiamo $\mathcal{O}(\log n)$ se non compattiamo l'array, mentre il tempo nel caso della tabella a ricerca completa è $\mathcal{O}(n)$.

Un altro metodo di ricerca è mediante il metodo hash, dove la funzione hash è applicata alla chiave. Per rendere indipendente il tipo di dato astratto dalla funzione hash usata, passeremo la funzione hash al costruttore della classe, attraverso un *funtore*. Un funtore è la versione C++ del concetto di puntatore a funzione del C. Viene realizzato tramite una classe o struttura che ridefinisce l'operatore di chiamata di funzione `operator()`. A questo scopo, definiamo la struttura astratta `hash_function` come derivata dal funtore predefinito `unary_function<A, B>` che rappresenta una funzione con un solo parametro di tipo `A` e risultato di tipo `B`.

```
template <typename Key>
struct hash_function : public std::unary_function<Key, unsigned int> {
  int size;
  explicit hash_function(int s) : size(s) {};
  virtual unsigned int operator()(Key k) = 0;
};
```

Una funzione hash sarà un funtore di un tipo concreto derivato da `hash_function`, che dovrà ridefinire l'operatore di chiamata di funzione, passandogli un oggetto di tipo `Key` e restituendo un `unsigned int`.

Con questa premessa, vediamo adesso una possibile implementazione del tipo di dato astratto `Dictionary` tramite una tabella hash ad accesso diretto.

```
template <typename Key, typename Value>
class DirectAccessHash : public Dictionary<Key, Value>{
  Value** table;
  unsigned int size;
  hash_function<Key>& hash;

public:
  DirectAccessHash(unsigned int _size, hash_function<Key>& h) : hash(h){
    size = _size;
    table = new Value*[size];
    for (unsigned int i=0;i<size;i++)
      table[i] = 0;
  }

  bool is_present(Key k) const { return table[hash(k)]; }

  std::pair<bool, Value> find(Key k) {
    std::pair<bool, Value> result;
    Value* e = table[hash(k)];
    if (e) {
      result.first = true;
      result.second = *e;
    } else
      result.first = false;
    return result;
  }

  bool insert(Key k, Value v) {
    Value* e = table[hash(k)];
    if (e)
      return false;
```

```
    table[hash(k)] = new Value(v);
    return true;
  }

  bool erase(Key k) {
    Value* e = table[hash(k)];
    if (!e)
      return false;
    delete e;
    table[hash(k)] = 0;
    return true;
  }

  ~DirectAccessHash() {...}
};
```

Con questa implementazione, ricerca, inserimento e cancellazione sono tutte operazioni a complessità costante.

Un possibile utilizzo della classe `DirectAccessHash` è il seguente (si suppone che le chiavi siano interi compresi fra 100 e 199).

```
struct hash_injective : public hash_function<int> {
  hash_injective(unsigned int s) : hash_function<int>(s) {};
  unsigned int operator()(int k) {
    return k-100;
  }
}; // funzione hash ad accesso diretto

int main() {
  unsigned int tableSize = 100;
  hash_injective hi(tableSize);
  Dictionary<int, std::string>* dp;
  dp = new DirectAccessHash<int, std::string>(tableSize, hi);
  dp->insert(156,"pippo");
  dp->insert(132,"pluto");
  std::cout << dp->find(156).second << std::endl; // OUTPUT: pippo
  std::cout << dp->find(167).first << std::endl;  // OUTPUT: 0
  delete dp;
}
```

Tuttavia, la scelta di una funzione hash ad accesso diretto non è sempre praticabile, specialmente quando lo spazio delle chiavi è vasto. Una versione di `Dictionary` come tabella hash con liste di concatenazione è la seguente.

```
template <typename Key, typename Value>
class ChainedHash : Dictionary<Key, Value>{
  struct Elem {
    Key key;
    Value value;
    Elem* next;
    Elem(Key k, Value v) { key = k; value = v;}
  };
```

```
  Elem** table;
  unsigned int size;
  hash_function<Key>& hash;
public:
  ChainedHash(unsigned int _size, hash_function<Key>& h) : hash(h){
    size = _size;
    table = new Elem*[size];
    for (unsigned int i=0;i<size;i++)
      table[i] = 0;
  }

  bool is_present(Key k) const {
    Elem* e = table[hash(k)];
    while (e) {
      if (e->key == k)
        return true;
      else
        e = e->next;
    }
    return false;
  }

  std::pair<bool, Value> find(Key k) {
    std::pair<bool, Value> result;
    Elem* e = table[hash(k)];
    while (e) {
      if (e->key == k) {
        result.first = true;
        result.second = e->value;
        return result;
      }
      else
        e = e->next;
    }
    result.first = false;
    return result;
  }

  bool insert(Key k, Value v) {
    if (is_present(k))
      return false;
    unsigned int position = hash(k);
    Elem* e = table[position];
    table[position] = new Elem(k,v);
    table[position] -> next = e;
    return true;
  }

  bool erase(Key k) {
    unsigned int position = hash(k);
```

```
    Elem* e = table[position];
    if (!e) return false;
    if (e->key == k) {
      table[position] = e->next;
      delete e;
      return true;
    }
    while (e->next) {
      if (e->next->key == k) {
        Elem* p = e->next;
        e->next = p->next;
        delete p;
        return true;
      }
      e = e->next;
    }
    return false;
  }

  ~ChainedHash() {...}
};
```

Volendo invece implementare una tabella hash con indirizzamento aperto al variare della legge di scansione, possiamo nuovamente utilizzare un funtore come nel seguente tipo di dato astratto.

```
template <typename Key, typename Value>
class OpenHash : public Dictionary<Key, Value>{
  struct Elem {
    Key key;
    Value value;
    Elem(Key k, Value v) {key = k; value = v;}
  };
  Elem** table;
  Elem* erased; // celle cancellate
  unsigned int size;
  hash_function<Key>& hash;
  scan_law& scan;
public:
  OpenHash(unsigned int _size,
           hash_function<Key>& h,
           scan_law& s) : hash(h), scan(s){
    size = _size;
    table = new Elem*[size];
    for (unsigned int i=0;i<size;i++)
      table[i] = 0;
    erased = (Elem*)table;
  }

  bool is_present(Key k) const {
    int i = hash(k);
    for (int j=0; j<size; j++) {
```

```
    int pos = scan(i,j);
    if (!table[pos]) return false;
    if (table[pos] == erased) continue;
    if (table[pos]->key == k) return true;
  }
  return false; // si arriva qui solo con tabella piena
}

std::pair<bool, Value> find(Key k) {
  int i = hash(k);
  std::pair<bool, Value> result;
  result.first = false;
  for (int j=0; j<size; j++) {
    int pos = scan(i,j);
    if (!table[pos]) return result;
    if (table[pos] == erased) continue;
    if (table[pos]->key == k) {
      result.first = true;
      result.second = table[pos]->value;
      return result;
    }
  }
  return result;
}

bool insert(Key k, Value v) {
  int i = hash(k);
  for (int j=0; j<size; j++) {
    int pos = scan(i,j);
    if (!table[pos] || table[pos] == erased) {
      table[pos] = new Elem(k,v);
      return true;
    }
    if (table[pos]->key == k) return false;
  }
  return false;
}

bool erase(Key k) {
  int i = hash(k);
  for (int j=0; j<size; j++) {
    int pos = scan(i,j);
    if (!table[pos]) return false;
    if (table[pos] == erased) continue;
    if (table[pos]->key == k) {
      delete table[pos];
      table[pos]=erased;
      return true;
    }
  }
  return false;
```

```
  }

  ~OpenHash() {
    for(int i=0;i<size;i++)
      if (table[i] != erased)
        delete table[i];
    delete[] table;
  }
};
```

Il terzo parametro del costruttore è nuovamente un funtore che specifica la legge di scansione. Un'esempio di utilizzo della classe `OpenHash` è il seguente:

```
// funzione hash
struct hash_modulus : public hash_function<float> {
  hash_modulus(unsigned int s) : hash_function<float>(s) {};
  unsigned int operator()(float d) {
    int i = *((int*) (&d));
    return (i%size);
  }
};

//legge di scansione lineare
struct linear_scan : public scan_law {
  linear_scan(int s) : scan_law(s) {};
  unsigned int operator() (unsigned int start, unsigned int offset) {
    return (start + offset)%size;
  }
};

int main() {
  unsigned int tableSize = 10000;
  hash_modulus hm(tableSize);
  linearscan ls(tableSize);
  Dictionary<int, std::string>* dp;
  dp = new OpenHash<int, std::string>(tableSize, hi, ls);
  ...
}
```

In generale, nessuno dei metodi sopra descritti di implementazione di una tabella è preferibile in assoluto. Nella scelta di un metodo di memorizzazione per una tabella data, bisogna tener conto, come in un qualsiasi caso di implementazione di tipo di dato astratto, dei requisiti del problema in termini di tempo di esecuzione e occupazione di memoria.

Supponiamo che la tabella sia fissa, cioè tale che inserzioni e cancellazioni siano eventi estremamente rari, come nel caso di un elenco del telefono. Se abbiamo come requisito fondamentale la minimizzazione dell'occupazione di memoria, sceglieremo comunque un array (liste e alberi occupano memoria per i puntatori). Dobbiamo quindi scegliere il tipo di ricerca fra completa, binaria e hash con $k = n$. La ricerca completa è da scartare, perchè comunque di complessità superiore alle altre due. Se riusciamo a trovare una funzione hash iniettiva, ovviamente questa è da preferire. Nel caso generale, per scegliere tra binaria e hash, dobbiamo confrontare $\log n$ con uno dei valori ottenuti per $\frac{n}{k} = 1$ e una qualche legge di scansione.

Facciamo ora il caso di tabella fissa, ma obiettivo primario minimizzazione del tempo medio di ricerca. Se non abbiamo un limite all'occupazione di memoria possiamo utilizzare una ricerca ad accesso diretto, definendo una funzione hash iniettiva. In genere, però, anche se abbiamo come principale requisito l'efficienza, vi sarà sempre un limite all'occupazione di memoria. Se il limite è , supponiamo, k, dobbiamo confrontare la ricerca binaria (con un array dimensionato comunque ad n) con una ricerca hash con rapporto $\frac{n}{k}$. Se non riusciamo a trovare una funzione hash iniettiva con codominio $[0..k]$, dobbiamo fare i confronti fra ricerca binaria e hash come nel caso precedente.

Se la tabella invece non è fissa e si vuole minimizzare l'occupazione di memoria, bisogna confrontare l'occupazione dell'array con quella della lista e dell'albero binario di ricerca, che potrebbero occupare in media meno memoria di un array, nonostante i puntatori, essendo strutture a dimensione variabile. Bisogna anche considerare una tabella hash con il metodo di concatenazione. Una volta scelta la struttura che occupa meno memoria, si può definire la ricerca in modo opportuno. Se la struttura più conveniente è la lista, la ricerca obbligata è la completa. La ricerca è anche obbligata se abbiamo scelto il metodo di concatenazione. Se la struttura scelta è l'array, la ricerca binaria è da scartare a causa del fatto che la tabella non è fissa e si può scegliere una codifica hash con $\frac{n}{k} = 1$. L'albero binario di ricerca non è mai competitivo con la lista dal punto di vista dell'occupazione di memoria, ma potrà essere scelto quando venga richiesto un compromesso tra occupazione di memoria ed efficienza della ricerca.

