SAPIENZA
UNIVERSITÀ DI ROMA

Esercizi ragionati relativi agli insegnamenti di

Informatica generale
e
Introduzione agli Algoritmi

Prof. Tiziana Calamoneri
Prof. Giancarlo Bongiovanni

Questi esercizi ragionati si riferiscono ai corsi di Informatica Generale per il Corso di Laurea in Matematica e di Introduzione agli Algoritmi per il Corso di Laurea in Informatica, e vanno associati alle relative dispense, di cui vengono richiamati i numeri dei capitoli.

Gli studenti di questo secondo corso possono fare a meno degli esercizi riguardanti i grafi, argomento che viene svolto soltanto all'interno del corso di Informatica Generale.

Questi esercizi rispecchiano piuttosto fedelmente il livello di dettaglio che viene richiesto durante l'esame scritto, ma allo stesso tempo, vogliono condurre lo studente verso il corretto modo di ragionare, per cui possono sembrare a volte prolissi.

Sono benvenuti esercizi svolti dagli studenti, che verranno uniformati a questi ed annessi al presente documento.

Gli studenti interessati a contribuire possono mandare il loro lavoro a:

calamo@di.uniroma1.it ed a bongio@di.uniroma1.it

Licenza 2014 Giancarlo Bongiovanni e Tiziana Calamoneri

Distribuzione Creative Commons

Il lettore ha libertà di riprodurre, stampare, inoltrare via mail, fotocopiare, distribuire questa opera alle seguenti condizioni:

- Attribuzione: deve attribuire chiaramente la paternità dell'opera nei modi indicati dall'autore o da chi ha dato l'opera in licenza;
- Non commerciale: non può usare quest'opera per fini commerciali;
- Non opere derivate: Non può alterare o trasformare quest'opera, né usarla per crearne un'altra.

Licenza Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0). Testo completo: http://creativecommons.org/licenses/by-nc-nd/4.0/)

Esercizio 1.

Riferimento ai capitoli: 2. Notazione asintotica

Si consideri il seguente pseudocodice:

```
1.    somma ← 0
2.    for i = 1 to n do
3.       somma ← somma + i
4.    return somma
```

e si calcoli il costo computazionale asintotico nel caso peggiore.

Si dica, inoltre, se l'algoritmo descritto dallo pseudocodice dato è efficiente per il problema che esso si prefigge di risolvere.

Soluzione.

Come primo passo, dobbiamo riconoscere quale sia il parametro che guida la crescita del costo computazionale. In casi come questo, in cui compare un elenco di n elementi, è immediato definire n come parametro.

Quindi il costo sarà una funzione $T(n)$.

Secondo il modello RAM, una riga i può richiedere un tempo diverso da un'altra, ma ognuna impiega un tempo costante c_i. Poiché la riga 2. è un ciclo che viene ripetuto n volte, si ha:

$$T(n) = c_1 + c_4 + c_2 (n+1) + n\, c_3 = an + b$$

per opportuni valori di a e b. Ne consegue che $T(n) = \Theta(n)$.

E' immediato riconoscere che lo pseudocodice descrive l'algoritmo di somma dei primi n interi. Tuttavia, esso non è l'algoritmo più efficiente.

Si consideri, infatti, il seguente pseudocodice:

```
1.    return n*(n+1)/2
```

il suo costo computazionale è, ovviamente $T(n) = c_1 = \Theta(1)$.

Esercizio 2.

Riferimento ai capitoli: 5. Equazioni di ricorrenza

Si risolva la seguente equazione di ricorrenza con due metodi diversi, per ciascuno

dettagliando il procedimento usato:

$T(n) = T(n/2) + \Theta(n^2)$ se $n > 1$

$T(1) = \Theta(1)$

tenendo conto che *n* è una potenza di 2.

Soluzione.

Utilizziamo dapprima il metodo principale, che è il più rapido, e poi verifichiamo la soluzione trovata con un altro metodo.

Le costanti *a* e *b* del teorema principale valgono qui 2 e 1, rispettivamente, quindi $log_b\, a = 0$ ed $n^{log_b a} = 1$, per cui, ponendo $\varepsilon < 2$, siamo nel caso 3. del teorema, se vale che $af(n/b) \leq cf(n)$.

Poiché $f(n)$ è espressa tramite notazione asintotica, per verificare la disuguaglianza, dobbiamo eliminare tale notazione, e porre ad esempio $f(n)= \Theta(n^2)= hn^2 + k$.

Non è restrittivo supporre $h, k \geq 0$ visto che ci troviamo in presenza di un costo computazionale e quindi non è sensato avere tempi di esecuzione negativi.

Ci chiediamo se esista una costante *c* tale che $h(n/2)^2 + k \leq c(hn^2 + k)$.

Risolvendo otteniamo: $hn^2(1/4 - c) + k(1 - c) \leq 0$ che è verificata ad esempio per $c = 1/2$.

Ne possiamo dedurre che $T(n)= \Theta(n^2)$.

Come secondo metodo usiamo quello iterativo.

Dall'equazione di ricorrenza deduciamo che:

$$T\left(\frac{n}{2}\right) = T\left(\frac{n}{2^2}\right) + \Theta\left(\left(\frac{n}{2}\right)^2\right)$$

$$T\left(\frac{n}{2^2}\right) = T\left(\frac{n}{2^3}\right) + \Theta\left(\left(\frac{n}{2^2}\right)^2\right)$$

e così via. Sostituendo:

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(n^2) =$$

$$= T\left(\frac{n}{2^2}\right) + \Theta\left(\left(\frac{n}{2}\right)^2\right) + \Theta(n^2) =$$

$$= T\left(\frac{n}{2^3}\right) + \Theta\left(\left(\frac{n}{2^2}\right)^2\right) + \Theta\left(\left(\frac{n}{2}\right)^2\right) + \Theta((n)^2) =$$

$$\ldots$$

$$= T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} \Theta\left(\left(\frac{n}{2^i}\right)^2\right)$$

Continuiamo ad iterare fino al raggiungimento del caso base, cioè fino a quando $n/2^k = 1$, il che avviene se e solo se $k = \log n$.

L'equazione diventa così:

$$T(n) = T(1) + \sum_{i=0}^{\log n - 1} \Theta\left(\left(\frac{n}{2^i}\right)^2\right) = \Theta(1) + n^2 \sum_{i=0}^{\log n - 1} \Theta\left(\frac{1}{4^i}\right)$$

Ricordando che

$$\sum_{i=0}^{b} a^i = \frac{a^{b+1} - 1}{a - 1}$$

otteniamo infine:

$$T(n) = \Theta(1) + n^2\, \Theta\left(\frac{1 - \left(\frac{1}{4}\right)^{\log n}}{1 - \frac{1}{4}}\right) = \Theta(n^2).$$

Anche se forse superfluo, concludiamo con l'osservare che i due metodi ***devono*** portare allo stesso risultato o, quanto meno a risultati compatibili, altrimenti bisogna concludere che si è fatto un errore.

Esercizio 3.

Riferimento ai capitoli: 5. Equazioni di ricorrenza

Si risolva la seguente equazione di ricorrenza con tutti e quattro i metodi, per ciascuno dettagliando il procedimento usato:

$T(n) = T(n/3) + T(2n/3) + \Theta(n)$ *se* $n > 1$

$T(1) = \Theta(1)$

Soluzione.

Osserviamo dapprima che il <u>metodo principale</u> non può essere utilizzato, in quanto l'equazione di ricorrenza non è del tipo $T(n)=aT(n/b) +f(n)$.

Procediamo con il <u>metodo iterativo</u>. Si vede subito che i calcoli diventano subito difficili da gestire, quindi utilizziamo la disuguaglianza $T(n/3) \leq T(2n/3)$, sperando di arrivare ad una limitazione inferiore e ad una superiore che coincidano in termini asintotici.

