Fondamenti di Algoritmi: Esercizi

Paolo Milazzo

7 giugno 2011

1 Esercizi sulla complessità degli algoritmi iterativi

1.1 Preliminari

Lo scopo di questi esercizi è imparare a calcolare la complessità asintotica di un algoritmo iterativo. La risoluzione di questi algoritmi richiede di conoscere (i) le regole per la stima del tempo di esecuzione dei comandi usati nell'algoritmo, (ii) la definizione della notazione asintotica O e (iii) le proprietà delle notazioni asintotiche. Richiamiamo ora questi concetti.

Regole per la stima del tempo di esecuzione dei comandi di un algoritmo. Le seguenti regole consentono di stimare il tempo di esecuzione **al caso pessimo** di un dato algoritmo. Le regole si applicano ai singoli comandi di un algoritmo. Il tempo di esecuzione dell'intero algoritmo sarà pari alla somma dei tempi di esecuzione dei singoli comandi che lo costituiscono.

Le regole sono le seguenti:

- Le singole operazioni logico-aritmetiche e di assegnamento hanno un tempo di esecuzione costante;
- Nel costrutto condizionale

  ```
  if (guardia) { blocco1 } else { blocco2 }
  ```

  uno solo tra i rami viene eseguito, in base al valore di `guardia`. Non potendo prevedere in generale tale valore (e quindi quale dei due blocchi sarà eseguito) il tempo di esecuzione di tale costrutto è pari a:

  $$\text{t(}\texttt{guardia}\text{)}+\max\{\ \text{t(}\texttt{blocco1}\text{)}\ ,\ \text{t(}\texttt{blocco2}\text{)}\ \}$$

  ossia, è pari al tempo della valutazione della guardia più il massimo tra i tempi di esecuzione dei due rami dell'`if`;
- Nel costrutto iterativo

  ```
  for (i=0; i<m; i=i+1) { corpo }
  ```

il tempo di esecuzione totale è pari a:

$$\sum_{i=0}^{m-1} t_i$$

dove t_i il tempo dell'esecuzione di `corpo` all'iterazione i-esima del ciclo.

- Nei costrutti iterativi

  ```
  while (guardia) { corpo }
  ```

  e

  ```
  do { corpo } while (guardia)
  ```

  sia m il numero di volte in cui `guardia` è soddisfatta. Sia t'_i il tempo della sua valutazione all'iterazione i del ciclo, e t_i il tempo di esecuzione `corpo` all'iterazione i. Poiché `guardia` viene valutata una volta in più rispetto a quante volte viene eseguito `corpo`, abbiamo il seguente tempo di esecuzione totale:

  $$\sum_{i=0}^{m} t'_i + \sum_{i=0}^{m-1} t_i$$

  La difficoltà nel calcolare il tempo di esecuzione totale di un ciclo `while` di solito sta nel determinare il valore di `m` (il numero di iterazioni). Spesso nella determinazione di `m` bisogna tenere conto del fatto che si sta facendo un'analisi al caso pessimo!

- Il tempo di esecuzione di una chiamata a funzione è pari al tempo di esecuzione della funzione stessa più quello dovuto al calcolo degli argomenti passati al momento dell'invocazione;

- Il tempo di esecuzione di una sequenza di comandi è pari alla somma dei tempi di esecuzione dei singoli comandi.

Notazione O. Data una funzione $g(n)$, si denota con $O(g(n))$ l'insieme di funzioni

$$O(g(n)) = \{f(n) \mid \text{esistono } c \in IR+ \text{ e } n_0 \in IN \text{ tali che } 0 \leq f(n) \leq cg(n) \text{ per ogni } n \geq n_0\}$$

Una funzione $f(n)$ appartiene all'insieme $O(g(n))$ se esiste la costante positiva c tale che $cg(n)$ sia sempre superiore a $f(n)$ per valori sufficientemente grandi di n. Malgrado $O(g(n))$ sia un insieme, per indicare che $f(n)$ appartiene a tale insieme si scriverà $f(n) = O(g(n))$. Si dice che $g(n)$ è un **limite asintotico superiore** per $f(n)$.

Proprietà delle notazioni asintotiche (nelle espressioni). Le notazioni asintotiche possono essere usate all'interno di espressioni per semplificare e rimuovere dettagli inesseziali dalle espressioni stesse. Ad esempio, supponiamo che il tempo di esecuzione di un algoritmo calcolato usando le regole sui comandi abbia una forma simile a $T(n) = \frac{1}{2}c'n^2 + (c + \frac{1}{2}c')n$. Possiamo scrivere più semplicemente questa espressione come $T(n) = \frac{1}{2}c'n^2 + \Theta(n)$. Se si è interessati al comportamento asintotico di $T(n)$ non è di nessuna importanza specificate tutti i termini di ordine più basso.

Vediamo alcune proprietà che sono utili per la determinazione della complessità asintotica degli algoritmi, permettendo di semplificarne il calcolo.

- *Regola dei fattori costanti:* per ogni costante positiva k vale $O(f(n)) = O(kf(n))$;
- *Regola della somma:* se $f(n) \in O(g(n))$ allora $f(n) + g(n) \in O(g(n))$;
  - nelle espressioni scriveremo anche $O(f(n)) + O(g(n)) = O(g(n))$;
- *Regola del prodotto:* se $f_1(n) \in O(g_1(n))$ e $f_2(n) \in O(g_2(n))$ allora $f_1(n)f_2(n) \in O(g_1(n)g_2(n))$;
  - nelle espressioni scriveremo anche $O(g_1(n))O(g_2(n)) = O(g_1(n)g_2(n))$;
- *Regola dei polinomi:* se $f(n) = a_m n^m + \ldots + a_1 n + a_0$ allore $f(n) \in O(n^m)$;
- *Regola transitiva:* se $f(n) \in O(g(n))$ e $g(n) \in O(h(n))$ allora $f(n) \in O(h(n))$.

1.2 Esempio di risoluzione di un esercizio

Dato un algoritmo iterativo si deve innanzitutto stimarne il tempo di esecuzione usando le regole sui comandi. Questo porta ad avere come risultato una formula $T(n) = f(n)$. A questo punto si può cercare di sviluppare $f(n)$ portandolo ad una forma polinomiale, per poi applicare la regola dei polinomi. Questo procedimento può però essere piuttosto complicato, soprattutto in merito alla risoluzione delle sommatorie che possono essere contenute in $f(n)$. Un metodo alternativo molto più semplice consiste nel procedere inserendo gradualmente la notazione asintotica O all'interno di $f(n)$ usando le proprietà delle notazioni asintotiche nelle espressioni.

Consideriamo ad esempio il seguente algoritmo che trova duplicati all'interno di un array `a` di dimensione `n`.

```
ContieneDuplicati( a ) {
  trovato = false;
  for (i=0 ; i<n-1 ; i=i+1) {
    for (j=i+1 ; j<n ; j=j+1) {
      if (a[i]=a[j]) trovato=true;
    }
  }
  return trovato;
}
```

Usando le regole per la stima del tempo di esecuzione dei comandi otteniamo che l'`if` interno ai due cicli ha tempo di esecuzione costante e i due cicli `for` danno origine a due sommatorie contenute una dentro l'altra. Il risultato è quindi la seguente formula:

$$T(n) = C_1 + \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} C_2$$

dove C_1 è una costante che rappresenta la somma dei tempi dell'inizializzazione di `trovato` a `false` e dell'esecuzione del comando `return` finale, mentre C_2 rappresenta il tempo di esecuzione dell'`if`.