12 Altre strategie di programmazione

12.1 Programmazione dinamica

La *programmazione dinamica* o *tabulazione* viene spesso utilizzata per risolvere problemi di ottimizzazione. Questi problemi in generale ammettono diverse soluzioni alle quali si può associare un valore. Il problema consiste nel trovare il valore ottimo ed eventualmente una delle soluzioni che hanno tale valore. Il valore ottimo viene costruito con una strategia "bottom-up". La programmazione dinamica può essere utilizzata quando non è possibile applicare il metodo del divide et impera, poiché non si sa con esattezza quali sottoproblemi risolvere per affrontare un problema dato e non è possibile partizionare l'insieme in sottoinsiemi disgiunti. Allora si risolvono tutti i sottoproblemi a partire dal basso e si conservano i risultati ottenuti per poterli usare successivamente. Naturalmente la complessità del metodo dipende dal numero dei sottoproblemi: il metodo dà buoni risultati solo se questo numero è polinomiale.

Le caratteristiche che un problema deve avere perchè possa essere risolto in modo efficiente con questa tecnica sono due:

- *sottostruttura ottima*: una soluzione ottima del problema contiene la soluzione ottima dei sottoproblemi
- *sottoproblemi comuni* : un algoritmo ricorsivo richiede di risolvere lo stesso sottoproblema più volte.

Vediamo un esempio di programmazione dinamica. Data una sequenza di elementi α, una *sottosequenza* è ottenuta cancellando da α zero o più elementi e mantenendo gli altri nello stesso ordine con cui appaiono in α. Date due sequenze, una *sottosequenza comune* è una sequenza che è sottosequenza di entrambe; una *più lunga sottosequenza comune* PLSC è una sottosequenza con lunghezza maggiore o uguale a quella di una qualsiasi altra sottosequenza comune.

Esempio 12.1
Siano α = abcabba e β = cbabac, due PLSC sono baba e cbba. Non ci sono sottosequenze comuni di lunghezza maggiore, quindi la lunghezza massima delle sottosequenze comuni è 4.

Vogliamo trovare un algoritmo che calcoli la lunghezza massima delle sottosequenze comuni di due sequenze (valore della soluzione ottima). Modificando leggermente tale algoritmo, saremo in grado di dare anche una PLSC. Impostiamo prima il problema in modo ricorsivo. Consideriamo due sequenze

$\alpha = a_1..a_m$ e $\beta = b_1..b_n$ di lunghezza m e n rispettivamente, e definiamo la massima lunghezza $L(i,j)$, per ogni $0 \leq i \leq m, 0 \leq j \leq n$, delle sottosequenze comuni fra $a_1..a_i$ e $b_1..b_j$, indicando con $i = 0$ e $j = 0$ la sequenza vuota.

- $L(0,0) = L(i,0) = L(0,j) = 0$
- $L(i,j) = L(i-1, j-1) + 1$ se $a_i = b_j$
- $L(i,j) = max(L(i,j-1), L(i-1,j)$ se $a_i \neq b_j$

Un programma ricorsivo basato su questa definizione è il seguente, dove α e β sono memorizzate negli array `a` e `b` (dimensionate a `m+1` e `n+1` elementi rispettivamente), `char` è il tipo degli elementi e `max` è una funzione che restituisce il maggiore fra due elementi. La funzione è chiamata con `L(a,b,m,n)`.

```
int length(char*  a, char* b, int i, int j) {
    if (i==0 || j==0) return 0;
    if (a[i]==b[j]) return length(a, b, i-1, j-1)+1;
    else return max(length(a,b,i,j-1),length(a,b,i-1,j));
};
```

La funzione ha un tempo esponenziale nel più piccolo fra n e m. Il problema ha la caratteristica della sottostruttura ottima: infatti la soluzione ottima considerando due sequenze α e β è un prefisso della soluzione ottima considerando due sequenze di cui α e β sono prefissi. Inoltre il problema ha la caratteristica dei sottoproblemi comuni, come si vede dal programma ricorsivo descritto sopra, che esegue più volte la stessa chiamata.

Per costruire un algoritmo basato sulla programmazione dinamica possiamo ragionare così: costruiamo una tabella contenente i valori di L(i,j) per ogni i e j, a partire dai valori per gli indici più piccoli.

```
int quickLength(char* a, char* b) {
    int L [m+1][n+1];
    for (int j=0; j<=n; j++) L[0][j]=0;
    for (int i=1; i<=m; i++) {
        L[i][0]=0;
        for (j=1; j<=n; j++)
            if (a[i] != b[j]) L[i][j] = max(L[i][j-1],L[i-1][j]);
            else  L[i][j]=L[i-1][j-1]+1;
    }
    return L[m][n];
}
```

La tabella L(i,j) relativa alle sequenze dell'esempio 12.1 è mostrata in figura 17. Una volta costruita la tabella L(i,j), è possibile utilizzarla per trovare una PLSC, seguendo un cammino attraverso la tabella, che parte dall'angolo in basso a destra e costruisce la PLSC in ordine inverso, cioè a partire dall'ultimo elemento. Supponiamo di partire dalla riga i e colonna j. Se $a_i = b_j$, allora L(i,j) è stato scelto come 1+L(i-1,j-1); allora a_i appartiene a PLSC, e ci muoviamo alla posizione (i-1,j-1). Se invece $a_i \neq b_j$, vuol dire che L(i,j) deve essere uguale ad almeno uno fra L(i,j-1) e L(i-1,j). Se L(i,j)=L(i-1,j), ci muoviamo in su di una riga, cioè andiamo alla posizione (i-1,j); se invece L(i,j)=L(i,j-1), ci muoviamo a sinistra di una colonna, cioè andiamo alla posizione (i,j-1). Seguendo questa regola, alla fine arriviamo all'angolo in alto a sinistra e a questo punto abbiamo trovato una PLSC. In figura 17 è mostrata la matrice dell'esempio 12.1 e il cammino corrispondente alla PLSC *cbba* è rappresentato in grassetto. Il procedimento è lineare.

Il programma `quickLength` ha complessità $\mathcal{O}(nm)$.

		c	b	a	b	a	c
	0	0	0	0	0	0	0
a	**0**	0	0	1	1	1	1
b	**0**	0	1	1	2	2	2
c	0	**1**	1	1	2	2	3
a	0	**1**	1	2	2	3	3
b	0	1	**2**	**2**	3	3	3
b	0	1	2	2	**3**	3	3
a	0	1	2	3	3	**4**	**4**

Figura 17: tabella L(i,j) per cbba

12.2 Algoritmi greedy

Un'altra metodologia di programmazione è quella *greedy* (ingorda): la soluzione ottima si ottiene mediante una sequenza di scelte. In ogni punto dell'algoritmo, viene scelta la strada che in quel momento sembra la migliore. Questa strategia euristica non sempre trova la soluzione ottima, ma è molto vantaggiosa.

Questa metodologia è utilizzabile quando la "scelta locale" è in accordo con la "scelta globale", quando cioè scegliendo ad ogni passo l'alternativa che sembra la migliore non si perdono alternative che potrebbero rivelarsi migliori nel seguito.

Le metodologie di programmazione dinamica e greedy affrontano i problemi in modo completamente diverso: mentre la programmazione dinamica risolve il problema in modo bottom-up, cioè prima risolvendo i sottoproblemi e poi prendendo le decisioni, la strategia greedy è una tecnica top-down, che decide il sottoproblema da risolvere e risolve solo quello, scartando gli altri sottoproblemi.

Come esempio di algoritmo greedy consideriamo i codici di Huffman, che sono una tecnica per la compressione di dati. Questi codici permettono in generale di risparmiare dal 20 al 90 per cento di spazio. Consideriamo un file di caratteri appartenenti ad un alfabeto e supponiamo di voler costruire una stringa di caratteri binari (codice) per rappresentarlo. Ipotizziamo di conoscere le frequenze di ogni singolo carattere del nostro alfabeto. Un codice a lunghezza fissa (in cui i caratteri sono codificati con stringhe binarie della stessa lunghezza) è meno efficiente di uno a lunghezza variabile, se assegnamo stringhe più corte a caratteri più frequenti e stringhe più lunghe a caratteri meno frequenti. Ad esempio se l'alfabeto è composto dai simboli {a,b,c,d} e le frequenze dei simboli sono quelle riportate in Figura 18(a), utilizzando 2 bit per la codifica di ogni singolo carattere, avremmo bisogno di $960 \cdot 2 = 1920$ bit per memorizzare il file. Ipotizzando invece di codificare i simboli come rappresentato in Figura 18(b) sarebbero necessari solamente $35 \cdot 3 + 253 \cdot 2 + 27 \cdot 3 + 645 \cdot 1 = 105 + 506 + 81 + 645 = 1337$ bit, ovvero solamente il 70% della dimensione originale del file. Il codice proposto ha la caratteristica si essere un codice *prefisso*, ovvero nessuna codifica di un simbolo è prefisso della codifica di un'altro simbolo. Grazie a questa proprietà, un decodificatore che conosca l'inizio di una sequenza codificata è in grado di separare i diversi simboli anche se nella sequenza non è presente alcun tipo di separatore. Il codice scelto può essere rappresentato come un albero binario ad archi etichettati, le cui foglie rappresentano i simboli dell'alfabeto, e le codifiche dei simboli si possono ottenere percorrendo l'abero dalla radice fino alle foglie, annotandosi le etichette degli archi percorsi. Ad esempio, l'albero associato al codice proposto per l'esempio precedente è visibile in Figura 18(c).

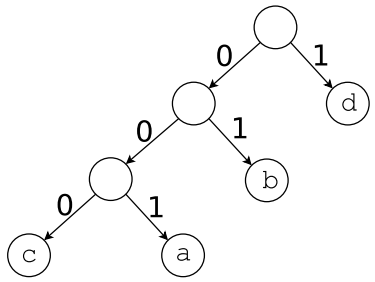
Illustriamo adesso l'algoritmo per ottenere un codice di Huffman, date le occorrenze dei singoli caratteri dell'alfabeto. Prima di tutto occorre memorizzare tali coppie carattere-frequenza in uno heap H ordinato inversamente (ovvero l'estrazione, con complessità costante, estrae l'elemento più piccolo dello heap). Per costruire l'albero, al posto di coppie carattere-frequenza, si memorizzano nello heap elementi

Simbolo	Occorrenze
a	35
b	253
c	27
d	645
Totale	960

(a)

Simbolo	Codice
a	001
b	01
c	000
d	1

(b)



(c)

Figura 18: Un esempio di codifica utilizzando il codice di Huffman: le frequenze dei simboli (a), un possibile codice (b), e l'albero associato (c)

del tipo:

```
  struct NodeH{
    char symbol;      // carattere alfabeto
    int freq;         // frequenza carattere
    NodeH* left; NodeH* right;
};
```

il campo `symbol` sarà significativo solamente per le foglie dell'albero che andremo a costruire. Supponiamo di avere n simboli da codificare e che lo heap sopracitato abbia nome `H` e contenga inizialmente n istanze di `NodeH` foglie e con i campi `symbol` e `freq` opportunamente inizializzati. La procedura seguente:

```
Node* huffman(Heap H, int n){
  for(int i=0; i< n-1; i++) {
    NodeH *t = new NodeH();
    t->left = H.extract();
    t->right = H.extract();
    t->freq= t->left->freq + t->right->freq;   //somma le radici
    H.insert(t);                               // inserimento nello heap
  }
  return H.extract();   //ritorna la radice dell'albero
}
```

provvede a costruire un codice di Huffman. Ad ogni iterazione del ciclo `for`, i due nodi minori nello heap vengono fusi a creare un nuovo nodo (che non è una foglia) e che contiene la somma delle frequenze. Tale nodo viene nuovamente inserito nello heap. Quindi, ad ogni iterazione il numero degli elementi dello heap diminuisce di uno. Alla fine, lo heap conterrà un solo elemento, corrispondente alla radice dell'albero del codice. Gli archi dell'albero ottenuto non sono etichettati ma possiamo facilmente associare una cifra binaria agli archi sinistri e l'altra agli archi destri. Valutando la complessità della procedura `huffman`, possiamo notare che ogni iterazione comporta due estrazioni in uno heap (che hanno complessita costante) e una inserzione (complessità logaritmica). Poiché vengono effettuate n-1 iterazioni, la complessità di `huffman` è $\mathcal{O}(n \log n)$. Si può dimostrare che la compressione ottima ottenibile con un codice per i caratteri può essere ottenuta con un codice prefisso. Inoltre, il codice ottimo è rappresentato da un

albero pienamente binario. Come si può notare, la procedura `huffman` costruisce un albero pienamente binario e quindi realizza un codice ottimo.