Maggiorando e minorando rispettivamente, ottenendo le due disequazioni:

$T(n) \leq 2T(2n/3)+\Theta(n)$	e	$T(n) \geq 2T(n/3) +\Theta(n)$
$T(1)=\Theta(1)$		$T(1)=\Theta(1)$

Come verifica preliminare, possiamo applicare il metodo principale a queste due ricorrenze, ottenendo che $T(n)=O(n^{\log_{3/2} 2})$ e $T(n)=\Omega(n)$; in effetti $\log_{3/2} 2$ è circa *1.7* quindi sfortunatamente le due limitazioni non coincidono, anche se sono piuttosto vicine. Ovviamente, il metodo iterativo dovrà darci gli stessi risultati. Vediamo:

$$T(n) \leq 2T(2n/3)+\Theta(n) \leq 2(2T(2^2n/3^2)+ \Theta(2n/3))+\Theta(n) = 2^2T(2^2n/3^2)+2\Theta(2n/3)+\Theta(n) \leq$$

$$\ldots \leq 2^k T\left(\frac{2^k n}{3^k}\right) + \sum_{i=0}^{k-1} 2^i \Theta\left(\frac{2^i n}{3^i}\right) = 2^k T\left(\frac{2^k n}{3^k}\right) + \Theta(n) \sum_{i=0}^{k-1} \left(\frac{4}{3}\right)^i \leq$$

… finché $2^k n/3^k=1 \Leftrightarrow k=\log_{3/2} n$…

$$\ldots \leq 2^{\log_{3/2} n} T(1) + \Theta(n) \sum_{i=0}^{\log_{\frac{3}{2}} n-1} \left(\frac{4}{3}\right)^i = 2^{\log_2 n \log_{\frac{3}{2}} 2} \Theta(1) + \Theta(\mathrm{n})\Theta(\left(4/3\right)^{\log_{\frac{3}{2}} n}) =$$

$$=\Theta\left(n^{\log_{\frac{3}{2}} n}\right) + \Theta(n)\Theta\left(2^{\log_{\frac{3}{2}} n} \left(\frac{2}{3}\right)^{\log_{\frac{3}{2}} n}\right) = \Theta\left(n^{\log_{\frac{3}{2}} n}\right) + \Theta\left(n^{\log_{\frac{3}{2}} n}\right) = \Theta\left(n^{\log_{\frac{3}{2}} n}\right).$$

Da qui deduciamo che $T(n)=O(n^{\log_{3/2} 2})$.

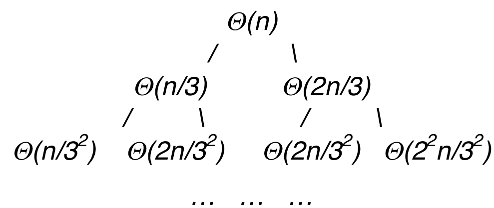
$T(n) \geq 2T(n/3) + \Theta(n) \geq \ldots \geq 2^k T\left(\frac{n}{3^k}\right) + \sum_{i=0}^{k-1} 2^i \Theta\left(\frac{n}{3^i}\right) \geq \ldots$

…finché $n/3^k=1 \Leftrightarrow k=\log_3 n$…

$\ldots\ 2^{\log_2 n \log_2 3}\Theta(1) + \Theta(n) \sum_{i=0}^{\log_3 n-1} (\frac{2}{3})^i = \Theta(n^{\log_2 3}) + \Theta(n) = \Theta(n).$

Da qui deduciamo che $T(n)=\Omega(n)$.

Utilizziamo ora il metodo dell'albero per vedere se riusciamo ad ottenere un limite asintotico stretto. Si può costruire un albero binario la cui parte superiore è così fatta:



E' abbastanza semplice rendersi conto che il contributo di ogni livello è sempre lo stesso, pari a $\Theta(n)$. Cerchiamo di valutare l'altezza di questo albero. Purtroppo non siamo di fronte ad un albero binario completo, ed in particolare il ramo sinistro sarà più corto di quello destro, infatti il numero di livelli nel ramo sinistro è $\log_3 n$ mentre il numero di livelli nel ramo destro è $\log_{3/2} n$; ogni ramo intermedio avrà un numero di livelli compreso tra i due.

Per questo, dobbiamo di nuovo maggiorare e minorare il valore di $T(n)$. Ricordando che il contributo di ciascun livello è sempre $\Theta(n)$, abbiamo:

$$\log_3 n\ \Theta(n) \leq T(n) \leq \log_{3/2} n\ \Theta(n).$$

Se utilizziamo la formula per la trasformazione della base dei logaritmi $\log_a n=\log_b n\ \log_a b$, otteniamo che sia il primo che il terzo membro della precedente catena di disuguaglianze è pari a $\Theta(n \log n)$, per cui in questo caso siamo riusciti a trovare due limitazioni coincidenti, che ci consentono di valutare in senso stretto la soluzione della ricorrenza. Osserviamo che questa soluzione è compatibile con le limitazioni inferiore e superiore ottenuti con il metodo iterativo.

Risolviamo, infine, con il metodo di sostituzione, dapprima eliminando la notazione asintotica dall'equazione di ricorrenza:

$T(n) = T(n/3) + T(2n/3) + cn$ se $n > 1$

$T(1) = d$

e poi ipotizzando una soluzione. Visto ciò che abbiamo ottenuto con il metodo dell'albero, possiamo ipotizzare $T(n) \leq an \log n$, dove a è una costante da determinare. Poiché quando $n=1$ $\log n=0$, dobbiamo recuperare un altro caso base. Consideriamo $n=2$; in tal caso $T(n)=2d+2c=d'$, per cui aggiungiamo il caso base $T(2)=d'$.

Ora procediamo per induzione.

Passo base: Se $n=2$, $d'=T(2) \leq 2a \log 2 = 2a$; questa relazione impone che la disuguaglianza sia verificata se $a \geq d'/2$.

Passo induttivo: Se $n > 2$, sfruttando l'ipotesi induttiva, possiamo scrivere:

$T(n) = T(n/3) + T(2n/3) + cn \leq an/3 \log n/3 + 2an/3 \log 2/3n + cn =$

$= an/3 \log n - an/3 \log 3 + 2an/3 \log n - 2an/3 \log 3/2 + cn \leq an \log n + cn$ per ogni a.

Abbiamo così dimostrato che $T(n)=O(n \log n)$.

Ipotizziamo ora la soluzione $T(n) \geq sn \log n + tn$ per qualche costante s e t da determinare.

Anche in questo caso, procediamo per induzione.

Passo base: Se $n=1$, $d=T(1) \geq t$; la disuguaglianza è verificata se $t \leq d$.

Passo induttivo: Se $n > 1$, sfruttando l'ipotesi induttiva ed evitando di ripetere alcuni passaggi già dettagliati nel caso precedente, possiamo scrivere:

$T(n) = T(n/3) + T(2n/3) + cn \geq sn/3 \log n - sn/3 \log 3 + tn/3 + 2sn/3 \log n - 2sn/3 \log 3/2 +$
$+ 2tn/3 + cn = sn \log n - sn/3 (\log 3 + 2 \log 3/2) + tn + cn$

Osservando che $\log 3 < 2$ e che $\log 3/2 < 1$ (attenzione: usiamo < e non > perché c'è il segno – davanti alla parentesi!), possiamo minorare l'espressione precedente con:

$sn \log n - 4/3sn + tn + cn \geq sn \log n + cn$ se $-4/3sn+tn \geq 0$. Deduciamo che la nostra ipotesi è vera se s e t sono tali che $t \geq 4/3\, s$.

Abbiamo così dimostrato che $T(n)= \Omega(n \log n)$.

Dai due risultati parziali concludiamo infine affermando che $T(n)= \Theta(n \log n)$.

Esercizio 4.

Riferimento ai capitoli: 4. Ricorsione; 5. Equazioni di ricorrenza

Dati in input *n* e *k* interi positivi, si scrivano due funzioni, una iterativa ed una ricorsiva che restituiscano in output l'intero n^k. Si studi poi il costo computazionale delle due funzioni.

Soluzione.

Le due funzioni sono entrambe molto semplici, e possono essere scritte come segue:

```
Power_ric(n, k)

   1. if (k==0) return 1;
   2. return n*Power_ric(n, k-1);
```

```
Power_it(n,k)

   1. if (k==0) return 1;
   2. int i; aux ← n;
   3. for (i=1; i<=k; i++)
   4.     aux ← n*aux;
   5. return aux;
```

Per calcolare il costo computazionale, individuiamo dapprima i parametri della funzione che, ragionevolmente, saranno *n* e *k*, quindi cerchiamo una funzione $T(n,k)$.

Cominciamo dalla funzione iterativa, che non richiede particolari strumenti: le linee 1., 2. e 5. hanno costo costante; il ciclo della linea 3. è ripetuto esattamente *k* volte e contiene, al suo interno, una sola istruzione che viene eseguita in tempo costante. Pertanto, $T(n,k)= \Theta(1)+ \Theta(k)= \Theta(k)$, deducendo così che la funzione $T(n,k)$ è, in realtà, semplicemente $T(k)$.

Per calcolare il costo computazionale della funzione ricorsiva, osserviamo che la linea 1. richiede tempo costante e costituisce il caso base della ricorsione. La linea 2., invece, è il cuore della ricorsione ed ha costo $\Theta(1)+T(n, k-1)$. Abbiamo pertanto:

$T(n,0)= \Theta(1)$

$T(n,k)= \Theta(1)+T(n,k-1)$.

Risolvendo questa equazione di ricorrenza ad esempio tramite il metodo iterativo, otteniamo:

$T(n,k)= \Theta(1)+T(n, k-1)= \Theta(1)+ \Theta(1)+T(n, k-2)=\ldots=j\, \Theta(1)+T(n, k-j)$

Procediamo a svolgere l'equazione fino a quando non giungiamo al caso base, cioè fino a che $j=k$. In tal caso si ha:

$T(n,k)=kT(n, k-k)=k\, \Theta(1)= \Theta(k)$.

Cioè, per entrambe le versioni otteniamo lo stesso costo computazionale.