A questo punto iniziamo a inserire la notazone O dove possibile, ossia al posto delle costanti, e otteniamo:

$$T(n) = O(1) + \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} O(1)$$

Ora possiamo semplificare la sommatoria più interna calcolando (facendo attezione) il numero di elementi della sommatoria stessa e moltiplicando tale numero per il contenuto.

$$\begin{aligned} T(n) &= O(1) + \sum_{i=0}^{n-2} ((n-1) - (i+1) + 1) O(1) \\ &= O(1) + \sum_{i=0}^{n-2} (n - i - 1) O(1) \end{aligned}$$

ATTENZIONE: in quest'ultimo passaggio non abbiamo potuto applicare la regola della somma alla sommatoria (semplificando quindi $\sum_{j=i+1}^{n-1} O(1)$ con $O(1)$) perchè tale regola si applica a un numero fissato di elementi sommati (o meglio, si applica alla somma di due soli elementi e va applicata un numero fissato di volte). La sommatoria in questione, invece, è composta da un numero di elementi che dipende da n, quindi dovremmo applicare la regola della somma un numero di volte che varia a seconda del valore di n, mentre noi stiamo cercando un unico risultato che valga per ogni valore di n.

Procediamo ora con la semplificazione della formula. ATTENZIONE: è bene notare che i nella formula non è una costante, ma una variabile che assume valori diversi nei diversi elementi della sommatoria. Quindi non possiamo scrivere $i = O(1)$, ma dobbiamo cercare di ricondurre il valore di i (in questo caso di $n-i$) al valore di n. Possiamo quindi notare che $(n-i-1) = O(n)$ (in quanto $n-i-1 \leq n$ per qualunque valore di i compreso tra 0 e $n-2$), e sostituire nella formula $(n-i-1)$ con $O(n)$, applicando poi la regola del prodotto.

$$\begin{aligned} T(n) &= O(1) + \sum_{i=0}^{n-2} O(n) O(1) \\ &= O(1) + \sum_{i=0}^{n-2} O(n) \end{aligned}$$

Semplifichiamo ora la seconda sommatoria in maniera simile a come abbiamo fatto per la prima. A questo punto possiamo trattare $O(n)$ come una costante e calcolare il numero di elementi della sommatoria. I passaggi successivi sono banali applicazioni della definizione di O e delle regole delle notazioni asintotiche:

$$\begin{aligned} T(n) &= O(1) + ((n-2) - 0 + 1)O(n) \\ &= O(1) + (n-1)O(n) \\ &= O(1) + O(n)O(n) \\ &= O(1) + O(n^2) \\ &= O(n^2) \end{aligned}$$

Il risultato ottenuto è che l'algoritmo considerato ha complessità quadratica $O(n^2)$ (o meglio, n^2 è un limite asintotico superiore della complessità dell'algoritmo).

1.3 Esercizi

Esercizio 1 *Calcolare, dettagliando i passaggi, la complessità computazionale (limite asintotico superiore O) del seguente algoritmo iterativo di calcolo del fattoriale di un numero* `n` *maggiore di 0.*

```
FattorialeIterativo( n ) {
  risultato = 1;
  for ( i=1 ; i<=n ; i=i+1 ) {
    risultato = risultato * i;
  }
  return risultato;
}
```

Esercizio 2 *Calcolare, dettagliando i passaggi, la complessità computazionale (limite asintotico superiore O) nel caso pessimo del seguente algoritmo di ricerca sequenziale di un elemento* `k` *in un array* `a` *di dimensione* `n`.

```
RicercaSequenziale( a , k ) {
  trovato = false;
  indice = -1;
  i = 0;
  while ((i<n) && (!trovato)) {
    if (a[i] == k) {
      trovato = true;
      indice = i;
    }
    else i = i+1;
  }
  return indice;
}
```

Esercizio 3 *Calcolare, dettagliando i passaggi, la complessità computazionale (limite asintotico superiore O) nel caso pessimo del seguente algoritmo di ordinamento per inserimento di un array* `a` *di dimensione* `n`*.*

```
InsertionSort( a ) {
  for (i = 0 ; i < n ; i = i +1) {
    prossimo = a[i];
    j = i;
    while ((j>0) && (a[j-1] > prossimo)) {
      a[j] = a[j-1];
      j = j-1;
    }
    a[j] = prossimo;
  }
}
```

Esercizio 4 *Calcolare, dettagliando i passaggi, la complessità computazionale (limite asintotico superiore O) nel caso pessimo del seguente algoritmo che calcola il numero di massimi locali in un array* `a` *di dimensione* `n`*. Per massimo locale si intende un elemento che sia maggiore dei due elementi che lo precedono (se presenti) e dei due elementi che lo seguono (se presenti). Ad esempio, in* `[3,5,4,6,3,2,9]` *i massimi locali sono* `6` *(che è maggiore di* `5,4,3,2`*) e* `9` *(che è maggiore di* `3,2`*).*

```
MassimiLocali( a ) {
  contatore = 0;
  for (i=0 ; i<n ; i=i+1) {
    massimo = true;
    for (j=-2; j<=2 ; j=j+1) {
      if ((i+j>0) && (i+j<n) && (j!=0)) {
        if (a[i+j]>a[i]) massimo=false;
      }
    }
    if (massimo) contatore = contatore+1;
  }
 return contatore;
}
```

Esercizio 5 *Calcolare, dettagliando i passaggi, la complessità computazionale (limite asintotico superiore O) nel caso pessimo del seguente algoritmo verifica se un array* `a` *di dimensione* `n` *è palindromo, ossia se* a[i] == a[n-1-i] *per ogni* i *compreso nell'intervallo* $[0, n/2]$ *(ad esempio,* [1,4,2,6,3,6,2,4,1] *è palindromo).*

```
PalindromoIterativo( a ) {
  i = 0;
  j = n-1;
```

```
  risposta = true;
  while (i<j) {
    if (a[i]!=a[j]) risposta=false;
    i=i+1;
    j=j-1;
  }
 return risposta;
}
```

Esercizio 6 *Calcolare, dettagliando i passaggi, la complessità computazionale (limite asintotico superiore O) nel seguente algoritmo che dato in input un array bidimensionale* `a` *di* `n` *righe e* `n` *colonne, verifica che ogni riga e ogni colonna di* `a` *non contenga elementi duplicati. (Nota: questo algoritmo controlla contemporaneamente righe e colonne facendo un opportuno uso degli indici)*

```
VerificaRigheColonne( a ) {
  trovato = false;
  for (i=0 ; i<n ; i=i+1) {
    for (j=0 ; j<n-1 ; j=j+1) {
      for (k=j+1 ; k<n ; k=k+1) {
        if (a[i][j]=a[i][k]) trovato=true;
        if (a[j][i]=a[k][i]) trovato=true;
      }
    }
  }
  return trovato;
}
```

1.4 Soluzioni degli esercizi

Ecco le soluzioni agli esercizi.