13 Grafi

Definizione 13.1 (grafo orientato)
Un grafo orientato *è una coppia* (N, A)*, dove* N *è un insieme di* nodi *e* $A \subseteq N \times N$ *è un insieme di* archi*, cioè coppie ordinate di nodi in* N*. Se* $(p, q) \in A$*, diciamo che* p *è predecessore di* q *e* q *è successore di* p*.*

Un grafo può essere rappresentato con un disegno, dove i nodi sono rappresentati con cerchi. Gli archi sono rappresentati con frecce: un arco (p, q) in A è rappresentato con una freccia che congiunge p con q, orientata da p a q.

La complessità degli algoritmi che manipolano grafi è in genere valutata in funzione del numero $n = |N|$ dei nodi e $m = |A|$ degli archi. Un grafo orientato con n nodi ha al massimo n^2 archi.

Dato un grafo orientato (N, A) un *cammino* è una sequenza di nodi $(n_1, \ldots, n_k), k \geq 1$ tale che esiste un arco da n_i a n_{i+1} per ogni $1 \leq i < k$. La lunghezza del cammino è data dal numero degli archi. Un *ciclo* è un cammino che comincia e finisce con lo stesso nodo $(n_1 = n_k)$, Un grafo è aciclico se non contiene cicli.

Un grafo orientato può essere rappresentato in memoria in due modi: con le *liste di adiacenza* e con le *matrici di adiacenza*. In entrambi i casi supponiamo che nodi siano numerati da 0 a $n - 1$. Nella realizzazione con liste di adiacenza, viene definito un array con dimensione uguale al numero dei nodi e ogni elemento dell'array rappresenta un nodo con i suoi successori. Ogni elemento dell'array è un puntatore ad una lista: l'elemento i-esimo punta alla lista che contiene i nodi successori del nodo i-esimo:

```
class Graph{
 struct Node {
   int nodeNumber;
   Node* next;
};
Node* graph [N];
 ...
};
```

Nella realizzazione con matrici di adiacenza, il grafo viene rappresentato con una matrice quadrata $n \times n$: l'elemento della matrice di indici i, j è 1 se c'è un arco dal nodo i al nodo j e 0 altrimenti.

```
class Graph{
int graph [N][N];
...
};
```

Dal punto di vista dell'occupazione di memoria, le matrici di adiacenza sono preferibili per grafi *densi*, cioè in cui il numero di archi è elevato, mentre per grafi *sparsi*, cioè con pochi archi ripetto ai nodi, è preferibile usare le liste di adiacenza.

Semplici operazioni su di un grafo orientato sono: determinare se esiste l'arco che connette due nodi, trovare tutti i successori o i predecessori di un nodo, trovare il *grado di ingresso* (*di uscita*) di un nodo, cioè il numero dei suoi predecessori (successori). La convenienza dell'una o dell'altra rappresentazione del grafo nell'effettuare queste operazioni varia con la maggiore o minore densità del grafo.

Esempio 13.1
La figura 19 mostra un grafo orientato, con le rispettive rappresentazioni con mostrano la sua rappresentazione rispettivamente con lista e matrice di adiacenza.
Il nodo 5 è successore di 4, 4 è predecessore di 5, 2 è successore di 3 e di 0, 0 è successore di 2, ecc.

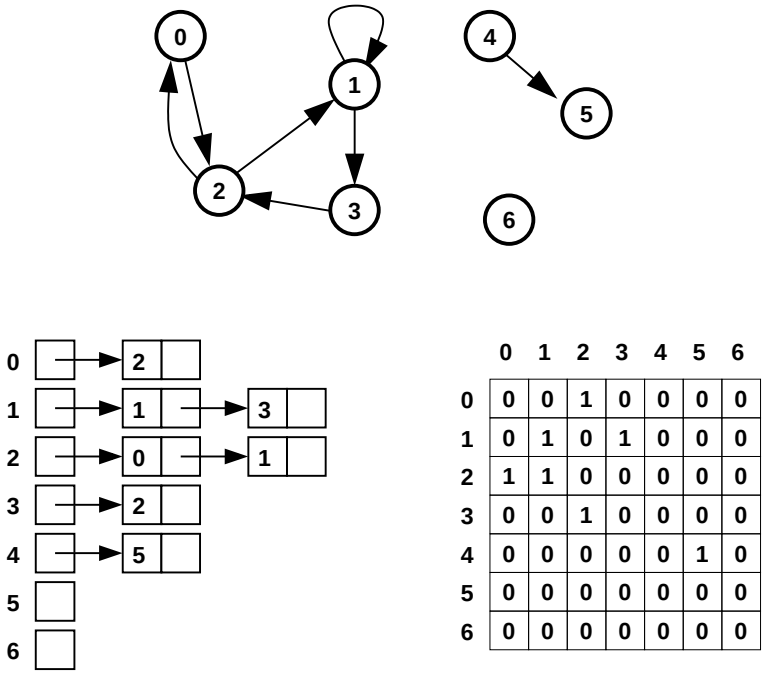


Figura 19: *Esempio di grafo orientato, con due diversi tipi di rappresentazione.*

Un esempio di cammino è il seguente: 1, 3, 2, 0. Un esempio di ciclo è 1, 3, 2.
Il nodo 2 ha grado 2 sia in ingresso che in uscita; anche il nodo 1 ha grado 2 sia in ingresso che in uscita.

Definizione 13.2 (grafo non orientato)
Un grafo non orientato *è una coppia* (N, A)*, dove* N *è un insieme di* nodi *e* A *un insieme di coppie non ordinate di nodi in* N*. Se* $(p, q) \in A$*,* $p \neq q$*, diciamo che* p *è* adiacente *a* q *e viceversa.*

Nei grafi non orientati quello che conta è la connessione tra nodi, e non la direzione di tale connessione. Nella rappresentazione grafica dei grafi non orientati, gli archi sono rappresentati con linee (senza orientamento): un arco (p, q) in A è rappresentato con una linea che congiunge p con q. Si noti come non ci sia mai un arco da un nodo a sé stesso. Un grafo non orientato ha al massimo $n(n-1)/2$ archi. La figura 20 mostra un grafo non orientato. Un cammino in un grafo non orientato è una sequenza di nodi $(n_1, \ldots, n_k), k \geq 1$ tale che n_i è adiacente a n_{i+1} per ogni i. Un ciclo è un cammino che inizia e termina con lo stesso nodo e non ha ripetizioni, eccettuato l'ultimo nodo. Ogni ciclo ha lunghezza maggiore o uguale a 3.

Un grafo non orientato è *connesso* se esiste un cammino fra due nodi qualsiasi del grafo.

Anche la complessità degli algoritmi che manipolano grafi non orientati è in genere valutata in funzione del numero dei nodi (n) e degli archi (m).

Un grafo non orientato può essere visto come un grafo orientato tale che, per ogni arco da un nodo p a un nodo q, ne esiste uno da q a p. La rappresentazione in memoria dei grafi non orientati può essere fatta con le matrici o con le liste di adiacenza tenendo conto di questa equivalenza. Naturalmente ogni arco del grafo non orientato sarà rappresentato due volte (la matrice di adiacenza è sempre simmetrica).

Spesso i grafi, orientati e non, hanno anche un'etichetta associata agli archi e/o un'etichetta associata ai nodi.

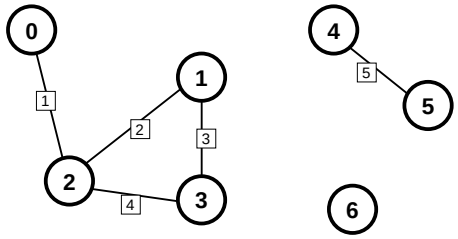


Figura 20: *Un grafo non orientato, con archi etichettati.*

Per rappresentare grafi con archi etichettati mediante matrici di adiacenza, l'elemento della matrice di indici (i, j) contiene l'etichetta dell'arco da i a j. Quindi il tipo della matrice, se le etichette degli archi sono del tipo `arcType`, sarà:

```
ArcType graph [N][N];
```

Naturalmente bisogna scegliere un opportuno valore di tipo `ArcType` per indicare l'assenza di arco tra due nodi. Se, ad esempio, i nodi sono città e gli archi sono etichettati con interi che rappresentano le distanze di possibili connessioni dirette, la mancanza di comunicazione diretta tra due città può essere rappresentata con `-1`. Con liste di adiacenza, si possono usare liste con elementi a tre campi, uno dei quali è l'etichetta dell'arco che va dal nodo iniziale della lista al successore rappresentato da quell'elemento di lista:

```
class Graph{
struct Node {
   int nodeNumber;
   ArcType arcLabel;
   Node* next;
};
Node* graph [N];
..
};
```

Per rappresentazione grafi con nodi etichettati, se le etichette nei nodi hanno tipo `NodeType`, basta aggiungere alla matrice di adiacenza o alle liste di adiacenza un array `NodeType nodeLabels [N]` tale che `nodeLabels[i]` contiene l'etichetta del nodo i.

Esempio 13.2
La figura 20 mostra un grafo non orientato, con etichette sugli archi. Il grafo è non connesso e contiene anche un nodo isolato.

13.1 Visita in profondità

Un'operazione sui grafi è la visita, cioè l'esame di ogni nodo del grafo. Consideriamo la *visita in profondità*: gli archi vengono esplorati a partire dall'ultimo nodo esaminato che abbia ancora degli archi non esplorati uscenti da esso. Possiamo usare un algoritmo simile a quello per la visita anticipata che abbiamo definito per gli alberi, partendo da un nodo qualsiasi e visitando i successori, stando però attenti a non entrare in un ciclo. Per fare questo, è sufficiente ricordare i nodi già esaminati e saltarli durante la visita. L'algoritmo informale è il seguente:

```
{ esamina il nodo; marca il nodo;
  visita i successori non marcati del nodo;
}
```

L'algoritmo può essere programmato nel modo seguente per un grafo con nodi etichettati se si usano liste di adiacenza. `mark` è un array inizializzato con tutti `0`.

```
 class Graph{
 struct Node {
    int nodeNumber;
    Node* next;
 };
 Node* graph [N];
 NodeType nodeLabels [N];
 int mark[N];
 void nodeVisit( int i) {
    mark[i]=1;
    cout << nodeLabels[i];
    Node* g; int j;
    for (g=graph[i]; g; g=g->next) {
       j=g->nodeNumber;
       if (!mark[j]) nodeVisit(j);
    }
 }
public:
 void depthVisit() {
    for (int i=0; i<N; i++) mark[i]=0;
    for (i=0; i<N; i++) if (! mark[i]) nodeVisit (i);
}
..
};
```

L'algoritmo `nodeVisit` tocca tutti i nodi se il grafo è non orientato e connesso. La sua complessità è $\mathcal{O}(m)$, se m è il numero di archi incontrati. Se il grafo è orientato, i nodi visitati sono solo quelli reggiungibili con un cammino dal nodo di partenza. L'algoritmo completo (`depthVisit`) è quello di chiamare `nodeVisit` su tutti i nodi del grafo. La complessità della visita in profondità è $\mathcal{O}(n) + \mathcal{O}(m)$. Se $m \geq n$, la complessità è $\mathcal{O}(m)$. Ad esempio, l'ordine di visita dei nodi del grafo di figura 19 è : 0, 2, 1, 3, 4, 5, 6.