Si osservi che con un criterio di misura diverso (non più di costo uniforme ma di costo logaritmico), effettivamente il costo computazionale è funzione sia di *k* che di *n*, poiché ad

esempio ciascuna moltiplicazione per *n* non impiega tempo costante ma funzione almeno di log *n.* Per semplicità, non approfondiamo ulteriormente questo argomento.

Esercizio 5.

Riferimento ai capitoli: 4. Ricorsione; 5. Equazioni di ricorrenza

Dati in input un intero *n* ed un vettore $A=(a_0, \ldots, a_{n-1})$ di n numeri interi, si scrivano due funzioni, una iterativa ed una ricorsiva, che restituiscano in output la somma $a_0+a_1+\ldots+a_{n-1}$. Si studi poi il costo computazionale delle due funzioni.

Soluzione.

Questo esercizio è del tutto analogo al precedente, e pertanto possiamo essere più stringati nelle spiegazioni.

Le due funzioni possono essere scritte come segue:

```
Somma_Ric(A[], n)

   1. if (n=0) return 0;
   2. return A[n-1]+Somma_Ric(A[],
      n-1)
```

```
Somma_Iter(A[], n)

   1. if (n=0) return 0;
   2. int i, aux=A[0];
   3. for (i=1; i<n; i++)
   4.    aux=aux+A[i];
   5. return aux;
```

Qui il parametro del costo computazionale è ovviamente *n;* nel caso iterativo esso è facilmente $T(n)= \Theta(1)+n\, \Theta(1)= \Theta(n)$ mentre nel caso ricorsivo si ha:

$T(0)= \Theta(1)$

$T(n)= \Theta(1)+ \Theta(1)T(n-1)$

che è analoga all'equazione di ricorrenza incontrata nel precedente esercizio.

Esercizio 6.

Riferimento ai capitoli: 4. Ricorsione; 5. Equazioni di ricorrenza

Dati in input un intero *n* ed un vettore $A=(a_0, \ldots, a_{n-1})$ di *n* numeri reali, si scrivano due funzioni, una iterativa ed una ricorsiva, che restituiscano il valore 1 se il vettore è palindromo, il valore 0 altrimenti.

Soluzione.

Verificare se un vettore è palindromo significa domandarsi se a_0 e a_{n-1}, a_1 e a_{n-2}, ecc. hanno a coppie lo stesso valore. L'unica piccola difficoltà che può dunque presentare quest'esercizio è la gestione degli indici.

La funzione ricorsiva può essere costruita riportandosi al problema di dimensione inferiore che si ottiene eliminando dal vettore il primo e l'ultimo elemento. Sappiamo che modificando il valore di *n* possiamo agire sulla parte finale del vettore; introduciamo una nuova variabile *m* che è, inizialmente, settata a 0, ed indica il primo elemento da considerare nel vettore. Cioè:

```
Palindromo_Ric(A[], n, m)

   1. if (n<m) return 1;
   2. if (A[n-1]=A[m]) return Palindromo_Ric(A[], n-1, m+1);
   3. return 0;
```

Si osservi che la condizione della linea 1, corrispondente al caso base, è verificata solo se non si è mai passati per la linea 3, cioè se fino ad ora tutti i controlli hanno dato esito positivo. E' chiaro, quindi che in tal caso la funzione restituisca 1, indicando cioè che il vettore nullo è, ovviamente, palindromo. Nel caso generale, se una coppia viene trovata non rispondente alla definizione di vettore palindromo, la funzione restituisce 0 senza proseguire con inutili controlli.

Il costo computazionale di questa funzione si può calcolare grazie alla seguente equazione di ricorrenza:

$T(0)= \Theta(1)$

$T(n)= \Theta(1)+ \Theta(1)T(n\text{-}2)$

Che si risolve analogamente ai casi precedenti, ottenendo $T(n)= n/2\Theta(1)+T(0)= \Theta(n/2)= \Theta(n)$. Tenendo conto del fatto che in effetti la funzione ricorsiva non viene richiamata esattamente n/2 volte ma al più n/2 volte, possiamo scrivere, più correttamente, che $T(n)=O(n)$.

Con identica filosofia, la funzione iterativa può essere così scritta:

```
Palindromo_Iterat(A[], n)

   1. if (n=0) return 1;
   2. int m=0;
   3. while (m<n && A[n-1]=A[m])
   4.    n--; m++;
   5. if n<=m return 1;
   6. return 0;
```

Anche il suo costo computazionale è *T(n)=O(n)* poiché il ciclo while della linea 3 viene eseguito al più *n/2* volte.

Esercizio 7.

Riferimento ai capitoli: 4. Ricorsione; 5. Equazioni di ricorrenza

Dato un vettore *A* di *n* interi, progettare un algoritmo ricorsivo che restituisce il massimo ed il minimo di *A* in tempo *O(n)*. Verificare tale costo computazionale tramite l'impostazione e la risoluzione di un'equazione di ricorrenza.

Soluzione.

Sapendo di dover produrre un algoritmo ricorsivo, dobbiamo decidere come ridurre il problema ad uno o più sottoproblemi. Le uniche possibilità sensate appaiono le seguenti:

- calcolare ricorsivamente il massimo ed il minimo nei due sottovettori destro e sinistro di dimensione *n/2*, confrontare poi i due massimi ed i due minimi dando in output il massimo e minimo globali;
- calcolare ricorsivamente il massimo ed il minimo del sottovettore di dimensione *n-1* ottenuto eliminando il primo (o l'ultimo) elemento, confrontare poi il massimo ed il minimo ottenuti con l'elemento eliminato dando in output il massimo e minimo globali.

Per comprendere quale dei due approcci sia migliore, proviamo a calcolarne il costo computazionale.

Per quanto riguarda il primo metodo, il tempo di esecuzione su un input di *n* elementi, *T(n),* è pari al tempo di esecuzione dello stesso algoritmo su ciascuno dei due sottovettori di *n/2* elementi, più il tempo necessario per confrontare i due massimi ed i due minimi; il caso base si ha quando i sottovettori sono di dimensione 1, per cui il massimo ed il minimo coincidono, e

l'unica operazione da fare è quella di confrontare tra loro massimi e minimi per dare in output i valori globali. L'equazione di ricorrenza è dunque:

$T(n) = 2T(n/2) + \Theta(1)$

$T(1) = \Theta(1)$

Risolvendo questa equazione con il metodo iterativo (ma si può fare anche con il metodo dell'albero), si ottiene:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(1) =$$

$$= 2\left(2T\left(\frac{n}{2^2}\right) + \Theta(1)\right) + \Theta(1) =$$

$$= 2^2\left(2T\left(\frac{n}{2^3}\right) + \Theta(1)\right) + 2\Theta(1) + \Theta(1) =$$

$$\ldots$$

$$= 2^k T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} 2^i \Theta(1)$$

Procediamo fino a quando $n/2^k$ non sia uguale ad 1 e cioè fino a quando $k = \log n$. In tal caso, sostituendo nell'espressione di $T(n)$ e sfruttando il caso base si ha:

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

Studiamo ora il secondo approccio: il tempo di esecuzione dell'algoritmo $T(n)$ su un input di n elementi è pari al tempo di esecuzione dello stesso algoritmo su $n - 1$ elementi più il tempo per confrontare l'elemento rimasto con il massimo ed il minimo trovati; anche qui il passo base si ha quando $n = 1$. L'equazione di ricorrenza allora diventa:

$T(n) = T(n - 1) + \Theta(1)$

$T(1) = \Theta(1)$

Anche questa equazione si può risolvere con il metodo iterativo ottenendo:

$T(n) = T(n - 1) + \Theta(1) = T(n - 2) + \Theta(1) + \Theta(1) = \ldots = T(n - k) + k\Theta(1).$

Procediamo fino a quando $n - k = 1$ cioè fino a quando $k = n - 1$ e sostituiamo nell'equazione:

$$T(n) = T(1) + (n - 1)\,\Theta(1) = \Theta(n)$$

se si tiene conto del caso base.

Deduciamo dai precedenti ragionamenti che entrambi i metodi sono ugualmente corretti ed efficienti.

Esercizio 8.

Riferimento ai capitoli: 4. Ricorsione; 5. Equazioni di ricorrenza; 6. Il problema dell'ordinamento

L'algoritmo Insertion-Sort può essere espresso come una procedura ricorsiva nel modo seguente:

- per ordinare *A[1 … n]*, si ordina in modo ricorsivo *A[1 … n −1]* e poi si inserisce *A[n]* nell'array ordinato *A[1 … n − 1]*.

Si scriva l'equazione di ricorrenza per il tempo di esecuzione di questa versione ricorsiva di insertion sort.

Soluzione.

Denotiamo con $T(n)$ il tempo impiegato nel caso peggiore dalla versione ricorsiva dell'algoritmo Insertion-Sort. Abbiamo che $T(n)$ è pari al tempo necessario per ordinare ricorsivamente un vettore di $n - 1$ elementi (che è uguale a $T(n - 1)$) più il tempo per inserire $A[n]$ (che nel caso peggiore è $\Theta(n)$).