Soluzione dell'esercizio 1 (Algoritmo `FattorialeIterativo`):

$$\begin{aligned} T(n) &= C_1 + \sum_{i=1}^{n} C_2 \\ &= O(1) + \sum_{i=1}^{n} O(1) \\ &= O(1) + nO(1) \\ &= O(1) + O(n)O(1) \\ &= O(1) + O(n) \\ &= O(n) \end{aligned}$$

dove:

- C_1 rappresenta il tempo di esecuzione dei comandi esterni al ciclo `for`
- C_2 rappresenta il tempo di esecuzione del comando interno al ciclo `for`

Soluzione dell'esercizio 2 (Algoritmo `RicercaSequenziale`):

$$\begin{aligned} T(n) &= C_1 + (n+1)C_2 + nC_3 \\ &= C_1 + nC_2 + C_2 + nC_3 \\ &= O(1) + O(n) + O(1) + O(n) \\ &= O(n) \end{aligned}$$

dove:

- il caso pessimo (usato per stimare numero di iterazioni del `while`) è quello in cui l'array in input non contiene l'elemento cercato
- C_1 rappresenta il tempo di esecuzione dei comandi esterni al ciclo `while`
- C_2 rappresenta il tempo di valutazione della guardia del ciclo `while`
- C_3 rappresenta il tempo di esecuzione dell'`if`

Soluzione dell'esercizio 3 (Algoritmo `InsertionSort`):

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-1}(C_1 + (i+1)C_2 + iC_3) \\ &= \sum_{i=0}^{n-1}(C_1 + iC_2 + C_2 + iC_3) \\ &= \sum_{i=0}^{n-1}(O(1) + O(n) + O(1) + O(n)) \\ &= \sum_{i=0}^{n-1} O(n) \\ &= ((n-1) - 0 + 1)O(n) \\ &= O(n)O(n) \\ &= O(n^2) \end{aligned}$$

dove:

- il caso pessimo (usato per stimare numero di iterazioni del `while`) è quello in cui l'array in input contiene elementi disposti in ordine decrescente

- C_1 rappresenta il tempo di esecuzione dei comandi esterni al ciclo `while`
- C_2 rappresenta il tempo di valutazione della guardia del ciclo `while`
- C_3 rappresenta il tempo di esecuzione del contenuto del ciclo `while`

Soluzione dell'esercizio 4 (Algoritmo `MassimiLocali`):

$$\begin{aligned}
T(n) &= C_1 + \sum_{i=0}^{n-1}(C_2 + \sum_{i=-2}^{2} C_3) \\
&= O(1) + \sum_{i=0}^{n-1}(O(1) + \sum_{i=-2}^{2} O(1)) \\
&= O(1) + \sum_{i=0}^{n-1}(O(1) + 5O(1)) \\
&= O(1) + \sum_{i=0}^{n-1}(O(1) + O(1)O(1)) \\
&= O(1) + \sum_{i=0}^{n-1} O(1) \\
&= O(1) + ((n-1) - 0 + 1)O(1) \\
&= O(1) + O(n)O(1) \\
&= O(n)
\end{aligned}$$

dove:

- C_1 rappresenta il tempo di esecuzione dei comandi esterni al primo ciclo `for` (il più esterno)
- C_2 rappresenta il tempo di esecuzione dei comandi esterni tra il primo e il secondo ciclo `for`
- C_3 rappresenta il tempo di esecuzione del contenuto del secondo ciclo `for` (il più interno)

Soluzione dell'esercizio 5 (Algoritmo `PalindromoIterativo`):

$$\begin{aligned}
T(n) &= C_1 + \sum_{i=0}^{(n/2)-1} C_2 \\
&= O(1) + \sum_{i=0}^{(n/2)-1} O(1) \\
&= O(1) + (n/2)O(1)
\end{aligned}$$

$$
\begin{aligned}
&= O(1) + O(n)O(1) \\
&= O(1) + O(n) \\
&= O(n)
\end{aligned}
$$

dove:

- C_1 rappresenta il tempo di esecuzione dei comandi esterni al ciclo `while`
- C_2 rappresenta il tempo di esecuzione dei comandi interni al ciclo `while`

Soluzione dell'esercizio 6 (Algoritmo `VerificaRigheColonne`):

$$
\begin{aligned}
T(n) &= C_1 + \sum_{i=0}^{n-1}\sum_{j=0}^{n-2}\sum_{k=j+1}^{n-1} C_2 \\
&= O(1) + \sum_{i=0}^{n-1}\sum_{j=0}^{n-2}\sum_{k=j+1}^{n-1} O(1) \\
&= O(1) + \sum_{i=0}^{n-1}\sum_{j=0}^{n-2}(n-j-1)O(1) \\
&= O(1) + \sum_{i=0}^{n-1}\sum_{j=0}^{n-2} O(n)O(1) \\
&= O(1) + \sum_{i=0}^{n-1}\sum_{j=0}^{n-2} O(n) \\
&= O(1) + \sum_{i=0}^{n-1}(n-1)O(n) \\
&= O(1) + \sum_{i=0}^{n-1} O(n)O(n) \\
&= O(1) + \sum_{i=0}^{n-1} O(n^2) \\
&= O(1) + nO(n^2) \\
&= O(1) + O(n)O(n^2) \\
&= O(1) + O(n^3) \\
&= O(n^3)
\end{aligned}
$$

dove:

- C_1 rappresenta il tempo di esecuzione dei comandi esterni ai cicli `for`
- C_2 rappresenta il tempo di esecuzione dei due `if`

2 Esercizi sulla complessità degli algoritmi ricorsivi (divide-et-impera)

Lo scopo di questi esercizi è imparare a calcolare la complessità asintotica di un algoritmo ricorsivo che segue l'approccio divide-et-impera. La risoluzione di questi algoritmi richiede di conoscere (i) come esprimere il tempo di esecuzione di un algoritmo con un'equazione di ricorrenza e (ii) il teorema fondamentale delle ricorrenze. Richiamiamo ora questi concetti.

Equazioni di ricorrenza. Un algoritmo ricorsivo generalmente ha una struttura simile alla seguente:

```
AlgoritmoRicorsivo(a) {
  if (.....) {
     // casi di base (possono essere più di uno)
  }
  else {
     // caso ricorsivo (include k chiamate ricorsive)
     ....
     AlgoritmoRicorsivo(a1)
     ....
     AlgoritmoRicorsivo(ak)
     ....
  }
}
```

Dato un algoritmo ricorsivo, si può esprimere il suo tempo di esecuzione con una funzione definita per casi come segue:

$$T(n) = \begin{cases} O(f(n)) & \text{nei casi di base} \\ T(n_1) + \ldots + T(n_k) + O(g(n)) & \text{nel caso ricorsivo} \end{cases}$$

dove $f(n)$ rappresenta la complessità del caso di base, $n_1, \ldots, n_k$ sono le dimensioni dei sottoproblemi su cui si effettuano le k chiamate ricorsive e $g(n)$ rappresenta la complessità delle altre operazioni del caso ricorsivo. Nel caso degli algoritmi che seguono l'approccio divide-et-impera $g(n)$ include il tempo delle fasi di decomposizione e ricombinazione.