Esiste anche una *visita in ampiezza* dei grafi, che non trattiamo qui, che consiste nell'esaminare, per ogni nodo, tutti i successori, prima di continuare la visita degli archi uscenti dai successori stessi.

13.2 Componenti connesse e minimo albero di copertura

Ogni grafo non orientato può essere suddiviso in *componenti connesse*. Ogni componente connessa è un sottografo connesso del grafo. Inoltre una componente connessa è massimale se nessun nodo appartenente ad essa è collegato ad un nodo appartenente ad un'altra componente.

Un algoritmo per trovare tutte le componenti connesse massimali di un grafo non orientato consiste nel numerare gli archi di G da 1 ad a e quindi costruire una catena di grafi G_i, $i \geq 0$, definita induttivamente nel modo seguente:

- G_0 consiste nei nodi di G senza nessun arco: ogni nodo è una diversa componente in G_0.
- G_i, con $i > 0$, si ottiene da G_{i-1} aggiungendo l'arco i solo se questo congiunge due nodi non connessi (appartenenti a due componenti connesse diverse) in G_{i-1}. Se l'arco i connette due nodi che appartengono alla stessa componente connessa di G_{i-1}, allora $G_i = G_{i-1}$.

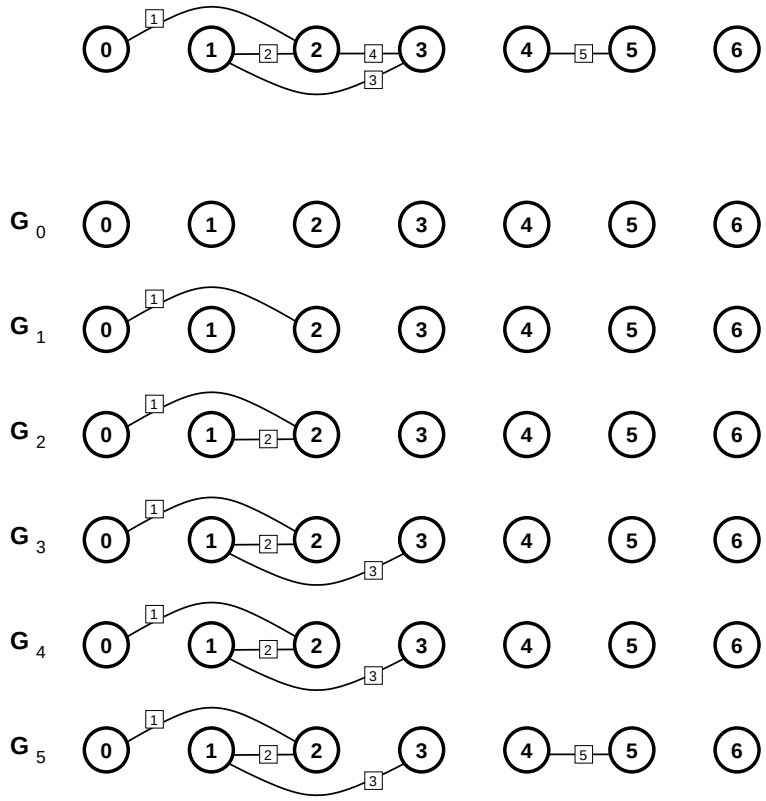


Figura 21: *Applicazione al grafo di figura 20 dell'algoritmo per trovare le componenti connesse.*

Esempio 13.3
La catena induttiva per il grafo di figura 20, se consideriamo l'ordine degli archi con etichetta crescente, è mostrata in figura 21.

Ogni grafo G_i della catena, compreso l'ultimo, è aciclico. Un grafo aciclico può essere visto come un albero in cui ogni nodo può essere la radice. Nel seguito parleremo di albero nel senso di grafo aciclico. Per realizzare l'algoritmo, prima dobbiamo costruire la sequenza degli archi, che saranno esaminati uno alla volta per costruire gli elementi della catena. Gli archi possono essere memorizzati in una lista semplice di tipo `Arc*`, dove il tipo `Arc` è definito con una struttura a quattro campi: i primi due contengono i nodi connessi dall'arco, il terzo l'etichetta dell'arco (se si tratta di archi etichettati), il quarto è il puntatore all'arco seguente:

```
struct Arc {
   int node1, node2;
   ArcType label;
   Arc* next;
};
```

L'algoritmo scorre la lista degli archi e ogni arco incontrato viene cancellato dalla lista se connette due nodi appartenenti alla stessa componente; se invece l'arco connette due nodi appartenenti a componenti diverse, l'arco viene lasciato nella lista e le due componenti connesse da quell'arco vengono fuse in un'unica componente. È necessario dare una opportuna rappresentazione delle componenti connesse del grafo, in modo da rendere piú efficiente il controllo per vedere se due nodi appartengono alla stessa componente

e la fusione fra componenti. Per far questo, possiamo rappresentare ogni singola componente con un albero generico, che ha come radice un nodo qualsiasi appartenente alla componente. L'intero insieme di componenti sarà una foresta di alberi. Per vedere se due nodi p e q appartengono alla stessa componente, basta vedere se appartengono allo stesso albero, cioè se la radice dell'albero cui appartiene p è uguale alla radice dell'albero cui appartiene q. Per fondere due alberi, si può inserire uno dei due alberi come nuovo sottoalbero della radice dell'altro. Per effettuare efficientemente l'operazione di ricerca della radice comune, conviene mantenere gli alberi con un basso numero di livelli. Inoltre, è opportuno rappresentare in memoria l'albero in modo tale che ogni nodo punti al proprio padre. Un nodo sarà quindi una struttura con un campo puntatore che punta al padre e un campo intero che indica il livello dell'albero di cui il nodo è radice (vedremo tra poco come usare questo campo). Per rappresentare la foresta utilizziamo un array di nodi. Ogni nodo è un puntatore al padre, oppure è NULL, se il nodo è una radice.

```
struct Node {
   Node* parent;
   int height;
};

Node* nodes [N];
```

La seguente funzione può essere usata per trovare la radice dell'albero a cui appartiene un nodo i.

```
Node* findRoot(int i) {
   Node* x=nodes[i];
   for(; x->parent; x=x->parent);
   return x;
}
```

La seguente strategia può essere utilizzata per fondere due alberi: per scegliere la radice da mantenere si confrontano i livelli dei rispettivi alberi e si sceglie di inserire la radice dell'albero meno profondo come figlio di quella dell'albero più profondo. Questo serve a mantenere più basso possibile il livello dell'albero. Per fare il confronto fra le profondità viene utilizzato il campo `height` delle due radici.

```
void merge(Node* r1, Node* r2) {
   Node* higher, lower;
   if (r1->height > r2->height) { higher=r1; lower=r2; }
   else { higher=r2; lower=r1; }
   lower->parent=higher;
   if (lower->height==higher->height) higher->height++;
}
```

Infine, il codice per trovare le componenti connesse di un albero, una volta costruita una lista `arcList` di tipo `Arc*`, contenente gli archi del grafo, è il seguente. La funzione `f` scorre la lista degli archi e, per ogni arco, lo cancella dalla lista se i nodi che esso connette appartengono alla stessa componente, lo lascia nella lista altrimenti e, in questo secondo caso, unisce le due componenti. Prima di chiamare la funzione `f`, viene inizializzato l'array `nodes` in modo che tutti i nodi siano radici.

Infine per calcolare l'albero minimo si procede come segue: si scorre la lista degli archi e, per ogni arco, lo si cancella dalla lista se i nodi che esso connette appartengono alla stessa componente, lo lascia nella lista altrimenti e, in questo secondo caso, unisce le due componenti.

```
void f(Arc* & l) {
   if (l) {
      Node* root1, root2;
      root1=findRoot(l->node1);
```

```
        root2=findRoot(l->node2);
        if (root1 == root2) {
            l=l->next;
            f(l);
        }
        else {
            merge(root1, root2);
            f(l->next);
        }
    }
}

for(int i=0; i<N; i++) {
    nodes[i]=new Node;
    nodes[i].parent=NULL;
    nodes[i].height=0;
}

f(arcList);
```

Si può dimostrare che, utilizzando l'algoritmo di fusione prima suggerito, il livello degli alberi che rappresentano le componenti è al massimo $\mathcal{O}(log\ n)$ con n numero dei nodi. l'algoritmo di fusione ha complessità $\mathcal{O}(1)$. Ricercare il padre di un nodo, se il livello dell'albero è $\mathcal{O}(log\ n)$, ha complessità $\mathcal{O}(log\ n)$, in quanto ogni iterazione del `for` risale di un livello. Quindi la relazione di ricorrenza di `f` in funzione di n e m è :

$$\begin{aligned} T(n,0) &= b \\ T(n,m) &= log\ n + T(n,m-1) \quad a > 0 \end{aligned}$$

da cui `f` ha complessità $\mathcal{O}(m\ log\ n)$. Quindi l'algoritmo per le componenti connesse ha complessità $\mathcal{O}(n) + \mathcal{O}(m\ log\ n)$. Se $m \geq n$, la complessità è $\mathcal{O}(m\ log\ n)$

Se consideriamo un grafo connesso, l'algoritmo per le componenti connesse costruisce quello che si chiama albero di copertura del grafo:

> dato un grafo G, un *albero di copertura* di G è un albero (grafo aciclico) cui appartengono tutti i nodi di G.

L'algoritmo per le componenti connesse può trovare alberi di copertura diversi a seconda dell'ordine in cui gli archi compaioni nella lista.

Esempio 13.4
Se nell'esempio 13.3 consideriamo gli archi nell' ordine 1, 4, 3, 2, 5, l'algoritmo troverà una componente connessa con i nodi 0, 1, 2, 3 e gli archi 1, 4 e 3 (diversamente da quanto indicato in figura 21).

Una specializzazione dell'algoritmo delle componenti connesse serve a risolvere il problema del *minimo albero di copertura*. Supponiamo di avere un grafo G con archi etichettati con etichette fra le quali esiste un ordinamento (per esempio interi o reali). Per semplicità supponiamo che l'albero sia connesso. Si tratta di trovare un albero di copertura che minimizzi la somma delle etichette dei suoi archi, cioè tale che un qualsiasi altro albero di copertura abbia la somma delle etichette degli archi maggiore o uguale. Per risolvere questo problema, basta applicare l'algoritmo per le componenti connesse *partendo da un ordinamento degli archi in ordine crescente*. Questo algoritmo è chiamato *algoritmo di Kruskal*. La sua complessità è data dalla complessità dell'ordinamento degli archi ($\mathcal{O}(m\ log\ m)$) più la complessità

dell'algoritmo delle componenti connesse ($\mathcal{O}(m\ log\ n)$), quindi $\mathcal{O}(m\ log\ m) + \mathcal{O}(m\ log\ n) = \mathcal{O}(m(log\ n + log\ m))$. Ma poiché $m \leq n^2$ (ci sono $n(n-1)/2$ coppie di nodi), abbiamo $log\ m \leq 2log\ n$ e quindi l'algoritmo ha complessità $\mathcal{O}(m\ log\ n)$.