Pertanto la ricorrenza diviene:

$$T(n) = T(n - 1) + \Theta(n).$$

Per quanto riguarda il caso base, esso si ha ovviamente per $n = 1$ ed, in tal caso, il tempo di esecuzione è $\Theta(1)$.

La soluzione dell'equazione di ricorrenza può essere ottenuta con uno qualunque dei metodi studiati. Qui utilizziamo il metodo iterativo:

$$T(n) = T(n - 1) + \Theta(n) =$$

$$= (T(n - 2) + \Theta(n - 1)) + \Theta(n) =$$

$$= ((T(n - 3) + \Theta(n - 2)) + \Theta(n - 1)) + \Theta(n) =$$

$$= \dots =$$

$$= T(n - k) + \sum_{i=0}^{k-1} \Theta(n - i)$$

Raggiungiamo il caso base quando $n - k = 1$, cioè quando $k = n - 1$. In tal caso, l'equazione diventa:

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

Si consiglia al lettore di risolvere l'equazione di riscorrenza con gli altri metodi per esercizio.

Esercizio 9.

Riferimento ai capitoli: 4. Ricorsione; 5. Equazioni di ricorrenza; 6. Il problema dell'ordinamento

Dato un vettore di *n* numeri reali non nulli, progettare un algoritmo efficiente che posizioni tutti gli elementi negativi prima di tutti gli elementi positivi.

Dell'algoritmo progettato si dia:

a) la descrizione a parole
b) lo pseudocodice
c) il costo computazionale.

Soluzione.

Sia *A* il vettore dato in input. Lo scopo dell'algoritmo che dobbiamo progettare è quello di separare i numeri positivi dai negativi, senza introdurre alcun ordine particolare.

Una funzione simile a quella che partiziona il vettore rispetto ad una soglia nel Quicksort potrebbe fare al caso nostro, con l'unica accortezza che la soglia qui deve essere 0. Non avremo difficoltà a gestire elementi del vettore nulli perché il testo ci assicura che non ce ne sono.

La descrizione a parole è dunque la seguente:

- scorri il vettore *A* da sinistra a destra con un indice *i* e da destra a sinistra con un indice *j*; ogni volta che *i* indica un elemento positivo e *j* un elemento negativo esegui uno scambio; prosegui fino a quando il vettore non sia stato completamente visionato.

Lo pseudocodice, del tutto analogo alla funzione di partizionamento del Quicksort, è il seguente:

```
Funzione PartizionaPosNeg (A: vettore; i, j: intero)
1. i ← 1
2. j ← n
3. while (i < j)
4.    while (A[j] > 0) and (i ≤ j)
5.       j ← j - 1
6.    while ((A[i] < 0)and(i ≤ j)
7.       i ← i + 1
8.    if (i < j)
9.       scambia A[i] e A[j]
```

Infine, il costo dell'algoritmo è esattamente lo stesso della funzione a cui ci siamo ispirati, ma ricalcoliamolo qui per completezza:

- le linee 1 e 2 hanno costo *O(1)*;
- i tre `while` alle linee 3, 4 e 6, complessivamente, fanno in modo che gli elementi del vettore vengano tutti esaminati esattamente una volta e le istruzioni all'interno del ciclo impiegano tempo costante.

Il tempo di esecuzione è indipendente dal vettore in input; se ne conclude che il costo è $\Theta(n)$.

Il costo spaziale è anch'esso lineare in *n*, visto che l'algoritmo utilizza il vettore di input *A* di lunghezza *n* e due contatori.

Questo esercizio si può anche risolvere osservando che il problema di separare gli elementi positivi dai negativi si può ridurre a quello di ordinare *n* numeri interi nel range [0,1]; si può quindi applicare una modifica dell'algoritmo di Counting Sort, come segue.

Sia *A* il vettore di input e *B* il vettore di output, entrambi di lunghezza *n*.

```
Funzione SeparaPosNeg2(A, B)
   1.  pos ← n;
   2.  neg ← 1;
   3.  for i = 1 TO n do
   4.     if A[i]>0 then
   5.        B[pos] ← A[i];
   6.        pos ← pos-1;
   7.     else
   8.        B[neg] ← A[i];
   9.        neg ← neg+1;
```

Si osservi che non abbiamo bisogno del terzo vettore *C* utilizzato dal Counting Sort, essendo i dati semplicemente degli interi (senza dati satellite). Osserviamo che l'algoritmo proposto posiziona gli elementi positivi nell'ordine dato e gli elementi negativi nell'ordine inverso.

Il costo computazionale di questo algoritmo è, ovviamente, lineare in *n*, visto che il suo pseudocodice è costituito da un ciclo `for` che esegue operazioni costanti ad ogni iterazione. Il costo spaziale è anch'esso lineare in *n*, visto che l'lagoritmo utilizza, oltre al vettore di input *A*, un vettore ausiliario *B* di lunghezza *n* e tre contatori.

Concludiamo osservando che anche un qualunque algoritmo di ordinamento (per confronti) avrebbe assolto allo scopo, facendo anzi più del richiesto, ma richiedendo anche un costo computazionale di $\Omega(n \log n)$. Se ne deduce che una tale soluzione è senz'altro da scartare.

Esercizio 10.

Riferimento ai capitoli: 6. Il problema dell'ordinamento

Dati due insiemi di interi $A = \{a_1, a_2, \ldots, a_n\}$ e $B = \{b_1, b_2, \ldots, b_m\}$, sia C la loro intersezione, ovvero l'insieme degli elementi contenuti sia in A che in B.

Si discutano, confrontandoli, i costi computazionali dei seguenti due approcci al problema:

- applicare un algoritmo "naive" che confronta elemento per elemento;
- applicare un algoritmo che prima ordina gli elementi di A e B.

Soluzione.

La soluzione "naive" è la prima e più semplice che viene in mente: per ogni elemento di A (cioè `for` un certo indice i che va da 1 ad n) si scorre B (cioè `for` un certo altro indice j che va da 1 ad m) per vedere se l'elemento a_i è presente o no in B. Ne segue che il costo computazionale di questo approccio è $O(nm)$. Scriviamo O e non Θ perché, mentre il ciclo `for i` viene eseguito completamente, il ciclo `for j` si può interrompere appena l'elemento a_i è uguale all'elemento b_j. Pertanto, nel caso migliore, in cui tutti gli elementi di A sono uguali tra loro ed al primo elemento di B, il costo è $\Theta(n)$, ma nel caso peggiore, in cui l'intersezione è vuota, il costo è $\Theta(nm)$.

Tra l'altro, l'analisi del caso migliore fa venire in mente che una trattazione a parte meriterebbe il caso della duplicazione degli elementi; esso si risolve facilmente ricordando che un insieme, per definizione, non ammette ripetizioni al suo interno, e quindi dobbiamo considerare gli a_i tutti distinti tra loro, così come i b_j. Ne discende che il caso migliore presentato non è ammissibile, ma questo non cambia il costo del caso peggiore.

Consideriamo ora il secondo approccio.

Si ordinino gli insiemi A e B in modo crescente, e poi si confrontino i due vettori come segue:

- si pongano un indice i all'inizio di A e un indice j all'inizio di B;
- si confronti $A[i]$ con $B[j]$
    - se essi sono uguali, l'elemento si inserisce in C e si incrementano sia i che j,
    - se $A[i]$ è minore, si incrementa i,
    - se $B[j]$ è minore, si incrementa j;
- si ripeta dal confronto finché non si raggiunga la fine di uno dei due vettori.

Il costo computazionale $T(n, m)$ di questo approccio deve tener conto del costo dell'ordinamento e di quello del confronto, e pertanto è pari a

$$\Theta(n \log n) + \Theta(m \log m) + \Theta(\max(n, m))$$

Se assumiamo, senza perdere di generalità, che $n \geq m$, si ha che

$$T(n, m) = \Theta(n \log n) + \Theta(n) = \Theta(n \log n).$$

Un'idea alternativa richiede l'ipotesi che tutti gli elementi in ciascun insieme siano distinti. In tal caso, ponendo sempre senza perdere di generalità $n \geq m$, possiamo ordinare il vettore più piccolo –B- con un algoritmo efficiente ($\Theta(m \log m)$) e poi cercare gli elementi di A in B tramite una ricerca binaria. Ciò può essere fatto in tempo $\Theta(n \log m)$. In base all'ipotesi $n \geq m$ si ha che $T(n,m)= \Theta(m \log m)+ \Theta(n \log m)= \Theta(n \log m)$.

Se potessimo, infine, fare alcune ipotesi sugli elementi di *A* e di *B*, in modo da poter applicare uno degli algoritmi di ordinamento lineari, il costo si abbasserebbe ulteriormente a $\Theta(n)$.

Esercizio 11.

Riferimento ai capitoli: 6. Il problema dell'ordinamento

Siano dati due vettori $A[1..n]$ e $B[1..m]$, composti da $n \geq 1$ ed $m \geq 1$ numeri reali, rispettivamente. I vettori sono emtrambi ordinati in senso crescente. *A* e *B* non contengono valori duplicati; tuttavia, uno stesso valore potrebbe essere presente una volta in *A* e una volta in *B*.