Esempio: calcolo del fattoriale.

```
Fattoriale(n) {
  if (n=1) return 1;
  else return n*Fattoriale(n-1);
}
```

L'equazione corrispondente è:

$$T(n) = \begin{cases} O(1) & \text{se } n = 1; \\ T(n-1) + O(1) & \text{altrimenti} \end{cases}$$

Il teorema fondamentale delle ricorrenze. La definizione del teorema è la seguente.

Teorema 1 *Siano $\alpha \geq 1, \beta > 1$ e $n_0 \geq 0$ costanti e $f(n)$ una funzione, allora l'equazione di ricorrenza*

$$T(n) = \begin{cases} O(1) & \textit{se } n \leq n_0 \\ \alpha T(n/\beta) + f(n) & \textit{altrimenti} \end{cases}$$

(dove n/β va interpretato come $\lfloor n/\beta \rfloor$ o $\lceil n/\beta \rceil$) ha le seguenti soluzioni per ogni n:

1. *$T(n) = O(f(n))$ se esiste una costante $\gamma < 1$ tale che $\alpha f(n/\beta) = \gamma f(n)$;*
2. *$T(n) = O(f(n) \log_\beta n)$ se $\alpha f(n/\beta) = f(n)$;*
3. *$T(n) = O(n^{\log_\beta \alpha})$ se esiste una costante $\gamma' > 1$ tale che $\alpha f(n/\beta) = \gamma' f(n)$.*

2.1 Esempio di risoluzione di un esercizio

Si consideri il seguente algoritmo di ricerca binaria ricorsiva di un elemento `k` in un array ordinato `a` (restituisce l'indice dell'elemento se presente, oppure `-1` se non presente) in cui `sinistra` e `destra` sono indici di `a` e rappresentano gli estremi (inclusi) della porzione di array da considerare.

Le fasi dell'algoritmo (in stile divide-et-impera) sono le seguenti:

- **Decomposizione:** dividi l'array a metà e identifica la parte in cui sarà contenuto l'elemento da cercare
- **Ricorsione:** ricerca ricorsivamente l'elemento in una delle due parti
- **Ricombinazione:** restituisci il risultato della chiamata ricorsiva cosi' com'è

L'algorimo è il seguente:

```
RicercaBinariaRicorsiva( a , k , sinistra , destra ) {
  if (sinistra == destra) {
    if (k == a[sinistra]) return sinistra;
    else return -1;
  }
```

```
  else {
    centro = (sinistra + destra)/2;
    if (k <= a[centro])
      return RicercaBinariaRicorsiva( a , k , sinistra , centro );
    else
      return RicercaBinariaRicorsiva( a , k , centro+1 , destra );
  }
}
```

Il caso di base lo si ha quando `sinistra` e `destra` sono uguali, quindi la porzione di array considerata consiste di un solo elemento ($n = 1$). In questo caso si esegue un `if` in cui ogni ramo esegue una sola operazione a tempo costante (`return`), quindi il tempo totale del caso di base è O(1). Nel caso ricorsivo si esegue una sola chiamata ricorsiva su una porzione di array di dimensione $n/2$ (alternativamente sulla prima o sulla seconda metà dell'array). Le fasi di decomposizione e ricombinazione si eseguono con operazioni a tempo costante $O(1)$.

L'equazione di ricorrenza corrispondente è quindi:

$$T(n) = \begin{cases} O(1) & \text{se } n = 1; \\ T(n/2) + O(1) & \text{altrimenti} \end{cases}$$

Applichiamo ora il teorema fondamentale delle ricorrenze. Abbiamo $\alpha = 1, \beta = 2$ e $n_0 = 1$. Possiamo maggiorare il tempo di esecuzione delle fasi di decomposizione e ricombinazione con una constante c. In altre parole, qualunque sia la funzione $g(n)$ rappresentata da $O(1)$ sappiamo, per definizione della notazione asintotica O, che esiste una costante c tale che $g(n) \leq c$. Quindi se sostituiamo $O(1)$ con c sicuramente abbiamo una stima corretta (pessimistica) del tempo di esecuzione delle due fasi considerate.

Abbiamo quindi $f(n) = c$ e dobbiamo confrontare $\alpha f(n/\beta)$ con $f(n)$ per capire in quale dei tre casi del teorema ricada il nostro algoritmo. In questo caso $\alpha f(n/\beta) = c = f(n)$. Il caso in cui ricade l'algoritmo è quindi il secondo e ne consegue che il tempo di esecuzione è $T(n) = O(\log_2 n)$.

2.2 Esercizi

Esercizio 7 *Si consideri il seguente algoritmo ricorsivo che, dato un numero intero n, calcola il massimo valore $k \geq 0$ tale che $2^k \leq n$.*

```
PotenzeDiDue(n) {
  if (n<=1)
    return 0;
  else
    return 1 + PotenzeDiDue(n/2);
}
```

Determinare l'equazione di ricorrenza dell'algoritmo e calcolare la complessità usando il teorema fondamentale delle ricorrenze.

Esercizio 8 *Si consideri il seguente algoritmo ricorsivo che, dato un array* a *contenente numeri interi, calcola la somma dei valori contenuti nell'array.*

```
Somma(a,sinistra,destra) {
  if (sinistra==destra)
    return a[sinistra];
  else {
    centro = (sinistra+destra)/2;
    somma1 = Somma(a,sinistra,centro);
    somma2 = Somma(a,centro+1,destra);
    return somma1+somma2;
  }
}
```

Determinare l'equazione di ricorrenza dell'algoritmo e calcolare la complessità usando il teorema fondamentale delle ricorrenze.

Esercizio 9 *Si consideri il seguente algoritmo ricorsivo che, dato un array* a *di dimensione* $n = 2^k - 1$ *per un* k *qualsiasi e contenente numeri interi, verifica se tale array corrisponde al risultato di una visita simmetrica un albero binario di ricerca bilanciato e completo.*

```
ABR(a,sinistra,destra) {
  if (sinistra==destra)
    return true;
  else {
    verifica = true;
    centro = (sinistra+destra)/2;
    for (i=sinistra; i<centro; i++) {
      if (a[i]>a[centro]) verifica = false;
    }
    for (i=centro+1; i<=destra, i++) {
      if (a[i]<=[centro]) verifica = false;
    }
    verifica1 = ABR(a,sinistra,centro-1);
    verifica2 = ABR(a,centro+1,destra);
    return (verifica && verifica1 && verifica2);
  }
}
```

Determinare l'equazione di ricorrenza dell'algoritmo e calcolare la complessità usando il teorema fondamentale delle ricorrenze.