13.3 Cammini minimi e algoritmo di Dijkstra

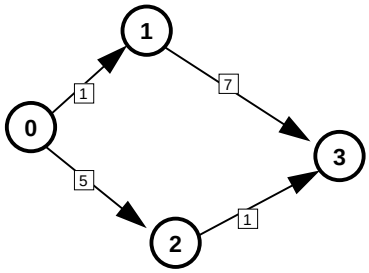
Supponiamo di avere un grafo, che può essere o no orientato, e tale che ad ogni arco è associata una "lunghezza". Per esempio il grafo potrebbe rappresentare una rete stradale: i nodi sono le città e gli archi rappresentano le strade fra una città e l'altra, con la loro lunghezza (oppure con il tempo che ci vuole per percorrerle). Un altro esempio significativo dell'uso di un grafo è quello di modellizzazione di una rete di calcolatori ad estensione geografica, in cui i nodi rappresentano "nodi effettivi" della rete (dove i dati vengono smistati) e gli archi le linee di connessione tra loro. In questo contesto, la "lunghezza" associata ad ogni arco potrebbe essere un valore che esprime la qualità della connessione, in termini di ritardo medio introdotto, probabilità di perdita di messaggi, banda residua disponibile, ecc; più alto è in suo valore, peggiore sarà la qualità. Generalmente, questo indice varia pesantemente al variare del tempo. Se si desidera spedire un messaggio da un nodo ad un altro nodo, si deve scegliere un percorso sulla rete dove farlo transitare (problema del *routing*).

In generale, un problema importante è quello di trovare il *cammino minimo* fra due nodi, cioè il cammino che parte da un nodo e arriva all'altro e che minimizza la somma delle lunghezze degli archi.

Dato un grafo (N, A) tale che le lunghezze di ogni arco è maggiore o uguale a 0, un algoritmo efficiente per trovare il cammino minimo da un nodo p_0 a un qualsiasi altro nodo è quello di Dijkstra. Per spiegare l'algoritmo, è utile utilizzare due insiemi disgiunti di nodi, che chiameremo S e Q, tali che in ogni momento $S \cup Q = N$: durante l'esecuzione dell'algoritmo, un nodo p appartiene all'insieme S se abbiamo già trovato il cammino minimo da p_0 a p e appartiene a Q se abbiamo soltanto una stima del cammino minimo da p_0 a p. Per ogni nodo p appartenente ad S o a Q, manteniamo la distanza minima da p_0 a p, che chiameremo $dist(p)$: se p appartiene ad S, $dist(p)$ è la distanza minima effettiva, se $p \in Q$, $dist(p)$ ed è la distanza minima stimata e può essere modificata. Per ogni nodo p, teniamo anche un puntatore, $pred(p)$, al nodo che lo precede nel cammino minimo (definitivo o stimato) da p_0 a p. L'algoritmo di Dijkstra, espresso in modo informale, è il seguente:

```
1   {Q=N;
2     per ogni nodo p diverso da p0, { dist(p)=infinito, pred(p)=vuoto; }
3     dist(p0)=0;
4     while (Q non vuoto) {
5       p=estrai da Q il nodo con minima distanza;
6       per ogni nodo q successore di p {
7         lpq=lunghezza dell'arco (p,q);
8         if (dist(p)+lpq < dist(q)) {
9           dist(q)=dist(p)+lpq;
10          pred(q)=p;
11          re-inserisci in Q il nodo q modificato;
12        }
13      }
14    }
15  }
```

Nella fase di inizializzazione (linee 1..3) l'insieme Q viene inizializzato con tutti i nodi: questo vuol dire che le distanze sono tutte stimate e non definitive. Tali distanze vengono poste a ∞ (maggiore di qualsiasi distanza) per i nodi diversi da quello di partenza e a 0 per p_0. Ad ogni iterazione del ciclo `while` (linee 4..14) il nodo con minima distanza stimata viene tolto da Q e considerato sistemato. Infatti si può dimostrare che per tale nodo la distanza stimata è anche quella effettiva. Successivamente, se p è il nome



S	Q	dist				pred			
		0	**1**	**2**	**3**	**0**	**1**	**2**	**3**
{}	{0, 1, 2, 3}	0	inft	inft	inft	-	-	-	-
{0}	{1, 2, 3}	0	1	5	inft	-	0	0	-
{0, 1}	{2, 3}	0	1	5	8	-	0	0	1
{0, 1, 2}	{3}	0	1	5	6	-	0	0	2
{0, 1, 2, 3}	{}	0	1	5	6	-	0	0	2

Figura 22: *Passi dell'algoritmo di Dijkstra applicato ad un grafo orientato ed etichettato.*

di tale nodo, vengono esaminati i successori di p, modificando le loro distanze stimate e predecessore nel modo seguente: per ogni nodo q successore di p, connesso a p dall'arco (p, q) con lunghezza lpq, se la distanza stimata di q è minore della distanza di p più lpq, allora non si modifica nulla; altrimenti vuol dire che abbiamo trovato un cammino (da p_0 a q), passante per p, di lunghezza più corta e quindi $dist(q)$ deve essere aggiornato e così pure il predecessore di q, che diventa p. Quando l'insieme Q è vuoto, $dist(p)$ contiene, per ogni nodo p, la sua minima distanza da p_0, mentre il minimo cammino da p_0 a p è ricostruibile utilizzando $pred(p)$. Quindi l'algoritmo di Dijkstra può essere utilizzato per trovare i cammini minimi da un nodo a tutti gli altri.

Esempio 13.5
La figura 22 mostra l'applicazione dell'algoritmo di Dijkstra a un semplice grafo orientato ed etichettato, prendendo il nodo 0 come partenza.

Calcoliamo ora complessità dell'algoritmo: abbiamo un tempo $\mathcal{O}(n)$ per l'inizializzazione più il tempo per il ciclo *while* che ha n iterazioni. Ogni iterazione del while ha un tempo $T(n)$ per l'estrazione del nodo con minima distanza da Q (linea 6) più il tempo per l'esame dei nodi adiacenti a p (ciclo interno linee 6..11). Quest'ultimo ha un numero di iterazioni pari a m_p, se m_p è il numero di successori di p, e ogni iterazione ha un tempo $R(n)$ per l'istruzione alla linea 11(`re-inserisci in Q il nodo q modificato;`). Quindi il tempo del ciclo interno è $\mathcal{O}(m_p R(n))$. Per semplificare il conto, supponiamo che da ogni nodo esca lo stesso numero di archi: abbiamo quindi $m_p = \frac{m}{n}$ e il tempo del ciclo interno è $\mathcal{O}(\frac{m}{n} R(n))$. Quindi il tempo del ciclo `while`, trascurando i tempi costanti, è:

$$\mathcal{O}(n)\ (T(n) + \tfrac{m}{n} R(n)) = \mathcal{O}(n\ T(n)) + \mathcal{O}(m\ R(n))$$

che è anche al tempo dell'intero programma. Per completare il calcolo, dobbiamo conoscere la complessità di $T(n)$ e di $R(n)$, che dipendono dalla realizzazione dell'insieme Q. La realizzazione più efficiente di Q è uno heap con ordinamento contrario rispetto a quello considerato nel capitolo 9, tale cioè che ogni

nodo è minore o uguale dei suoi successori. Utilizzando uno heap abbiamo che $T(n)$ è il tempo per estrarre dallo heap e quindi è $\mathcal{O}(log\ n)$; $R(n)$ è il tempo per modificare la posizione di q nello heap e anche questa operazione può essere implmentata con tempo $\mathcal{O}(log\ n)$. Quindi il tempo del programma è $\mathcal{O}(n\ log\ n) + \mathcal{O}(m\ log\ n)$, che è uguale a $\mathcal{O}(m\ log\ n)$ se $m \geq n$.

Sia l'algoritmo di Kruskal che quello di Dijkstra sono algoritmi basati sulla metodologia di programmazione *greedy*.

14 NP-completezza

Consideriamo i seguenti problemi decisionali:

1. *Commesso viaggiatore*: Date n città, le distanze tra esse e un intero k, è possibile partire da una città, attraversare ogni città esattamente una volta e tornare alla città di partenza, percorrendo una distanza complessiva non superiore a k?

2. *colorazione mappe*: data una mappa, è possibile colorare le regioni con al più k colori in modo che ogni regione sia colorata in modo diverso dalle regioni adiacenti?

3. *soddisfattibilità di una formula logica*: data una espressione booleana, esiste una assegnazione di valori di verità alle variabili che compaiono nella formula che rende l'espressione vera? Per esempio, $(x\ and\ y)\ or\ z$ è soddisfattibile, mentre $x\ and\ not\ x$ non lo è .

Per tutti questi problemi sono stati definiti soltanto algoritmi che sostanzialmente esaminano tutte le situazioni possibili. Poiché il numero delle situazioni possibili cresce in modo esponenziale (con il numero delle città, delle regioni, delle variabili che compaiono in una formula), tali algoritmi hanno complessità esponenziale e i problemi relativi sono quindi detti "intrattabili". Per esempio, consideriamo il problema P_S della soddisfattibilità di una formula logica. Un possibile programma ricorsivo che lo risolve è il seguente, dove `value(f,a)` è una funzione che calcola il valore della formula `f` con gli assegnamenti alle variabili contenuti nell'array `a` (prima chiamata `sat(f,a,0,n)`):

```
int sat(Formula f, int* a, int i, int n) {
   if (i>n-1) return value(f,a);
   else {
      a[i]=1;
      if (sat(f,a,i+1,n)) return 1;
      else {
         a[i]=0;
         return sat(f,a,i+1,n);
      }
   }
}
```

Il programma ha complessità esponenziale: infatti calcola la formula per tutti i possibili valori delle variabili, cioè 2^n.

Per tutti i problemi decisionali elencati sopra è un problema aperto se sia possibile trovare un algoritmo polinomiale che li risolve. Gli algoritmi conosciuti non riescono a sfruttare alcuna proprietà del problema e quindi procedono per enumerazione, cioè elencando tutti i casi possibili. Questo vuol dire che questi algoritmi esplorano tutto l'albero di decisione, cha ha un numero di foglie che cresce esponenzialmente con i dati, escludendone al più una parte che ha un peso irrilevante sulla complessità.

Un *algoritmo non deterministico* è un algoritmo che, posto di fronte a una decisione, ha la "virtù magica" di scegliere sempre la strada giusta. In altre parole, è come se l'algoritmo, avendo più alternative da poter seguire, le segua tutte contemporaneamente e si arresti appena una delle alternative raggiunge

una soluzione. Gli algoritmi nondeterministici hanno un interesse puramente teorico, non sono cioè utilizzabili in pratica. Per definirli in modo formale, supponiamo di avere a disposizione le seguenti espressioni, tutte eseguibili in tempo costante;

- `choice(I)`: sceglie arbitrariamente un elemento dell'insieme I;
- `success`: blocca la computazione in uno stato di "successo";
- `failure`: blocca la computazione in uno stato di "fallimento".