Progettare un algoritmo iterativo efficiente che stampi i numeri reali che appartengono all'unione dei valori di *A* e di *B*; l'unione va intesa in senso insiemistico, quindi gli eventuali valori presenti in entrambi i vettori devono essere stampati solo una volta. Ad esempio, se $A = [1,3,4,6]$ e $B = [2,3,4,7]$, l'algoritmo deve stampare *1, 2, 3, 4, 6, 7*.

Di tale algoritmo:

1. Si dia una descrizione a parole;
2. Si mostri in dettaglio come funziona sui due vettori dati in esempio sopra;
3. Si scriva lo pseudocodice;
4. Si determini il costo computazionale, in funzione di *n* ed *m*;

Soluzione.

Osserviamo innanzi tutto che viene posta l'attenzione sul problema dei duplicati: sappiamo che *A* e *B* contengono elementi tutti diversi tra loro ma possono esistere elementi che stanno sia in *A* che in *B*, e questi vanno stampati una volta sola. E' chiaro quindi che, prima di stampare un qualunque elemento di *A* dobbiamo prima verificare se esso sia presente anche in *B* perché, in

tal caso, possiamo fare a meno di stamparlo (infatti sarà stampato quando sarà il momento di stampare gli elementi di *B*). Ora, per verificare se ciascuno tra gli *n* elementi di *A* compare nel vettore *B*, impieghiamo tempo $\Theta(nm)$ se per ciascun elemento di *A* scorriamo *B*. Ma possiamo osservare che, in questo modo, non stiamo usando l'ipotesi che i due vettori siano ordinati. Un modo alternativo di procedere è di eseguire, per ciascun elemento di *A*, una ricerca binaria su *B*, il che ci porterebbe ad un costo di $O(n \log m)$, ma ancora non stiamo sfruttando tutte le ipotesi perché così abbiamo usato il fatto che *B* sia ordinato ma l'ordinamento su *A* non ci serve a niente. Volendo usare l'ordinamento di entrambi i vettori, potremmo proporre un algoritmo simile a quello di fusione del Mergesort, in cui si trascrive solo uno degli elementi uguali. Un tale approccio ci porta ad un costo computazionale di $\Theta(n+m)$, che è anche ottimo visto che –nel caso peggiore- gli elementi da stampare sono proprio *n+m.*

Fatte queste premesse, possiamo rispondere alle domande:

1. Consideriamo i due vettori *A* e *B*, e progettiamo un algoritmo simile a quello di fusione: consideriamo un indice *i* che scorre *A* ed un indice *j* che scorre *B* che partono entrambi da 1*.* Confrontiamo gli elementi *A[i]* e *B[j]* e operiamo in modo diverso a seconda dei casi:

- se *A[i]<A[j]* si stampa *A[i]* e si incrementa *i*
- se *A[i]>A[j]* si stampa *A[j]* e si incrementa *j*
- se *A[i]=A[j]* si stampa *A[i]* e si incrementano sia *i* che *j.*

Appena uno dei due vettori termina, stampiamo tutti gli elementi dell'altro vettore. Osserviamo che, a differenza della funzione di fusione del MergeSort, non abbiamo qui bisogno di un vettore ausiliario.

2. Dati i due vettori *A = [1,3,4,6]* e *B = [2,3,4,7]*, si pongono *i=j=1.* Poiché *A[1]=1<B[1]=2* si stampa 1 e si incrementa *i,* che passa a 2. Si confrontano *A[2]=3* con *B[1]=2* per cui si stampa 2 e si incrementa *j,* che passa a 2. Si confrontano ora i valori di *A[2]=3* e *B[2]=3;* i valori sono uguali per cui se ne stampa solo uno ma entrambi gli indici vengono incrementati, per cui sia *i* che *j* passano a 3. Prima di procedere, osserviamo che non possiamo trovare, nei vettori, altri valori pari a 3 poiché per ipotesi in *A* ed in *B* gli elementi sono tutti distinti. Si confrontano *A[3]=4* e *B[3]=4;* essi sono uguali e quindi ne stampiamo uno ed incrementiamo sia *i* che *j* passano a 4. *A[4]=6<B[4]=7,* quindi stampiamo *6* ed incrementiamo *i* che passa a 5, uscendo fuori dal vettore; concludiamo quindi stampando gli elementi nella parte rimanente del vettore *B*, cioè 7.

3. Questo è un possibile pseudocodice:

```
algoritmo StampaUnione(A[], B[])
i ← 1; j ← 1;
  while (i ≤ n AND j ≤ m) do
       if (A[i] < B[j]) then
              stampa A[i]; i ← i+1;
       else if (A[i] > B[j]) then
              stampa B[j]; j ← j+1;
       else stampa A[i];
             i ← i+1;
             j ← j+1;
  while ( i ≤ n ) do /* stampa la parte finale del vett. non ancora terminato
       stampa A[i]; i ← i+1;
  while ( j ≤ m ) do /* solo uno dei due cicli viene eseguito
       stampa B[j]; j ← j+1;
```

4. Il costo di questo algoritmo $\Theta(n+m)$ in quanto sostanzialmente si effettua una scansione di entrambi i vettori esaminando ciascun elemento una e una sola volta in tempo $\Theta(1)$.

Esercizio 12.

Riferimento ai capitoli: 4. Ricorsione; 5. Equazioni di ricorrenza; 6. Il problema dell'ordinamento

Si consideri lo pseudocodice del seguente algoritmo di ordinamento.

La funzione viene richiamata la prima volta con $i=1$ e $j=n$.

```
Funzione UnAltroSort (A[], i, j)
1.     if (A[i]>A[j])
2.        scambia(A[i], A[j])
3.     if (i+1 ≥ j)
4.        return
5.     k ← ⌊(j-i+1)/3⌋
6.     UnAltroSort(A, i, j-k)     /* si ordinano i primi 2/3 del vettore
7.     UnAltroSort(A, i+k, j)     /* si ordinano gli ultimi 2/3 del vettore
8.     UnAltroSort(A, i, j-k)     /* si ordinano di nuovo i primi 2/3 del vett.
```

1. si scriva l'equazione di ricorrenza che descrive il costo computazionale della funzione nel caso peggiore, dandone giustificazione;
2. si risolva l'equazione trovata usando il teorema principale ed un altro metodo e, per ciascuno, si mostri il procedimento usato;

3. si confronti il tempo di esecuzione del caso peggiore di `UnAltroSort` con quello dell'InsertionSort e del MergeSort, facendo le opportune considerazioni sull'efficienza dell'algoritmo proposto.

Soluzione.

1. Innanzi tutto dobbiamo individuare il parametro dell'equazione di ricorrenza. In questo ci aiutano i commenti, i quali ci suggeriscono che le chiamate ricorsive vengono effettuate su un vettore di lunghezza pari ai 2/3 del vettore originario; pertanto è naturale assumere che il parametro sia proprio la lunghezza del vettore; usando i nomi delle variabili presenti nella funzione, tale parametro si può esprimere come $n=j-i+1$. Questo è in accordo con le linee 6, 7 ed 8, dove le lunghezze dei tre sottovettori sui quali viene richiamata la funzione sono in tutti e tre i casi all'incirca $2/3n$.

Si ricordi ora che un'equazione di ricorrenza si compone della parte relativa al caso generale e quella relativa al caso base.

Cominciamo quindi ad individuare le linee di codice relative al caso base. Esse sono, ovviamente, le righe 3 e 4, e quindi interessano il caso in cui $i+1 \geq j$. Poiché i e j rappresentano rispettivamente l'inizio e la fine del vettore in considerazione, siamo in un caso base tutte le volte che l'elemento successivo a quello di indice i si trova in corrispondenza di j oppure alla sua destra, quindi il caso base si ottiene quando $n \leq 2$ ed, in tal caso, il costo è pari a $\Theta(1)$ poiché vengono eseguite le linee da 1 a 4, tutte operazioni di costo costante.

Nel caso generale, invece, il costo è dato dal contributo $\Theta(1)$ proveniente dalle linee 1-5 e dal contributo delle tre linee 6-8, ciascuna delle quali ha costo $T(2/3\ n)$, per cui:

$T(n)= \Theta(1)+3T(2/3\ n)$

$T(1)=T(2)= \Theta(1).$

2. Questa equazione di ricorrenza si può risolvere tramite teorema principale, tramite metodo iterativo o metodo dell'albero, ed la soluzione è T(n)= $\Theta(n^{\log_{3/2}3})$.

Poiché la soluzione non presenta particolari difficoltà, omettiamo qui i dettagli.