Esercizio 10 *Si consideri il seguente algoritmo ricorsivo che, dato un array* a *di dimensione* n*, verifica se esistono due numeri diversi* i,j *compresi nell'intervallo* $[0, n-1]$ *tali che* `a[i]=j` *e* `a[j]=i`*.*

```
IndiciValori(a,sinistra,destra) {
  if (sinistra == destra) {
    return false;
  }
  else {
    trovato = false;
    centro = (sinistra+destra)/2;
    for (int i=sinistra; i<=centro; i++) {
      for (int j=centro+1; j<=destra; j++) {
        if ((a[i]==j)&&(a[j]==i))
          trovato = true;
      }
    }
    trovato1 = IndiciValori(a,sinistra,centro);
    trovato2 = IndiciValori(a,centro+1,destra);
    return (trovato || trovato1 || trovato2);
  }
}
```

2.3 Soluzioni degli esercizi

Ecco le soluzioni agli esercizi.

Soluzione dell'esercizio 7 (Algoritmo `PotenzeDiDue`): L'equazione di ricorrenza che descrive il tempo di esecuzione dell'algoritmo è:

$$T(n) = \begin{cases} O(1) & \text{se } n = 1 \\ T(n/2) + O(1) & \text{altrimenti} \end{cases}$$

Abbiamo $\alpha = 1$,$\beta = 2$ e possiamo assumere $f(n) = c$.

Ora, $f(n/2) = c = f(n)$, quindi siamo nel caso 2 del teorema fondamentale delle ricorrenze.

La complessità asintotica dell'algoritmo è quindi in $O(\log_2 n)$.

Soluzione dell'esercizio 8 (Algoritmo `Somma`): L'equazione di ricorrenza che descrive il tempo di esecuzione dell'algoritmo è:

$$T(n) = \begin{cases} O(1) & \text{se } n = 1 \\ 2T(n/2) + O(1) & \text{altrimenti} \end{cases}$$

Abbiamo $\alpha = 2$,$\beta = 2$ e possiamo assumere $f(n) = c$.

Ora, $2f(n/2) = 2c = 2f(n)$, quindi $\gamma = 2 > 1$ e siamo nel caso 3 del teorema fondamentale delle ricorrenze.

La complessità asintotica dell'algoritmo è quindi in $O(n^{\log_2 2}) = O(n)$.

Soluzione dell'esercizio 9 (Algoritmo `ABR`): L'equazione di ricorrenza che descrive il tempo di esecuzione dell'algoritmo è:

$$T(n) = \begin{cases} O(1) & \text{se } n = 1 \\ 2T(n/2) + O(n) & \text{altrimenti} \end{cases}$$

Abbiamo $\alpha = 2$,$\beta = 2$ e possiamo assumere $f(n) = cn$.

Ora, $2f(n/2) = 2cn/2 = cn = f(n)$, quindi siamo nel caso 2 del teorema fondamentale delle ricorrenze.

La complessità asintotica dell'algoritmo è quindi in $O(n \log_2 n)$.

Soluzione dell'esercizio 10 (Algoritmo `IndiciValori`): L'equazione di ricorrenza che descrive il tempo di esecuzione dell'algoritmo è:

$$T(n) = \begin{cases} O(1) & \text{se } n = 1 \\ 2T(n/2) + O(n^2) & \text{altrimenti} \end{cases}$$

Abbiamo $\alpha = 2$,$\beta = 2$ e possiamo assumere $f(n) = cn^2$.

Ora, $2f(n/2) = 2c(n/2)^2 = cn^2/2 = (1/2)f(n)$, quindi $\gamma = 1/2 < 1$ e siamo nel caso 1 del teorema fondamentale delle ricorrenze.

La complessità asintotica dell'algoritmo è quindi in $O(n^2)$.

3 Esercizi su algoritmi basati su array

Lo scopo di questi esercizi è imparare a progettare algoritmi che utilizzano informazioni memorizzate in un array. Un algoritmo può essere anche suddiviso in "parti" delegate ad algoritmi separati (come gli algoritmi MergeSort e QuickSort che usano gli algoritmi Fusione e Distribuzione). Alcuni di questi esercizi possono prevedere diverse soluzioni con complessità computazionali diverse. La soluzione che prevede la minore complessità computazionale sarà ovviamente la migliore. Le altre soluzioni saranno comunque considerate accettabili.

3.1 Esercizi

Esercizio 11 *Scrivere un algoritmo* **iterativo** Massimo *che calcola il valore massimo contenuto in un array* a *di dimensione* n. *Esprimere la complessità computazionale di questo algoritmo usando la notazione asintotica* O.

Esercizio 12 *Scrivere un algoritmo* **iterativo** Totocalcio *che dato un array bidimensionale* a *che rappresenta una tabella con 13 righe e 2 colonne da interpretare come i risultati di una giornata calcistica, restituisce un array di stringhe* b *di 13 elementi in cui ogni elemento conterrà* "1","X" *o* "2" *a seconda che il risultato corrispondente sia una vittoria della squadra di casa, un pareggio o una vittoria della squadra in trasferta. Esprimere la complessità computazionale di questo algoritmo usando la notazione asintotica* O.

Esercizio 13 *Scrivere un algoritmo* **iterativo** ParentesiBilanciate *che verifica se un array* a *di dimensione* n *in cui ogni elemento è una stringa* "(" *o* ")" *rappresenta una sequenza di parentesi bilanciate, ossia se l'ordine di apertura e chiusura delle parentesi è corretto rispetto a come di solito si usano nelle espressioni aritmetiche. Ad esempio* ["(","(",")","(","(",")",")",")"] *è corretto, mentre* ["(",")",")","(","("] *non lo è.*

Esercizio 14 *Scrivere un algoritmo* MinMaxMezzi *che, dato un array* a *di dimensione* n *contenente numeri naturali* **ordinati in modo crescente** *calcola il minimo* `min` *e il massimo* `max` *degli elementi contenuti nell'array e verifica se il valore* (min+max)/2 *è contenuto nell'array. Esprimere la complessità computazionale di questo algoritmo usando la notazione asintotica* O.

Esercizio 15 *Scrivere un algoritmo* **ricorsivo** MassimoRicorsivo *che calcola il valore massimo contenuto in un array* a *di dimensione* n.

Esercizio 16 *Scrivere un algoritmo* **ricorsivo** PalindromoRicorsivo *che verifica se un array* a *di dimensione* n *è palindromo, ossia se* a[i]==a[n-i-1] *per ogni* i *compreso nell'intervallo* $[1, n/2]$ *(ad esempio,* [1,4,2,6,3,6,2,4,1] *è palindromo).*

Esercizio 17 *Scrivere un algoritmo* **iterativo** MaxSequenzaElementiUguali *che dato un array* a *calcola il numero di elementi della più lunga porzione di array costituita tutta da elementi consecutivi uguali tra loro. Ad esempio, se* a = [5,7,3,3,8,9,9,9,5,3,2,2]*, allora la risposta è 3 in quanto la porzione* [9,9,9] *è la più lunga porzione di* a *fatta da elementi consecutivi tutti uguali.*

Esprimere la complessità computazionale di questo algoritmo usando la notazione asintotica O.