Esempio 14.1

- *Un algoritmo nondeterministico con complessità $\mathcal{O}(n)$ per il problema della soddisfattibilità è il seguente:*

```
int nsat(Formula f,int *a,int n) {
   for (int i=0; i < n; i++)
      a[i]=choice({0,1});
   if (value(f,a))
      return success;
   else
      return failure;
}
```

- *Un algoritmo nondeterministico con complessità $\mathcal{O}(1)$ per cercare un elemento in un insieme memorizzato in array è il seguente:*

```
int nsearch(int* a, int n, int x) {
   int i=choice({0..n-1});
   if (a[i]==x)
      return success;
   else
      return failure;
}
```

- *Un algoritmo nondeterministico con complessità $\mathcal{O}(n)$ per ordinare un insieme è il seguente (dove* `ordinato` *è una funzione che in tempo lineare decide se un array è ordinato):*

```
void nsort(int* a, int n) {
   int b [n];
   for (int i=0; i<n; i++)
      b[i]=a[i];
   for (int i=0; i<n; i++)
      a[i]=b[choice({0..n-1})];
   if (ordinato(a))
      return success;
   else
      return failure;
}
```

Ad ogni algoritmo nondeterministico corrisponde un albero di decisione, dove ad ogni nodo corrisponde una scelta (`choice`). Le foglie sono etichettate o con `success` o con `failure`. Per ogni algoritmo nondeterministico ne esiste uno deterministico che lo *simula*, esplorando lo spazio delle soluzioni, cioè le

foglie, fino a trovare un successo. Per esempio, l'algoritmo di ricerca in un array del capitolo 11 (con complessità lineare) simula l'algoritmo nondeterministico `nsearch` definito sopra. Se le foglie sono in numero esponenziale, l'algoritmo deterministico avrà complessità esponenziale. Per esempio, un algoritmo che simula `nsort` ha complessità esponenziale. In certi casi sono stati trovati degli algoritmi polinomiali dove la simulazione dell'algoritmo nondeterministico è esponenziale, come nel caso dell'ordinamento (*merge-sort* o *quicksort*). Invece per i problemi 1,2,3 elencati sopra non si conoscono algoritmi deterministici polinomiali, mentre esistono algoritmi nondeteministici polinomiali che li risolvono.

Definizione 14.1
Chiamiamo $\mathcal{P}$ l'insieme di tutti i problemi decisionali risolubili in tempo polinomiale con un algoritmo deterministico.
Chiamiamo $\mathcal{NP}$ l'insieme di tutti i problemi decisionali risolubili in tempo polinomiale con un algoritmo nondeterministico.

Appartengono a $\mathcal{P}$ il problema della ricerca di un elemento in un insieme e dell'ordinamento di un insieme. Ad $\mathcal{NP}$ appartengono anche i problemi 1,2,3 definiti sopra. Ovviamente vale che $\mathcal{P} \subseteq \mathcal{NP}$, in quanto gli algoritmi deterministici sono casi particolari di algoritmi nondeterministici. Tuttavia non è stato dimostrato se valga o no l'inclusione stretta, cioè se $\mathcal{P} = \mathcal{NP}$, e questo è a tuttoggi un problema aperto.

Tuttavia esistono alcuni risultati, che affermano che alcuni problemi hanno la stessa complessità e risolvendone uno in tempo polinomiale si risolverebbero tutti in tempo polinomiale. Introduciamo il concetto di *riducibilità*.

Definizione 14.2
Un problema P_1 si riduce in tempo polinomiale a un problema P_2 ($P_1 \preceq P_2$) se ogni soluzione di P_1 può ottenersi deterministicamente in tempo polinomiale da una soluzione di P_2.

In altre parole, se $P_1 \preceq P_2$, questo vuol dire che esiste un algoritmo deterministico polinomiale che trasforma un qualsiasi dato di ingresso D_1 di P_1 in un dato di ingresso D_2 di P_2, in modo che P_1 abbia risposta affermativa su D_1 se e solo se P_2 ha risposta affermativa su D_2. Come conseguenza, se P_2 ha un algoritmo A_2 che lo risolve in tempo polinomiale, anche P_1 può essere risolto in tempo polinomiale: basta trasformare (in tempo polinomiale) l'ingresso di P_1 nel corrispondente ingresso di P_2 e quindi applicare A_2.

Utilizzando il concetto di riducibilità è possibile definire classi di problemi riducibili l'uno all'altro, e quindi equivalenti dal punto di vista della complessità (purchè la complessità della riduzione non superi la complessità dei singoli problemi).

Teorema di Cook. Per qualsiasi problema $P \in \mathcal{NP}$ vale che $P \preceq P_S$.

Quindi P_S può essere considerato in un certo senso come "il più difficile" fra i problemi appartenenti ad $\mathcal{NP}$ e può essere impiegato per risolvere uno qualsiasi di questi problemi. Inoltre, se si trovasse un algoritmo polinomiale deterministico per risolvere P_S, qualunque problema in $\mathcal{NP}$ sarebbe risolubile in modo polinomiale deterministico (avremmo $\mathcal{P} = \mathcal{NP}$).

Definizione 14.3
Un problema P è detto NP-completo se

- $P \in \mathcal{NP}$; *e*
- $P_S \preceq P$

Poiché tutti i problemi in $\mathcal{NP}$ sono riducibili a P_S per il teorema di Cook, tutti i problemi NP-completi si riducono scambievolmente l'un l'altro. Se si scoprisse un algoritmo polinomiale deterministico per risolvere uno qualsiasi di questi problemi, avremmo dimostrato che $\mathcal{P} = \mathcal{NP}$.

Per dimostrare che un problema P è NP-completo si deve prima dimostrare che appartiene ad $\mathcal{NP}$ e poi che un qualsiasi problema notoriamente NP-completo si riduce a P (la riducibilità è transitiva). Il primo passo viene effettuato individuando un algoritmo polinomiale nondeterministico per risolvere P, il secondo scegliendo un problema NP-completo che sia facilmente riducibile a P. Si può dimostare che i problemi 1,2,3 elencati sopra sono tutti NP-completi.

Le tecniche di progetto di algoritmi (*backtracking*, *programmazione dinamica*, *branch and bound*, etc.) possono essere applicate per risolvere problemi intrattabili, con lo scopo o di diminuire il tempo di esecuzione, che comunque rimane sempre esponenziale, o di approssimare una soluzione con tempo polinomiale.

Si noti che ad ogni problema decisionale corrisponde un problema *risolvente*, che, oltre a richiedere l'esistenza di una soluzione, vuole anche trovarne una, e spesso un problema di *ottimizzazione*, che vuole anche trovare una soluzione ottima. Per esempio, nel caso di P_S, il problema risolvente corrispondente consiste nel trovare, se esiste, un assegnamento alle variabili che renda vera la formula, mentre nel caso del commesso viaggiatore, si può voler trovare un percorso minimo. Ovviamente ogni problema decisionale si riduce al corrispondente problema risolvente o di ottimizzazione. Questo vuol dire che il problema risolvente o di ottimizzazione è almeno altrettanto difficile del problema decisionale. Di conseguenza, se il problema decisionale è NP-completo, anche il corrispondente problema risolvente è intrattabile.

Consideriamo il problema della generazione di tutte le permutazioni di un insieme dato. Esso non appartengono ad $\mathcal{NP}$ ed infatti è inerentemente esponenziale. Infatti per risolverli non può esistere alcun algoritmo nondeterministico polinomiale: le permutazioni di n elementi sono $n!$ e l'algoritmo deve generarle tutte.

A Principi di induzione

A.1 Induzione naturale

Sia ϕ una proprietà sui naturali.

Se

- **base**: ϕ vale per 0 e
- **passo induttivo**: per ogni naturale n è vero che:
  Ipotesi: ϕ vale per n; Tesi: ϕ vale per $(n+1)$

allora

ϕ vale per tutti i naturali

A.2 Induzione completa

Sia ϕ una proprietà sui naturali.

Se

- **base**: ϕ vale per 0 e
- **passo induttivo**: per ogni naturale n è vero che:
  Ipotesi: ϕ vale per tutti i naturali $\leq n$: Tesi: ϕ vale per $(n+1)$

allora

ϕ vale per tutti i naturali

I principi di induzione naturale e completa si possono anche applicare prendendo come base un numero $n_0 > 0$ e dimostrando la proprietà per tutti i numeri maggiori o uguali di n_0.

A.3 Induzione ben fondata

Un insieme ordinato $(S, <)$ è un insieme S più una relazione di ordinamento o precedenza (transitiva) $<$. I minimali sono elementi che non hanno alcun precedente. Una catena decrescente è una sequenza $x_1 > x_2 > x_3 ...$ Un insieme ben fondato è un insieme ordinato che non ha catene decrescenti infinite.

Sia ϕ una proprietà su un insieme ben fondato $(S, <)$.

Se

- **base**: ϕ vale per i minimali di S e
- **passo induttivo**: per ogni elemento $s \in S$ è vero che:
  Ipotesi: ϕ vale per tutti gli elementi minori di s: Tesi: ϕ vale per s

allora

ϕ vale per tutti gli elementi di S.

B Implementazione di una lista con una classe C++

Questa appendice contiene l'implementazione di una classe modello che realizza una lista semplice. Per illustrare il metodo della ricorsione, molti metodi sono stati implementati ricorsivamente. Il codice è commentato in maniera che, utilizzando il sistema di documentazione Doxygen[2] (secondo lo stile JavaDoc).

```
#ifndef LISTABASE_H
#define LISTABASE_H

#include <iostream>

// forward declarations
template <class T> class ListaBase;
template <class T>
std::ostream& operator<<(std::ostream&, const ListaBase<T>&);

/**
 * Classe modello di una lista semplice. Molti metodi sono implementati
 * utilizzando la ricorsione.
 *
 */
template <class T>
class ListaBase{
protected:

  /**
   * Un elemento della lista.
   */
  struct Elem{

    T info;              /**< Il contenuto dell'elemento */
    Elem* next;          /**< Il puntatore all'elemento successivo */

    /**
     * Costruttore di un elemento.
     * @param _info l'informazione da inserire nell'elemento
     */
    Elem(const T& _info) : info(_info), next(NULL){};
  };

  Elem* testa;           /**< Il puntatore alla testa della lista */
  static void deleteList(Elem*&);
  static void inserisci(const T&, Elem*);
  static void stampa(std::ostream&, Elem*);
  static bool belongs(Elem*, const T&);
  static void deletex(Elem*&, const T&);
public:
  /**
   * Costruttore di default
```