3. Confrontiamo ora il costo della funzione data con quella dell'InsertonSort, $\Theta(n^2)$, e del MergeSort, $\Theta(n \log n)$. Per fare ciò, dobbiamo farci un'idea del valore di $\log_{3/2}3$. Osserviamo che la funzione logaritmica è una funzione monotona crescente, e quindi:

$1 = \log_{3/2}3/2 < 2 = \log_{3/2}9/4 < \log_{3/2}3.$

Pertanto $\log_{3/2}3$ è un valore strettamente maggiore di 2. Ne consegue che il costo di `UnAltroSort` è peggiore sia di quello di MergeSort che di quello di InsertionSort.

Esercizio 13.

Riferimento ai capitoli: 6. Il problema dell'ordinamento

Dato un heap *H* di *n* elementi, descrivere a parole l'algoritmo che cancella un prefissato nodo *i* e riaggiusta l'heap risultante di $n - 1$ elementi.

Valutare il costo computazionale dell'algoritmo presentato.

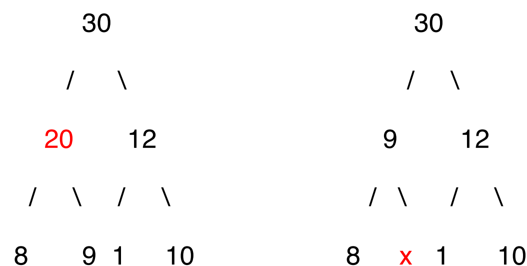
Soluzione.

Il problema si scompone in due parti: la prima consiste nel cercare l'elemento da eliminare e l'altra nel cancellarlo e ripristinare l'heap.

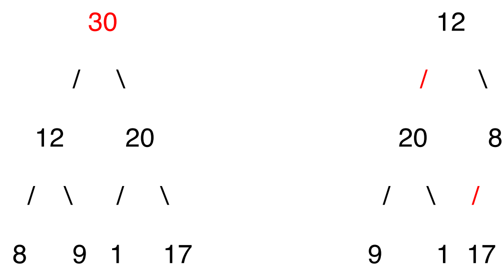
L'elemento *i* da eliminare può essere inteso come l'elemento di posizione *i* oppure l'elemento di chiave *i*. Nel primo caso l'individuazione dell'elemento richiede tempo costante (l'elemento è $H[i]$), nel secondo bisogna effettuare una ricerca, che può richiedere anche tempo lineare in *n*, cioè $O(n)$.

Una volta noto l'elemento, sia esso $H[x]$, è possibile eliminarlo apparentemente in molti modi, alcuni dei quali sono sintetizzati qui:

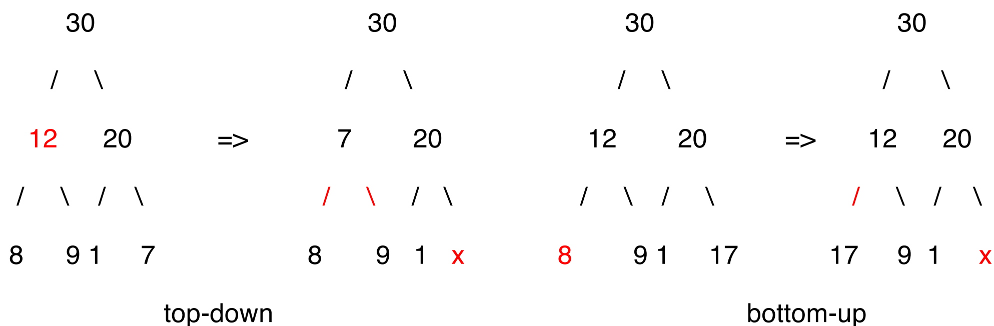
- si può portare verso le foglie l'elemento da cancellare scambiandolo con il maggiore dei suoi figli; una volta che l'elemento ha raggiunto le foglie, si elimina semplicemente; questo algoritmo e' errato perche' un heap è un albero binario completo o quasi completo e questo approccio può far perdere questa proprietà; si veda, ad esempio, nell'esempio qui sotto, cosa succede quando si tenta di eliminare il nodo di chiave 20 con questo approccio:



- si può shiftare ogni elemento *H[x+1], H[x+2], …, H[n]* di una posizione verso sinistra e poi riaggiustare l'heap, ma lo shift a sinistra provoca un disallineamento dei figli rispetto ai padri per cui nessun nodo è garantito mantenere la proprietà dell'heap, che – per essere aggiustato – dovrebbe essere interamente ricostruito; si veda l'esempio qui sotto:



- si può scambiare H[x] con H[n], cioè con la foglia più a sinistra, eliminare tale foglia e riaggiustare l'heap risultante. Per fare ciò, bisogna confrontare il nuovo H[x] con il maggiore dei suoi figli; se H[x] risulta minore, si può applicare la funzione di aggiustamento dell'heap top-down; se H[x] sembra in posizione corretta rispetto ai suoi figli, non è ancora detto che l'heap sia corretto, bisogna infatti confrontarlo con il padre, se questo è minore di H[x] si deve procedere all'aggiustamento bottom-up:



Se escludiamo la ricerca del nodo, l'algoritmo di cancellazione e riaggiustamento ha una complessità pari al massimo tra la complessità dell'aggiustamento top-down e di quello bottom-up. Poiché entrambi si possono eseguire in tempo *O(log n)*, ne consegue che *O(log n)* è proprio il costo dell'algoritmo presentato.

Esercizio 14.

Riferimento ai capitoli: 7. Strutture dati fondamentali

Siano date k liste ordinate in cui sono memorizzati globalmente n elementi. Descrivere un algoritmo con tempo $O(n \log k)$ per fondere le k liste ordinate in un'unica lista ordinata.

Soluzione.

Si consideri, innanzi tutto, il classico algoritmo di fusione applicato a due liste:

> al generico passo i, sono già stati sistemati nella lista finale i elementi, e vi sono due puntatori alle teste (contenenti gli elementi più piccoli) delle due liste; si individua quale tra i due elementi puntati sia il minore, e questo viene inserito nella lista risultante in coda. Si passa quindi al passo $i+1$.

Il numero di passi complessivi è ovviamente pari ad n, cioè al numero complessivo di elementi.

E' naturale cercare di generalizzare questo algoritmo a k liste ordinate, purché si sia in grado di calcolare il minimo fra k elementi. Banalmente, questo si può fare in tempo $\Theta(k)$, producendo quindi un algoritmo di fusione di k liste di costo $\Theta(nk)$.

Ma il tempo richiesto dal testo deve essere più basso, quindi dobbiamo trovare il modo di lavorare più efficientemente. Posto che non sembra possibile ridurre il numero di iterazioni, che dovrà comunque rimanere n, non resta che ragionare sulla ricerca del minimo tra k elementi.

Richiamiamo alla memoria il fatto che una struttura dati in grado di restituire efficientemente il valore minimo, tra quelli che contiene, è il min-heap. Allora, al generico passo i, i valori delle k teste delle k liste ordinate sono memorizzati in un min-heap, costruito in tempo $\Theta(k)$, da cui si può estrarre il minimo in tempo $O(\log k)$; una volta eliminato tale valore dalla struttura ed aggiornato il puntatore relativo alla testa della lista corrispondente, dovremo inserire nel min-heap il nuovo valore minimo, ed anche questo impiega un tempo $O(\log k)$.

Iterando questo procedimento per tutti gli n passi, si giunge ad un costo di $O(k+n \log k)=(n \log k)$, che è quella richiesta.

Esercizio 15.

Riferimento ai capitoli: 4. Equazioni di ricorrenza; 7. Strutture dati fondamentali

Sia dato un albero binario completo con *n* nodi, radicato al nodo puntato dal puntatore *r*. Calcolare il costo computazionale della seguente funzione, in funzione di *n*:

```
link Funzione(link r)
{
    int fogliasx,fogliadx; link r1;
1.  if (!r->fsin) && (!r->fdes)
2.     return r
3.  else {
4.     r1=r;
5.     while (r1->fsin) do
6.        r1=r1->fsin;
7.     fogliasx=r1->dato;
8.     r1=r;
9.     while (r1->fdes) do
10.       r1=r1->fdes;
11.    fogliadx=r1->dato;
12.    if (fogliasx<fogliadx) return Funzione(r->fsin)
13.    else return Funzione(r->fdes);
14. }
}
```

Soluzione.

La funzione è ricorsiva e pertanto dobbiamo trovare un'equazione di ricorrenza in funzione di *n*; il costo si ottiene sommando i contributi delle varie istruzioni:

- le linee 1 e 2 rappresentano il caso base, che costa $\Theta(1)$;
- le linee 4, 7, 8 e 11 costano $\Theta(1)$;
- i cicli while delle linee 5 e 9 vengono ripetuti un certo numero di volte indefinto tra 1 e l'altezza dell'albero, infatti il primo ciclo si interrompe quando il nodo non ha figlio sinistro ed il secondo ciclo quando il nodo non ha figlio destro;
- le linee 12 e 13 sono chiamate ricorsive e quindi il loro costo si dovrà scrivere come T(…) con un opportuno valore tra le parentesi, ma quale?