Esercizio 18 *Scrivere un algoritmo* ElementoPiuFrequente *che dato un array* a *calcola il valore più presente all'interno dell'array. Ad esempio, se* a = [2,6,8,5,2,3,6,8,9,5,3,1,2]*, allora la risposta è 2 in quanto questo valore compare 3 volte all'interno dell'array, mentre gli altri valori compaiono al più 2 volte.*

Esercizio 19 *Scrivere un algoritmo* **ricorsivo** ContieneDuplicatiRicorsivo *che dato un array* a *restituisce true o false a seconda che l'array contenga due elementi uguali (ossia esistono* i *e* j *con* i != j *tali che* a[i]==a[j]*).*

Esprimere la complessità computazionale di questo algoritmo usando la notazione asintotica O.

3.2 Soluzioni degli esercizi

Ecco le soluzioni agli esercizi.

Soluzione dell'esercizio 11 (Algoritmo `Massimo`):

```
Massimo( a ) {
  max = a[0];
  for (i=1 ; i<n ; i++) {
    if (a[i]>max) max=a[i];
  }
  return max;
}
```

La complessità di questo algoritmo è in $O(n)$ in quanto ogni elemento dell'array viene utilizzato esattamente una volta.

Soluzione dell'esercizio 12 (Algoritmo `Totocalcio`):

```
Totocalcio( a ) {
  for (i=0 ; i<13 ; i++) {
    if (a[i][1] > a[i][2]) b[i] = "1";
```

```
    else if (a[i][1] > a[i][2]) b[i] = "2";
    else b[i] = "X";
  }
  return b;
}
```

La complessità di questo algoritmo è in $O(1)$ in quanto il numero di iterazioni dell'unico ciclo presente è fissato (13). In realta' anche la dimensione dell'input è fissata!

Soluzione dell'esercizio 13 (Algoritmo `ParentesiBilanciate`):

```
ParentesiBilanciate( a ) {
  contatore = 0;
  for (i=0; i<n; i++) {
    if (a[i] =="(") contatore++;
    else if (contatore>0) contatore--;
         else return false;
  }
  if (contatore==0) return true;
  else return false;
}
```

Soluzione dell'esercizio 14 (Algoritmo `MinMaxMezzi`):

```
MinMaxMezzi( a ) {
  min = a[0];
  max = a[n-1];
  mmm = (min+max)/2;
  return RicercaBinaria(a,mmm);
}
```

Soluzione dell'esercizio 15 (Algoritmo `MassimoRicorsivo`):

```
MassimoRicorsivo ( a , sinistra , destra ) {
  if (sinistra==destra)
    return a[sinistra];
  else {
    centro = (sinistra+destra)/2;
    m1 = MassimoRicorsivo( a , sinistra , centro );
    m2 = MassimoRicorsivo( a , centro+1 , destra );
    if (m1>m2) return m1;
    else return m2;
  }
}
```

Soluzione dell'esercizio 16 (Algoritmo `PalindromoRicorsivo`):

```
PalindromoRicorsivo( a , sinistro , destra ) {
  if (sinistra==destra)
    return true;
  else if ((sinistra==(destra-1)) && (a[sinistra]==a[destra]))
    return true;
  else if (a[sinistra]==a[destra])
    return PalindromoRicorsivo(a,sinistra+1,destra-1);
  else
    return false;
}
```

Soluzione dell'esercizio 17 (Algoritmo `MaxSequenzaElementiUguali`):

```
MaxSequenzaElementiUguali( a ) {
  risultato=1;
  contatore=1;
  for (i=1; i<n; i++) {
    if (a[i]==a[i-1]) contatore++;
    else {
      if (contatore>risultato) {
        risultato = contatore;
      }
      contatore=1;
    }
  }
  if (contatore>risultato) return contatore;
  else return risultato;
}
```

Soluzione dell'esercizio 18 (Algoritmo `ElementoPiuFrequente`):

```
ElementoPiuFrequente( a ) {
  QuickSort(a,0,n-1);
  risultato=a[0];
  contatore_risultato=1;
  contatore=1;
  for (i=1; i<n; i++) {
    if (a[i]==a[i-1]) contatore++;
    else {
      if (contatore>contatore_risultato) {
        contatore_risultato = contatore;
        risultato=a[i-1];
      }
```

```
      contatore=1;
    }
  }
  if (contatore>contatore_risultato) return a[n];
  else return risultato;
}
```

Soluzione dell'esercizio 19 (Algoritmo `ContieneDuplicatiRicorsivo`):

```
ContieneDuplicatiRicorsivo ( a , sinistra , destra ) {
  if (sinistra==destra)
    return false;
  else {
    centro = (sinistra+destra)/2;
    for (i=sinistra; i<=centro; i++) {
      for (j=centro+1; j<=destra; j++) {
        if (a[i]==a[j]) return true;
      }
    }
    d1 = ContieneDuplicatiRicorsivo( a , sinistra , centro );
    d2 = ContieneDuplicatiRicorsivo( a , centro+1 , destra );
    return (d1 || d2);
  }
}
```

4 Esercizi su algoritmi basati su liste

Lo scopo di questi esercizi è imparare a progettare algoritmi che utilizzano informazioni memorizzate in una lista. Un algoritmo può essere anche suddiviso in "parti" delegate ad algoritmi separati (come gli algoritmi su array MergeSort e QuickSort che usano gli algoritmi Fusione e Distribuzione). Alcuni di questi esercizi possono prevedere diverse soluzioni con complessità computazionali diverse. La soluzione che prevede la minore complessità computazionale sarà ovviamente la migliore. Le altre soluzioni saranno comunque considerate accettabili.

4.1 Esercizi

Esercizio 20 *Scrivere un algoritmo* **iterativo** Conta *che calcola il numero di elementi contenuti in una data lista* l.

Esercizio 21 *Scrivere un algoritmo* **ricorsivo** ContaRicorsivo *che calcola il numero di elementi contenuti in una data lista* l.

Esercizio 22 *Scrivere un algoritmo* **iterativo** Somma *che calcola la somma degli elementi contenuti in una data lista* l.

Esercizio 23 *Scrivere un algoritmo* **ricorsivo** SommaRicorsiva *che calcola la somma degli elementi contenuti in una data lista* l.

Esercizio 24 *Scrivere un algoritmo* **iterativo** MassimoLista *che calcola il valore massimo contenuto in una lista* l *i cui valori sono tutti maggiori o uguali a zero. Restituire* -1 *nel caso di lista vuota.*

Esercizio 25 *Scrivere un algoritmo* **ricorsivo** MassimoListaRicorsivo *che calcola il valore massimo contenuto in una lista* l *i cui valori sono tutti maggiori o uguali a zero. Restituire* -1 *nel caso di lista vuota.*

Esercizio 26 *Scrivere un algoritmo* **iterativo** PalindromoListaDoppia *che verifica (restituendo* true *o* false*) se una lista doppia non vuota è palindroma (si vedano esercizi precedenti sugli array per la definizione di palindromo).* **Suggerimento:** *si può vedere se due variabili* l1 *e* l2 *contengono riferimenti allo stesso elemento di una lista usando l'espressione* l1 == l2 *in un* if.