[2] Vedi http://www.stack.nl/~dimitri/doxygen/

```
   */
  ListaBase() : testa(NULL){};

  /**
   * Distruttore di default. Richiama la funzione statica deleteList()
   */
  virtual ~ListaBase() {deleteList(testa);};

  /**
   * Controlla se un elemento con un certo valore appartiene alla
   * lista. Utilizza l'operatore di confronto == per le uguaglianze.
   *
   * @param x il valore da cercare
   *
   * @return vero se il valore cercato appartiene alla lista,
   * falso altrimenti
   */
  virtual bool belongs(const T& x) const { return belongs(testa,x);}

  virtual void inserisciTesta(const T&);
  virtual void inserisciCoda(const T&);
  virtual void cancellax(const T&);
  friend std::ostream& operator<< <>(std::ostream&,const ListaBase<T>&);
};

/**
 * Elimina da una sottolista tutti gli elementi con un certo valore
 *
 * @param e la testa della sottolista
 * @param x il valore da cercare
 */
template <class T>
void ListaBase<T>::deletex(Elem*& e, const T& x) {
  if (!e) return;
  if (e->info == x) {
    Elem* e1 = e ;
    e = e -> next;
    delete e1;
    deletex(e,x);
  }
  else
    deletex(e->next,x);
}

/**
 * Elimina dalla lista tutti gli elementi con un certo valore
 *
 * @param x il valore da cercare
 */
template <class T>
```

```
void ListaBase<T>::cancellax(const T& x) {
  deletex(testa,x);
}

/**
 * Cancella tutti gli elementi della sottolista, rendendola vuota
 *
 * @param l la sottolista da cancellare
 */
template <class T>
void ListaBase<T>::deleteList(Elem*& l) {
  if (l) {
    deleteList(l->next);
    delete l;
    l = NULL;
  }
}

/**
 * Inserisce un nuovo elemento in una sottolista, posizionandolo come
 * secondo elemento. Non fa niente se la sottolista e' vuota.
 *
 * @param i l'informazione contenuta nel nuovo elemento
 * @param e la sottolista in cui inserire l'elemento
 */
template <class T>
void ListaBase<T>::inserisci(const T& i, Elem* e) {
  if(e){
    Elem* e_new = new Elem(i);
    e_new->next = e->next;
    e->next = e_new;
  }
}

/**
 * Inserisce un nuovo elemento in testa alla lista
 *
 * @param i l'informazione contenuta nel nuovo elemento
 */
template <class T>
void ListaBase<T>::inserisciTesta(const T& i) {
  Elem* e = new Elem(i);
  e->next=testa;
  testa=e;
}

/**
 * Inserisce un nuovo elemento in coda alla lista
 *
 * @param i l'informazione contenuta nel nuovo elemento
```

```
 */
template <class T>
void ListaBase<T>::inserisciCoda(const T& i) {
  if (testa){
    Elem* e = testa;
    while (e->next) e = e->next;
    inserisci(i,e);
  } else { // inserimento in testa!
    inserisciTesta(i);
  }
}

/**
 * Stampa tutti gli elementi di una sottolista
 *
 * @param os lo stream di uscita su cui stampare
 * @param l la sottolista da stampare
 */
template <class T>
void ListaBase<T>::stampa(std::ostream& os, Elem* l) {
  if (l) {
    os << l->info << " ";
    stampa(os,l->next);
  }
}

/**
 * Ridefinizione dell'operatore << per la classe ListaBase.
 * Utilizza la funzione stampa().
 *
 * @param os lo stream di uscita
 * @param l la lista
 *
 * @return un riferimento allo stream di uscita modificato
 */
template <class T>
std::ostream& operator<<(std::ostream& os, const ListaBase<T>& l){
  os << "[";
  l.stampa(os,l.testa);
  os << "]";
  return os;
}


/**
 * Controlla se un elemento con un certo valore appartiene ad una
 * sottolista. Utilizza l'operatore di confronto == per le uguaglianze.
 *
 * @param l la sottolista
 * @param x il valore da cercare
```

```
 *
 * @return vero se il valore cercato appartiene alla sottolista,
 * falso altrimenti
 */
template <class T>
bool ListaBase<T>::belongs(Elem* l, const T& x) {
  if (!l) return false;
  if (l->info == x) return true;
  return belongs(l->next,x);
}


#endif
```

C Implementazione di albero binario con una classe C++

```
#include <iostream>
   class BinTree {

      struct Node {
         int label;
         Node *left, *right;
         Node(int info) {
            label = info;
            left = right = NULL;
         }
      };
      Node *root;

      void deleteNode(int, Node*&);
      Node* findNode(int, Node*);
      void preOrder(Node*);
      void inOrder(Node*);
      void postOrder(Node*);
      void delTree(Node*&);
      void printTree(Node*);
   public:
      BinTree() { root = NULL; };
      ~BinTree(){ delTree(root); };
      int insert(int, int, char);
      void cancel(int x) { deleteNode(x, root); };
      int find(int x) { return ! findNode(x, root); };
      void pre() { preOrder(root); };
      void post(){ postOrder(root); };
      void in()  { inOrder(root); };
      void print() { printTree(root); };

   };
```

```
void BinTree::deleteNode(int x, Node* &tree) {
   if (tree) {
      if(tree->label == x) {
         delTree(tree);
         tree = NULL;
         return;
      }
      deleteNode(x, tree->left);
      deleteNode(x, tree->right);
   }
}

BinTree::Node* BinTree::findNode (int n, Node *tree) {
   if (!tree)     return NULL;
   if (tree->label == n) return tree;
   Node *a = findNode(n, tree->left);
   if (a) return a;
   else   return findNode(n, tree->right);
}

void BinTree::preOrder(Node *tree) {
   if (tree) {
      cout << "(␣" << tree->label << '␣';
      preOrder(tree->left);
      preOrder(tree->right);
      cout << ")";
   }
}

void BinTree::inOrder(Node *tree) {
   if (tree) {
      cout  << "(";
      inOrder(tree->left);
      cout << "␣" << tree->label << '␣';
      inOrder(tree->right);
      cout << ")";
   }
}

void BinTree::postOrder(Node *tree) {
   if (tree) {
      cout  << "(";
      postOrder(tree->left);
      postOrder(tree->right);
      cout << "␣" << tree->label << "␣)";
   }
}

void BinTree::delTree(Node* &tree) {
   if (tree) {
      delTree(tree->left);
```

```
        delTree(tree->right);
        delete tree;
        tree = NULL;
    }
}

void BinTree::printTree(Node *tree) {
    static int level = 0;
    if (tree) {
      for(int i = 0; i<level; i++) cout << "␣␣";
      cout << tree->label << '\n';
      level++;
      printTree(tree->left);
      printTree(tree->right);
      level--;
    }
}

int BinTree::insert(int son, int father, char c) {
    if (!root) {
        root = new Node(son);
        return 1;
    }
    Node *a = findNode(father,root);
    if (!a) return 0;
    if (c == 'l' && !a->left) {
        a->left = new Node(son);
        return 1;
    }
    if (c == 'r' && !a->right) {
        a->right = new Node(son);
        return 1;
    }
    return 0;
}
```

D Gestione di alberi con gerarchie di classi

I vari tipi di alberi si differenziano per alcune caratteristiche specifiche, ma presentano comunque analogie significative sia riguardo alla loro rappresentazione in termini di strutture dati, sia riguardo alle funzioni per la loro gestione. Questa considerazione suggerisce la possibilità di sfruttare alcune proprietà tipiche della programmazione ad oggetti per implementare una libreria di classi dedicate alla rappresentazione e alla gestione dei vari tipi di alberi. In particolare, ci riferiremo alle seguenti caratteristiche del linguaggio C++, con cui andremo a scrivere questa libreria:

- utilizzo di *template*, per generalizzare il tipo di dato trattato da ciascuna specifica istanza di albero;
- utilizzo della *derivazione* per organizzare una gerarchia di classi che permetta il riuso del codice;
- utilizzo del meccanismo di *overriding*, per dare un'implementazione specifica a un metodo di una classe albero derivata, diversa da quella del metodo presente nella classe base;

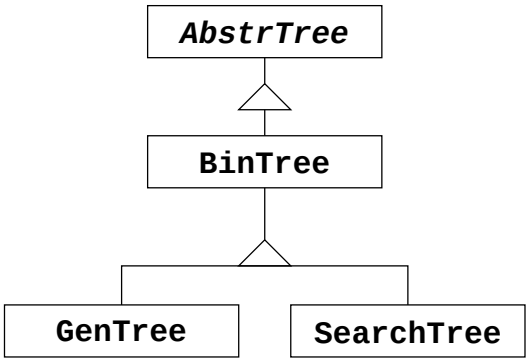


Figura 23: *Class diagram di una libreria di classi C++ per l'implementazione di vari tipi di albero.*

- modifica della *visibilità di un metodo ereditato*, in modo da non renderlo più disponibile in una sotto-classe.

Considerando le classi relative a diversi tipi di alberi (e soprattutto le loro funzionalità), si può pensare di organizzarle in una gerarchia come quella riportata in figura. In tale gerarchia è prevista una classe base astratta, che chiameremo `AbstrTree`. Essa deve contenere le strutture dati per l'implementazione di un albero qualsiasi, i metodi pubblici comuni a qualsiasi tipo di albero e relativi alla sua gestione, i metodi privati correntemente usati come funzioni locali dai metodi pubblici. `AbstrTree` è una classe astratta, in quanto non solo molti dei suoi metodi sono virtuali, ma alcuni di essi sono anche virtuali puri, cioè soltanto dichiarati, e privi di definizione.
Da `AbstrTree` si deriva la classe `BinTree`, che implementa un albero binario classico. `BinTree` ha due classi figlie: `GenTree` (albero generico) e `SearchTree` (albero binario di ricerca). Occorre precisare che in `GenTree` si utilizza una rappresentazione dell'albero del tipo *figlio-fratello*, che ci permette di ereditare direttamente la struttura base del nodo, con soltanto due puntatori a nodo. In figura 23 è riportato il class diagram della libreria che andiamo ad analizzare.

D.1 La classe `AbstrTree`

Riportiamo di seguito il codice della classe `AbstrTree`, contenuta nel file AbstrTree.h.

```
#ifndef _ABSTRTREE
#define _ABSTRTREE

#include <iostream>

   template <class LabelType>
   class AbstrTree {

   protected:
      struct Node {
         LabelType label;
         Node *left, *right;
         Node(LabelType info) {
            label = info;
            left = right = NULL;
```

```
        }
    };
    Node *root;

    virtual void deleteNode(LabelType, Node*&) = 0;
    virtual Node* findNode(LabelType, Node*);
    void preOrder(Node*);
    void inOrder(Node*);
    void postOrder(Node*);
    virtual void delTree(Node*&);
    virtual void printTree(Node*) = 0;
    int createNode(LabelType, Node*&);

public:
    AbstrTree() { root = NULL;};
    virtual ~AbstrTree() { delTree(root); };

    virtual int insert(LabelType son, LabelType father) = 0;
    int find(LabelType x) { return (int)findNode(x, root); };
    void cancel(LabelType x) { deleteNode(x, root); };
    virtual void pre()  { preOrder(root); };
    virtual void post() { postOrder(root); };
    virtual void in()   { inOrder(root); };
    void print() { printTree(root); };
};
// definitions

template <class LabelType>
AbstrTree<LabelType>::Node*
AbstrTree<LabelType>::findNode (LabelType x, Node *tree) {
    if (!tree) return NULL;
    if (tree->label == x) return tree;
    Node* a = findNode(x, tree->left);
    if (a) return a;
    else return findNode(x, tree->right);
}

template <class LabelType>
void AbstrTree<LabelType>::preOrder(Node *tree) {
    if (tree) {
        cout << "(␣" << tree->label << '␣';
        preOrder(tree->left);
        preOrder(tree->right);
        cout << ")";
    }
}

template <class LabelType>
void AbstrTree<LabelType>::inOrder(Node *tree) {
    if (tree) {
```

```
            cout  << "(";
            inOrder(tree->left);
            cout << "␣" << tree->label << '␣';
            inOrder(tree->right);
            cout << ")";
        }
    }

    template <class LabelType>
    void AbstrTree<LabelType>::postOrder(Node *tree) {
        if (tree) {
            cout  << "(";
            postOrder(tree->left);
            postOrder(tree->right);
            cout << "␣" << tree->label << "␣)";
        }
    }

    template <class LabelType>
    void AbstrTree<LabelType>::delTree(Node* &tree) {
        if (tree) {
            delTree(tree->left);
            delTree(tree->right);
            delete tree;
            tree = NULL;
        }
    }

    template <class LabelType>
    int AbstrTree<LabelType>::createNode(LabelType x, Node* &tree) {
        if (!tree) {
            tree = new Node(x);
            return 1;
        }
        return 0;
    }
#endif
```