Se indichiamo con k ed $n - 1 - k$ il numero dei nodi dei sottoalberi radicati ai figli sinistro e destro della radice, otteniamo:

$$T(n)= \Theta(1) + v_1\,\Theta(1) + v_2\,\Theta(1) + max(T(k), T(n-k))$$

dove v_1 e v_2 sono il numero di volte in cui vengono ripetuti i cicli delle linee 5 e 9.

Arrivati qui, non sappiamo come procedere, se non sostituendo a v_1 e v_2 l'altezza dell'albero, che ne è una limitazione superiore. L'altezza ha un valore indefinito tra *log n* ed *n*, e questo significa che dobbiamo anche qui sostituire ad *h* la sua limitazione superiore, cioè *n*. Viste tutte

queste approssimazioni che siamo costretti a fare, dovrebbe venire il dubbio di aver tralasciato qualcosa. Rileggendo con attenzione il testo, si osservi che l'albero in input è binario e ***completo***. Questa ipotesi, che abbiamo tralasciato finora, risolove numerosi problemi, infatti:

- tutti i nodi privi di figlio sinistro (ed anche di figlio destro) sono ad altezza *log n*, pertanto i due cicli alle linee 5 e 9 vengono ripetuti esattamente *log n* volte ciascuno;
- diventa facile calcolare max*(T(k), T(n – k))*: tanto il sottoalbero sinistro che quello destro contengono esattamente *(n –1)/2* nodi ciascuno.

L'equazione di ricorrenza diventa allora:

$T(n)= \Theta(1) + \Theta(\log n)+ T(n/2) = \Theta(\log n) + T(n/2)$

$T(1)= \Theta(1)$

Risolvendo per iterazione si ha:

$$\mathrm{T(n)} = \Theta(\log n)+ T(n/2)=$$

$$= \Theta(\log n) + (\Theta(\log\tfrac{n}{2})+ T(\tfrac{n}{2^2}))=$$

$$= \dots =$$

$$= \sum_{i=0}^{k-1} \Theta\left(\log\frac{n}{2^i}\right) + T\left(\frac{n}{2^i}\right)$$

*S*i procede fino a quando $n/2^k = 1$ e cioè fino a quando $k = \log n$, ottenendo:

$$T(n) = \sum_{i=0}^{\log n-1} \Theta\left(\log\frac{n}{2^i}\right) + T(1) =$$

$$= \Theta\left(\sum_{i=0}^{\log n-1} \left(\log\frac{n}{2^i}\right)\right) + T(1) =$$

$$= \Theta\left(\sum_{i=0}^{\log n-1} (\log n - \log 2^i)\right) + T(1) =$$

$$= \Theta\left(\sum_{i=0}^{\log n-1} (\log n - i)\right) + T(1) =$$

$$= \Theta\left(\log n \sum_{i=0}^{\log n-1} 1 - \sum_{i=0}^{\log n-1} i\right) + T(1) =$$

$$= \Theta\left(\log^2 n + \frac{\log n \log(n-1)}{2}\right) + T(1) =$$

$$= \Theta\left(\log^2 n - \frac{\log^2 n}{2} + \frac{\log n}{2}\right) + T(1) = \Theta(\log^2 n)$$

Esercizio 16.

Riferimento ai capitoli: 7. Strutture dati fondamentali

L'ordine delle operazioni effettuate da una coda può essere simulato grazie all'uso di due pile.

Immaginando di avere un esecutore che può solo fare dei `push(x, i)`, `pop(i)`, `test_di_lista_vuota(i)` su due pile (*i* può assumere i valori 1 e 2 ed indica su quale pila le operazioni sono eseguite):

- si descrivano a parole le operazioni di `dequeue` ed `enqueue` con le due pile;
- si scriva lo pseudocodice delle due funzioni,
- si valuti il costo computazionale delle due funzioni.

Soluzione.

Preliminarmente, chiariamo cosa viene richiesto dall'esercizio. Abbiamo due strutture dati di tipo pila (presumibilmente una –ad esempio la prima- usata per memorizzare i dati ed una –la seconda- di appoggio) con le quali dobbiamo simulare una coda. Questo significa, in particolare, che se vogliamo eseguire l'operazione `dequeue` dovremo estrarre l'elemento che è nella struttura da più tempo, mentre le pile ci offrono la possibilità di estrarre quello che è stato inserito per ultimo; analogamente, se vogliamo eseguire l'operazione `enqueue`, dovremo inserire l'elemento immediatamente dopo quello che è stato inserito al passo precedente.

Abbiamo in realtà più di un modo per procedere. Consideriamo quello più semplice, in cui cioè le due funzioni `enqueue` (per la coda virtuale) e `push` (per la pila) coincidono. Ciò è perfettamente ragionevole, visto che in entrambe le strutture dati gli elementi da inserire vengono accodati nella prima posizione libera.

Perciò, possiamo già scrivere:

```
Enqueue (elemento u):
Push(u, 1);
```

Il costo computazionale di questa operazione è, ovviamente, *O(1)*.

Le cose si complicano leggermente per l'operazione di dequeue, infatti per estrarre dalla pila l'elemento che vi giace da più tempo usando solo le operazioni su di essa permesse (`push` e `pop`) dobbiamo estrarre uno ad uno tutti gli elementi, appoggiandoli temporaneamente nella seconda pila, così da ribaltarne l'ordine; a questo punto possiamo estrarre dalla seconda pila l'ultimo elemento (corrispondente a quello che è stato inserito nella prima pila per primo) e poi ripristinare la situazione.

Seguendo questa filosofia, la funzione in pseudocodice può essere la seguente:

```
elemento Dequeue:
while (test_di_lista_vuota(1)=FALSE)
 push(pop(1), 2);
int aux ← pop(2);
while (test_di_lista_vuota(2)=FALSE)
 push(pop(2), 1);
return aux;
```

Tramite la funzione precedente, svuotiamo la prima pila un elemento per volta e la riversiamo sulla seconda, per poi –tolto l'elemento da estrarre- svuotare la seconda riversandola di nuovo nella prima; il risultato ottenuto è che la prima pila conterrà gli stessi elementi di prima nello stesso ordine, salvo il primo che sarà stato estratto. Detto *n* il numero di elementi che si trovano al momento nella pila, il costo computazionale di questa funzione è ovviamente $\Theta(1)n+ \Theta(1)+ \Theta(1)(n-1)+ \Theta(1)= \Theta(n)$.

Possiamo osservare che il numero di operazioni eseguite potrebbe essere diminuito nel caso in cui eseguiamo una sequenza di operazioni `dequeue`; infatti, supponiamo di dover estrarre due elementi uno dopo l'altro: per la prima estrazione dovremo riversare la prima pila nella seconda (*n* operazioni `push` degli elementi restituiti da altrettante operazioni `pop`) e, una volta estratto l'elemento, riversare nuovamente la seconda nella prima (*n-1* operazioni `push` degli elementi restituiti da altrettante operazioni `pop`), per la seconda estrazione dovremo fare tutto da capo, quindi riversare la prima pila nella seconda (*n-1* operazioni `push` degli elementi restituiti da altrettante operazioni `pop`) e, una volta estratto l'elemento, riversare nuovamente la seconda nella prima (*n-2* operazioni `push` dell'elemento restituito da altrettante operazioni `pop`). Se però potessimo essere in grado di capire che abbiamo due operazioni di tipo `dequeue` una dopo l'altra, potremmo fare a meno di riversare la seconda pila nella prima, e quindi anche di riversare dopo, ancora una volta, la prima nella seconda, dimezzando di fatto il numero di operazioni. Si può "reagire" differentemente a seconda della sequenza di richieste di `enqueue` e `dequeue` operando sulla funzione che si occupa del menu delle scelte. Per brevità, omettiamo dettagli ulteriori su questo punto.

Vogliamo infine sottolineare come la soluzione proposta non sia l'unica: se decidiamo di non riportare indietro i dati dalla seconda alla prima pila, possiamo fare in modo che le due funzioni `enqueue` e `dequeue` procedano in modo differente a seconda che i dati si trovino nella prima o nella seconda pila.

Più precisamente, la funzione `enqueue` è identica alla `push` (come prima) se i dati sono nella pila 1 oppure se entrambe le pile sono vuote, mentre se i dati sono nella pila 2 allora essi

saranno nell'ordine inverso, quindi dobbiamo prima ribaltarli nella pila 1 e poi aggiungere l'elemento. Analogamente, la funzione `dequeue` è identica alla `pop` se i dati sono nella pila 2, ma vanno prima ribaltati nella pila 2 se essi sono nella pila 1. Ovviamente, in questo modo, entrambe le funzioni avranno un costo computazionale di $O(n)$ ma l'approccio non è peggiore del primo poiché la differenza è che mentre nella prima proposta abbiamo semplificato la `enqueue` lasciando alla dequeue un doppio riversamento (dalla 1 alla 2 e poi indietro dalla 2 alla 1), qui abbiamo suddiviso i due riversamenti tra le due funzioni.

Esercizio 17.