Esercizio 27 *Scrivere un algoritmo* **iterativo** PalindromoLista *che verifica (restituendo* true *o* false*) se una lista (semplice, non doppia) sono vuota è palindroma (si vedano esercizi precedenti sugli array e suggerimento nell'esercizio precedente).*

Esercizio 28 *Scrivere un algoritmo* **ricorsivo** PalindromoListaDoppiaRicorsivo *che verifica (restituendo* true *o* false*) se una lista doppia non vuota è palindroma (si vedano esercizi precedenti sugli array e suggerimento nell'esercizio precedente).*

Esercizio 29 *Realizzare usando una lista una struttura dati astratta per rappresentare insiemi di valori. Si assuma che l'insieme vuoto sia rappresentato da un riferimento a* null *e che il riferimento al (o agli) insiemi su cui agiscono le operazioni della struttura dati astratta debbano essere passati a tutte le funzioni che implementano tali operazioni. La struttura dati astratta deve prevedere le seguenti operazioni:*

- Contains (set , x) – *restituisce* true *o* false *a seconda che l'elemento* x *sia presente o meno nell'insieme riferito da* set
- Insert (set , x) – *aggiunge l'elemento* x *all'insieme riferito da* set *evitando di creare duplicati (se l'elemento era già presente lascia* set *così com'è)*
- Remove (set , x) – *rimuove l'elemento* x *(se presente) dall'insieme riferito da* set
- Union (set1 , set2) – *restituisce un nuovo insieme che contiene l'unione degli elementi di* set1 *e* set2 *evitando duplicati nel caso un elemento sia presente in entrambi gli insiemi. Attenzione: questa operazione non deve modificare* set1 *e* set2 *ma fare* return *di un nuovo insieme.*
- Difference (set1 , set2) – *restituisce un nuovo insieme che contiene gli elementi di* set1 *che non sono presenti in* set2*. Attenzione: questa operazione non deve modificare* set1 *e* set2 *ma fare* return *di un nuovo insieme*
- Empty (set) – *restituisce* true *se l'insieme riferito da* set *non contiene elementi*

Nota: esistono diverse buone soluzioni con diverse complessità computazionali per le varie operazioni. In particolare, esiste una soluzione in cui Difference *si esegue in* $O(n1 \times n2)$ *e un'altra soluzione in cui* Difference *si esegue in* $O(n1 + n2)$*, dove* $n1$ *e* $n2$ *sono il numero di elementi in* set1 *e* set2*.*

Esercizio 30 *Realizzare usando una lista una struttura dati astratta per rappresentare un server di posta elettronica. Si assuma che ogni elemento della lista (oltre al campo* succ*) contenga due campi* destinatario *e* messaggio*, entrambi stringhe. Si assuma che la lista dei messaggi del server sia riferita dalla variabile* lista *senza bisogno di passarla a tutte le funzioni che implementano le operazioni. In caso di assenza di messaggi nel server avremo che* lista *è* null*. La struttura dati astratta deve prevedere le seguenti operazioni:*

- NewMessage (dest, msg) – *aggiunge nel server il nuovo messaggio* msg *con destinatario* dest
- GetMessages (dest) – *restituisce una lista contenente tutti i messaggi con destinatario* dest *rimuovendoli dal server*
- Contains (dest) – *restituisce* true *o* false *a seconda che siano presenti o meno messaggi nel server il cui destinatario è* dest*.*

4.2 Soluzioni degli esercizi

Ecco le soluzioni agli esercizi.

Soluzione dell'esercizio 20 (Algoritmo `Conta`):

```
Conta( l ) {
  contatore = 0;
  while (l!=null) {
    l=l.succ;
    contatore++;
  }
  return contatore;
}
```

Soluzione dell'esercizio 21 (Algoritmo `ContaRicorsivo`):

```
ContaRicorsivo( l ) {
  if (l==null) return 0;
  else return 1+ContaRicorsivo(l.succ);
}
```

Soluzione dell'esercizio 22 (Algoritmo `Somma`):

```
Somma( l ) {
  tot = 0;
  while (l!=null) {
    tot = tot+l.dato;
    l=l.succ;
  }
}
```

Soluzione dell'esercizio 23 (Algoritmo `SommaRicorsiva`):

```
SommaRicorsiva( l ) {
  if (l==null) return 0;
  else return l.dato + SommaRicorsiva(l.succ);
}
```

Soluzione dell'esercizio 24 (Algoritmo `MassimoLista`):

```
MassimoLista( l ) {
  risultato = -1;
  while (l!=null) {
    if (l.dato>risultato) {
```

```
      risultato=l.dato;
    }
    l = l.succ;
  }
}
```

Soluzione dell'esercizio 25 (Algoritmo `MassimoListaRicorsivo`):

```
MassimoListaRicorsivo( l ) {
  if (l==null) return -1;
  else {
    m = MassimoListaRicorsivo( l.succ );
    if (m>l.dato) return m;
    else return l.dato;
  }
}
```

Soluzione dell'esercizio 26 (Algoritmo `PalindromoListaDoppia`):

```
PalindromoListaDoppia( testa , coda ) {
  l1=testa;
  l2=coda;
  risultato=true;
  while ((l1!=l2)&&(l1.succ!=l2)) {
    if (l1.dato!=l2.dato) {
      risultato=false;
    }
    l1=l1.succ;
    l2=l2.pred;
  }
  return risultato;
}
```

Soluzione dell'esercizio 27 (Algoritmo `PalindromoLista`):

```
PalindromoLista( l ) {
  l1=l;
  l2=Coda(l);
  risultato=true;
  while ((l1!=l2)&&(l2.succ!=l1)) {
    if (l1.dato!=l2.dato) {
      risultato=false;
    }
    l1=l1.succ;
    l2=Pred(l,l2);
```

```
  }
  return risultato;
}

Coda(l) {
  ltemp = l;
  while (ltemp.succ!=null) {
    ltemp = ltemp.succ;
  }
  return ltemp;
}

Pred(l,l2) {
  ltemp = l;
  while (ltemp.succ!=l2) {
    ltemp = ltemp.succ;
  }
  return ltemp;
}
```

Soluzione dell'esercizio 28 (Algoritmo `PalindromoListaDoppiaRicorsivo`):

```
PalindromoListaDoppiaRicorsivo ( testa , coda ) {
  if (testa==coda) return true;
  else if ((testa.succ==coda)&&(testa.dato==coda.dato)) return true;
  else if (testa.dato==coda.dato) {
    return PalindromoListaDoppiaRicorsiva(testa.succ,coda.pred);
  }
  else return false;
  }
}
```

Soluzione dell'esercizio 29 (Struttura dati astratta insiemi di valori): Prima soluzione con Difference in $O(n1 \times n2)$:

```
Contains( set , x ) {
  l = set;
  while (l!=null) {
    if (l.dato==x) return true;
    l = l.succ;
  }
  return false;
}
```

```
Insert( set , x ) {
  if (!Contains(set,x)) {
    nuovo = new Object();
    nuovo.dato = x;
    nuovo.succ = set;
    set = nuovo;
  }
}

Remove( set , x ) {
  l = set;
  if (l.dato==x) {
    set = l.succ;
  }
  else {
    p = l;
    l = l.succ;
    trovato = false;
    while ((l!=null) && (!trovato)) {
      if (l.dato==x) {
        p.succ = l.succ;
        trovato = true;
      }
      else {
        p = l;
        l = l.succ;
      }
    }
  }
}

Union( set1 , set2 ) {
  newset = null;
  l1 = set1;
  while (l1!=null) {
    Insert(newset,l1.dato);
    l1=l1.succ;
  }
  l2 = set2;
  while (l2!=null) {
    Insert(newset,l2.dato);
    l2=l2.succ;
  }
```

```
  return newset;
}

Difference( set1, set2 ) {
  newset = null;
  l = set1;
  while (l!=null) {
    if (!Contains(set2,l.dato))
      Insert(newset,l.dato);
    l=l.succ;
  }
  return newset;
}

Empty( set ) {
  return (set==null);
}
```