In questa classe si stabilisce innanzitutto la struttura dati fondamentale che costituisce un albero: si tratta della `struct Node`, che contiene un campo informazione (`label`) di tipo generico `LabelType`, e due puntatori a `Node`. Tale struttura è collocata nella porzione `protected` della classe, in quanto dovrà essere utilizzata dalle sotto-classi. Il costruttore della `struct Node` permette di inizializzare il contenuto informativo del nodo, e assegna il valore `NULL` ai due puntatori.
Il tipo del contenuto informativo trattato in ciascuna istanza di albero è del tutto generico, e deve essere specificato a tempo di compilazione. Per poter utilizzare un tipo generico per il campo `label`, si dichiara `AbstrTree` come classe *template*, specificando il suo argomento formale come `<class LabelType>`. Nelle definizioni non *inline* dei vari metodi della classe, occorre fornire la specifica classe di appartenenza (con l'operatore `::`) dettagliando il suo tipo generico (ovvero ricorrendo ancora a *template*).
I vari metodi per la gestione dell'albero sono dichiarati nella parte pubblica della classe, ma spesso fanno direttamente riferimento ad alcuni analoghi metodi nella parte protetta. Molti di questi ultimi hanno una struttura ricorsiva. I metodi fondamentali, usati estensivamente in tutta la gerarchia, sono definiti

direttamente in questa classe. Altri metodi *sono virtuali puri* (o *astratti*), in quanto necessitano di una implementazione che varia distintamente a seconda del tipo di albero.
Sono presenti metodi pubblici per effettuare i vari tipi di visita, che richiamano i corrispondenti metodi protetti ricorsivi.

D.2 La classe `BinTree`

Riportiamo di seguito il codice della classe `BinTree`, contenuta nel file BinTree.h.

```
#ifndef _BINTREE
#define _BINTREE

#include "AbstrTree.h"

   template <class LabelType>
   class BinTree : public AbstrTree<LabelType> {

   protected:
      virtual void deleteNode(LabelType, Node*&);
      virtual void printTree(Node *tree);

   public:
      BinTree(): AbstrTree<LabelType>() {};
      virtual ~BinTree() {};
      virtual int insert(LabelType son, LabelType father) {
          return insert(son, father, 'r'); }; // right-inserted by default
      int insert(LabelType, LabelType, char);
   };

//definitions

   template <class LabelType>
   void BinTree<LabelType>::deleteNode(LabelType x, Node* &tree) {
      if (tree) {
         if(tree->label == x) {
            delTree(tree);
            tree = NULL;
            return;
         }
         deleteNode(x, tree->left);
         deleteNode(x, tree->right);
      }
   }

   template <class LabelType>
   void BinTree<LabelType>::printTree(Node *tree) {
      static int level = 0;
      if (tree) {
         for(int i = 0; i<level; i++) cout << "␣␣";
```

```
        cout << tree->label << '\n';
        level++;
        printTree(tree->left);
        printTree(tree->right);
        level--;
    }
}

template <class LabelType>
int BinTree<LabelType>::insert(LabelType son, LabelType father,
                               char c) {
    if ( createNode(son, root) ) { return 1; }
    Node *a = findNode(father, root);
    if (!a) return 0;
    if (c == 'l' && !a->left)  { return createNode(son, a->left); }
    if (c == 'r' && !a->right) { return createNode(son, a->right); }
    return 0;
}

#endif
```

La classe `BinTree` eredita la maggior parte delle sue funzionalità dalla classe `AbstrTree`. Inoltre, definisce i metodi (dichiarati virtuali puri nella classe base) per cancellare e inserire un nodo. Riguardo alla cancellazione, si implementa una semplice politica che prevede l'eliminazione dell'intero sotto-albero di cui il nodo da eliminare è radice.
Il metodo virtuale puro di `AbstrTree` per inserire un nodo prevede una *signature* che specifica due argomenti: il contenuto informativo del figlio da inserire, e il nodo padre (dato attraverso la sua label) a cui appenderlo. Nel caso dell'albero binario, occorre specificare però anche se si tratta del figlio destro o sinistro. Nell'implementazione presentata, si adotta la seguente soluzione: si fa l'*overloading* del metodo derivato, dandogli una signature che prevede un ulteriore parametro (discriminante tra destra e sinistra); il metodo derivato viene definito con un corpo che richiama la nuova versione overloaded (inserendo il figlio sempre a destra).
`BinTree` definisce anche il metodo protetto `printTree` per la stampa del contenuto di un sottoalbero, seguendo un'indentazione corrispondente al livello del nodo stampato.

D.3 La classe `GenTree`

Riportiamo di seguito il codice della classe `GenTree`, contenuta nel file GenTree.h.

```
#ifndef _GENTREE
#define _GENTREE

#include "BinTree.h"

    template <class LabelType>
    class GenTree : public BinTree<LabelType> {

    private:
        void deleteNode(LabelType, Node*&); // overridden
```

```
      void addSon (LabelType, Node*&);
      void printTree(Node*); // overridden
      void in() {}; // not accessible: meaningless for this kind of tree

   public:
      GenTree() : BinTree<LabelType>() {};
      virtual ~GenTree() {};
      int insert(LabelType, LabelType);
      void post() { inOrder(root); } ; // overridden: inOrder instead of postOrder
   };

// definitions

   template <class LabelType>
   void GenTree<LabelType>::deleteNode(LabelType x, Node* &tree) {
      if (tree) {
         if(tree->label == x) {
            Node *a = tree->right;
            tree->right = NULL;
            delTree(tree);
            tree = a;
            return;
         }
         deleteNode(x, tree->left);
         deleteNode(x, tree->right);
      }
   }

   template <class LabelType>
   void GenTree<LabelType>::addSon(LabelType x, Node* &tree) {
      if (!tree) { tree = new Node(x); }
      else addSon(x, tree->right);
   }

   template <class LabelType>
   void GenTree<LabelType>::printTree(Node *tree) {
      static int level = 0;
      if (tree) {
         for(int i = 0; i<level; i++) cout << "␣␣";
         cout << tree->label << '\n';
         tree = tree->left;
         level++;
         while(tree) {
            printTree(tree);
            tree = tree->right;
         }
         level--;
      }
   }
```

```
   template <class LabelType>
   int GenTree<LabelType>::insert(LabelType son, LabelType father) {
      if ( createNode(son, root) ) { return 1; }
      Node *a = findNode(father, root);
      if (!a) return 0;
      addSon(son, a->left);
      return 1;
   }

#endif
```

La classe `GenTree` eredita la maggior parte delle sue funzionalità dalla classe `BinTree` e, transitivamente, da `AbstrTree`. Essa si differenzia dalla sua classe base principalmente per l'implementazione di metodi di inserimento e cancellazione. Di conseguenza, viene fatto l'*overriding* dei metodi `insert` e `deleteNode`.
L'albero generico non prevede una visita in ordine simmetrico: dunque, per rendere inaccessibile il metodo pubblico in ereditato da `AbstrTree`, lo si ridefinisce con corpo vuoto nella sezione privata della classe. L'utilizzo della rappresentazione del tipo figlio-fratello comporta l'overriding del metodo pubblico `post` per la visita in ordine posticipato; la nuova definizione richiama il metodo protetto `inOrder` invece di `postOrder`. `GenTree` ridefinisce anche il metodo protetto `printTree` per la stampa del contenuto di un sottoalbero, in quanto il corrispondente metodo ereditato da `BinTree` opera correttamente soltanto su alberi binari.

D.4 La classe `SearchTree`

Riportiamo di seguito il codice della classe `SearchTree`, contenuta nel file SearchTree.h.

```
#ifndef _SEARCHTREE
#define _SEARCHTREE

#include "BinTree.h"

   template <class LabelType>
   class SearchTree : public BinTree<LabelType> {

   private:
      void insertNode(LabelType, Node*&);
      void deleteNode(LabelType, Node*&);
      Node* findNode(LabelType, Node*); // overridden
      // not accessible meaningless for this kind of tree
      void post() {};
      void deleteMin(Node*&, LabelType&);
      // not accessible meaningless for this kind of tree
      int insert(LabelType, LabelType, char) {};
   public:
      SearchTree() : BinTree<LabelType>() {};
      virtual ~SearchTree() {};
      // father is not taken into account
      int insert(LabelType son, LabelType father) { return insert(son);}
```

```
      int insert(LabelType x)  { insertNode(x, root); return 1; };
      void cancel(LabelType x) { deleteNode(x, root); };
   };

// definitions

   template <class LabelType>
   void SearchTree<LabelType>::insertNode (LabelType x, Node* &tree) {
      if (!createNode(x, tree) ) {
         if (x < tree->label) insertNode(x, tree->left);
         if (x > tree->label) insertNode(x, tree->right);
      }
   }

   template <class LabelType>
   void SearchTree<LabelType>::deleteNode(LabelType x, Node* &tree) {
      if (tree)
         if (x < tree->label) deleteNode(x, tree->left);
         else if (x > tree->label) deleteNode(x, tree->right);
         else if (!tree->left) {
            Node *a = tree;
            tree = tree->right;
            delete a;
         }
         else if (!tree->right) {
            Node *a = tree;
            tree = tree->left;
            delete a;
         }
         else deleteMin(tree->right, tree->label);
   }

   // overridden method
   template <class LabelType>
   SearchTree<LabelType>::Node*
   SearchTree<LabelType>::findNode (LabelType x, Node *tree) {
      if (!tree) return 0;
      if (x == tree->label) return tree;
      if (x < tree->label)  return findNode(x, tree->left);
      return findNode(x, tree->right);
   }

   template <class LabelType>
   void SearchTree<LabelType>::deleteMin (Node* &tree, LabelType &m) {
      if (tree->left) deleteMin(tree->left, m);
      else {
         m = tree->label;
         Node *a = tree;
         tree = tree->right;
         delete a;
```

```
        }
    }
#endif
```

La classe `SearchTree` eredita la maggior parte delle sue funzionalità dalla classe `BinTree` e, transitivamente, da `AbstrTree`. Essa si differenzia dalla sua classe base principalmente per l'implementazione di metodi di inserimento e cancellazione. Di conseguenza, viene fatto l'*overriding* dei metodi `insert` e `deleteNode`, facendo uso anche di vari metodi di utilità privati. Il metodo virtuale puro in `AbstrTree` per inserire un nodo prevede una *signature* che specifica due argomenti: il contenuto informativo del figlio da inserire, e il nodo padre (dato attraverso la sua label) a cui appenderlo. Nel caso dell'albero binario di ricerca, per l'inserimento di un nodo non occorre specificare il nodo padre. Nell'implementazione presentata, si adotta la seguente soluzione: si fa l'*overloading* del metodo derivato da `BinTree`, dandogli una signature con un solo parametro (il campo informativo del nodo); il metodo derivato viene definito con un corpo che richiama la nuova versione overloaded (scartando il dato relativo al padre).