Seconda soluzione con Difference in $O(n1+n2)$ (l'idea è di mantenere la lista ordinata):

```
Contains( set , x ) {
  l = set;
  while (l!=null) {
    if (l.dato==x) return true;
    if (l.dato>x) return false;
    l = l.succ;
  }
  return false;
}

Insert( set , x ) {
  nuovo = new Object();
  nuovo.dato = x;
  l = set;
  if (l=null) {
    // l'insieme e' vuoto
    nuovo.succ = null;
    set = nuovo;
  }
  else if (l.dato>=x) {
    // x e' minore del primo elemento quindi va inserito in testa alla lista
    nuovo.succ = set;
    set = nuovo;
  }
```

```
  else {
    // x va inserito in mezzo alla lista
    // p riferisce all'elemento precedente al punto in cui inserire x
    p = l;
    l = l.succ;
    inserito = false;
    while ((l!=null) && (!inserito)) {
      if (l.dato>x) {
        nuovo.succ = l;
        p.succ = nuovo;
        inserito = true;
      }
      else if (l.dato==x) {
        // se l'elemento c'e' gia' non lo inseriamo di nuovo
        inserito = true;
      }
      else {
        p = l;
        l = l.succ;
      }
    }
  }
}

Remove( set , x ) {
  l = set;
  if (l.dato==x) {
    set = l.succ;
  }
  else {
    // uso di p simile a Insert
    p = l;
    l = l.succ;
    trovato = false;
    finito = false;
    while ((l!=null) && (!trovato) && (!finito)) {
      if (l.dato==x) {
        p.succ = l.succ;
        trovato = true;
      }
      else if (l.dato>x) {
        finito = true;
      }
      else {
```

```
        p = l;
        l = l.succ;
      }
    }
  }
}

Union( set1 , set2 ) {
  l1 = set1;
  l2 = set2;

  // creo il primo elemento del risultato
  primoelemento = new Object();
  if (l1!=null) {
    primoelemento.dato = l1.dato;
    l1 = l1.succ;
  }
  else if (l2!=null) {
    primoelemento.dato = l2.dato;
    l2 = l2.succ;
  }
  else {
    // se entrambi gli insiemi sono vuoi l'esecuzione termina qui
    return null;
  }
  primoelemento.succ = null;

  // avendo già creato il primo elemento posso definire un
  // puntatore da far scorrere lungo la lista del risultato
  lnew = primolemento;
  while ((l1!=null) && (l2!=null)) {
    // scorro i due insiemi contemporanemente inserendo gli
    // elementi nel risultato ordinatamente
    lnew.succ = new Object();
    lnew = lnew.succ;
    if (l1.dato<l2.dato) {
      lnew.dato = l1.dato;
      l1 = l1.succ;
    }
    else if (l1.dato>l2.dato) {
      lnew.dato = l2.dato;
      l2 = l2.succ;
    }
    else if (l1.dato==l2.dato) {
```

```
      lnew.dato = l1.dato;
      l1 = l1.succ;
      l2 = l2.succ;
    }
  }

  // al piu' uno dei due seguenti cicli verra' eseguito
  // a seconda di quale insieme ho finito di scorrere per primo
  // nel ciclo precedente
  while (l1!=null) {
    lnew.succ = new Object();
    lnew = lnew.succ;
    lnew.dato = l1.dato;
  }
  while (l2!=null) {
    lnew.succ = new Object();
    lnew = lnew.succ;
    lnew.dato = l2.dato;
  }

  lnew.succ = null;
  return primoelemento;
}


Difference( set1, set2 ) {
  l1 = set1;
  l2 = set2;
  nuovalista = null;
  listavuota = true;

  // procedimento simile a Union. Questa volta invece che gestire
  // il primo elemento del risultato separatamente, uso una
  // variabile listavuota che mi fa gestire il primo elemento
  // in maniera diversa dentro al ciclo
  while ((l1!=null) && (l2!=null)) {
    if (l1.dato<l2.dato) {
      if (listavuota) {
        nuovalista = new Object();
        nuovalista.dato = l1.dato;
        lnew = nuovalista;
        listavuota = false;
      }
      else {
```

```
        lnew.succ = new Object();
        lnew = lnew.succ;
        lnew.dato = l1.dato;
      }
    else if (l1.dato>l2.dato) {
      l2 = l2.succ;
    }
    else {
      l1 = l1.succ;
      l2 = l2.succ;
    }
  }

  while (l1!=null) {
    lnew.succ = new Object();
    lnew = lnew.succ;
    lnew.dato = l1.dato;
    lnew.succ = null;
  }
}

Empty( set ) {
  return (set==null);
}
```

Soluzione dell'esercizio 30 (Struttura dati astratta server di posta):

```
NewMessage( dest , msg ) {
  nm = new Object();
  nm.destinatario = dest;
  nm.messaggio = msg;
  nm.succ = lista;
  lista = nm;
}

GetMessages( dest ) {
  l=lista;
  p=null;
  listadest = null;
  listadest_vuota = true;
  ld = null;

  fine_messaggi = false;
  while (!fine_messaggi) {
```

```
    trovato_messaggio = false;

    // cerco il primo messaggio per dest mantenendo p come
    // riferimento all'elemento precedente
    while ((!trovato_messaggio) && (!fine_messaggi)){
      if (l==null) fine_messaggi = true;
      else if (l.destinatario==dest)
        trovato_messaggio = true;
      else {
        p=l;
        l=l.succ;
      }
    }

    if (trovato_messaggio) {
      if (listadest_vuota) {
        // passaggio delicato: rimuovo l'elemento da lista e lo
        // inserisco in testa a listadest facendo attenzione
        // a mantere corretti p ed l in quanto mi serviranno ancora
        p.succ = l.succ;
        listadest = l;
        l = l.succ;
        listadest.succ=null;
        ld = listadest;
        listadest_vuota = false;
      } else {
        // passaggio delicato: rimuovo l'elemento da lista e lo
        // inserisco in mezzo a listadest (usando ld) facendo attenzione
        // a mantere corretti p, l ed ld in quanto mi serviranno ancora
        p.succ = l.succ;
        ld.succ = l;
        l = l.succ;
        ld = ld.succ;
        ld.succ=null;
      }
    }
  }
  return listadest;
}

Contains( dest ) {
  l = lista;
  while (l!=null) {
    if (l.destinatario==dest) return true;
```

```
    else l=l.succ;
  }
  return false;
}
```