Riferimento ai capitoli: 7. Strutture dati fondamentali; 9. Grafi

Sia dato un albero binario qualunque T memorizzato tramite liste di adiacenza L (quindi come un più generale grafo) ed un suo nodo r che ne sia la radice. Si progetti un algoritmo che, presi in input L ed r, stampi le chiavi di tutte le foglie di T.

Dell'algoritmo proposto:

- Si dia la descrizione a parole;
- Si scriva lo pseudocodice;
- Si calcoli il costo computazionale, dettagliando comunque il costo delle funzioni studiate eventualmente usate;
- Si consideri ora lo stesso albero memorizzato tramite la notazione posizionale.Qual è, nel caso peggiore, la dimensione della struttura dati rispetto alla dimensione dell'input? E' possibile adattare l'algoritmo proposto a questa struttura dati? Perché? Si discuta brevemente come cambiano eventualmente l'algoritmo ed il costo computazionale.

Soluzione.

Come prima idea, si potrebbe pensare di utilizzare un algoritmo di visita per grafi (in ampiezza o in profondità) sull'albero memorizzato in L per riconoscere le foglie: esse sono infatti tutti e soli quei nodi da cui non è possibile proseguire la visita poiché non ci sono nodi da esaminare. Un tale approccio richiede un costo di $\Theta(n+m)$.

Dopo una riflessione più attenta, però, ci si rende conto che si può fare di meglio: le foglie di un albero hanno un solo adiacente, il padre; pertanto la loro lista di adiacenza contiene un solo elemento. E' perciò sufficiente stampare i soli indici la cui lista corrispondente è formata di un solo record. Sembrerebbe che l'informazione sulla radice sia inutile, ma non è così: la radice è il solo nodo, oltre alle foglie, che può avere grado 1, e in tal caso va esclusa dall'elenco delle

foglie. Questa soluzione è certamente più efficiente, non richiedendo di scorrere per intero le liste di adiacenza, e richiede solo un costo di $\Theta(n)$.

Lo pseudocodice è molto semplice:

```
Funzione StampaFoglie(L,r)
for i=1 to n do
  if L[i]->next=NULL AND i≠r stampa i;
```

Supponiamo ora che *T* sia dato tramite la notazione posizionale. Poiché non abbiamo informazioni se *T* sia completo o no, possiamo solo dire che il vettore in cui *T* è memorizzato avrà una lunghezza *k* esponenziale nell'altezza di *T*, che può essere anche dell'ordine di 2^n. Non possiamo nemmeno usare il "trucco" di cercare le foglie nelle ultime *k/2* posizioni, proprio perché *T* non è necessariamente completo o quasi completo. L'unica possibilità in questo caso consiste quindi nel visitare l'albero (tutte le filosofie di visita sono corrette in questo caso) e stampare le foglie mentre si individuano. Nonostante la struttura dati abbia una dimensione $k=O(2^n)$, il costo di una visita di un albero memorizzato tramite notazione posizionale è $\Theta(n)$ in quanto non è necessario scorrere tutti gli elementi nulli del vettore.

Esercizio 18.

Riferimento ai capitoli: 4. Equazioni di ricorrenza; 7. Strutture dati fondamentali

L'albero di Kalkin-Wilf è un albero binario (infinito) che nei suoi nodi contiene ciascun numero razionale positivo esattamente una volta. E' così definito:

1) se un nodo ha chiave *a/b*:

- il suo figlio sinistro ha chiave $a/(a + b)$
- il suo figlio destro ha chiave $(a + b)/b$

2) la radice ha chiave 1/1.

Progettare una funzione ricorsiva che, dato un vettore di *n* elementi (con indici da *1* a *n*) considerato come un albero binario completo o quasi completo gestito con la notazione posizionale, lo riempia con i valori dei corrispondenti nodi dell'albero Kalkin-Wilf.

Soluzione.

Innanzi tutto osserviamo che, mentre nella definizione originale, l'albero di Kalkin-Wilf è infinito e quindi non presenta alcun caso base per la terminazione della costruzione, il nostro output deve essere un albero binario con n nodi.

E' allora facile scrivere una funzione ricorsiva che riempie il vettore dato a partire dalla posizione di indice 1, che corrisponde alla radice dell'albero.

```
Funzione crea_albero(V: vettore;n, ind, num, den: intero)
if (ind > n) return;
V[ind] ← num/den;
crea_albero(V, n, 2n, num, num + den);
crea_albero(V, n, 2n+1, num + den, den);
return
```

Questa funzione viene richiamata la prima volta con i parametri ind=num=den=1.

Calcoliamo, infine, il costo computazione. Possiamo scegliere se utilizzare come parametro il numero di nodi nel sottoalbero o la distanza dalle foglie del nodo di posizione ind. Se utilizziamo la prima opzione, e chiamiamo *h* la distanza dalle foglie, l'equazione di ricorrenza è:

$T(h)=2T(h-1)+ \Theta(1)$

$T(0)= \Theta(1)$

Ed il caso base è rappresentato dalla chiamata sull'albero vuoto (con ind >n). E' immediato verificare che la soluzione di questa equazione di ricorrenza è $\Theta(n)$.

Esercizio 19.

Riferimento ai capitoli: 4. Equazioni di ricorrenza; 7. Strutture dati fondamentali

Dato un albero binario di cui si conosce il puntatore alla radice, trovare il nodo che presenta il massimo valore del modulo della differenza fra il numero dei nodi del proprio sottoalbero sinistro e il numero dei nodi del proprio sottoalbero destro. Nel caso in cui vi siano più nodi con lo stesso valore massimo, restituire quello più vicino alla radice. La funzione deve essere ricorsiva.

Soluzione.

Cominciamo con l'osservare che, affinché ciascun nodo calcoli il modulo della differenza tra numero di nodi nel sottoalbero sinistro e numero di nodi nel sottoalbero destro, è necessario che esso riceva opportune informazioni da entrambi i suoi figli e poi effettui il calcolo. Per questo, la funzione che dobbiamo scrivere dovrà seguire la filosofia della visita in post-ordine.

Per semplicità, utilizziamo due variabili globali, condivise quindi da tutte le chiamate ricorsive:

- nodo_sbil: un puntatore, inizialmente NULL, al nodo individuato dalla funzione
- max_sbil: un intero, inizialmente -1, pari al massimo del modulo della differenza trovata.

```
Funzione conta(t: puntatore all'albero)
if (t = NULL) return 0
nodi_sin ← conta(left[t])
nodi_des ← conta(right[t])
if (modulo(nodi_sin - nodi_des) >= max_sbil)
 max_sbil ← modulo(nodi_sin - nodi_des)
 nodo_sbil ← t
return (nodi_sin + nodi_des +1)
```

La funzione ricorsiva appena scritta, invocata su un certo nodo *x*, riceve ricorsivamente il numero di nodi nei suoi due sottoalberi dai suoi figli, confronta la loro differenza col valore di max_sbil, se necessario aggiorna tale valore e il puntatore nodo_sbil, ed infine calcola il numero di nodi nel suo sottoalbero e lo restituisce al padre.

Il costo computazionale è, come è ovvio, quello della visita di un albero, e l'equazione di ricorrenza relativa allo pseudocodice è:

$T(n)=T(k)+T(n-1-k)+\Theta(1)$

$T(0)=\Theta(1)$

Che si può risolvere con il metodo di sostituzione (v. Dispense di teoria) dando come soluzione $\Theta(n)$.

Esercizio 20.

Riferimento ai capitoli: 8. Dizionari

Siano dati due alberi binari di ricerca: B_1 con n_1 nodi ed altezza h_1, e B_2 con n_2 nodi ed altezza h_2. Assumendo che tutti gli elementi in B_1 siano minori di quelli in B_2, descrivere un algoritmo che fonda gli alberi B_1 e B_2 in un unico albero binario di ricerca B di $n_1 + n_2$ nodi.

Determinare l'altezza dell'albero B e il costo computazionale.

Soluzione.

La prima idea che si può provare a perseguire è quella di inserire nell'albero con il maggior numero di nodi (senza perdere di generalità sia esso B_1) i nodi dell'altro albero uno ad uno. Sapendo che l'operazione di inserimento in un albero binario di ricerca ha un costo dell'ordine dell'altezza dell'albero si ha, nel caso peggiore, che il costo computazionale è:

$$O(h_1) + O(h_1+1) + O(h_1+2) + \ldots + O(h_1 + n_2 - 1) = n_2 O(h_1) + O(1 + 2 + \ldots + n_2 - 1)$$

Poiché nel caso peggiore $O(h_1) = O(n_1)$ ed $O(1 + 2 + \ldots + n_2) = O(n_2^2)$, si ottiene che il costo di questo approccio è $O(n_1 n_2) + O(n_2^2) = O(n_1 n_2)$, essendo per ipotesi $n_1 > n_2$.

Tuttavia, questa soluzione non utilizza l'ipotesi che tutti gli elementi di B_1 siano minori di quelli di B_2. Per tentare di sfruttare questa ipotesi (è buona norma tenere presente che, se un'ipotesi c'è, allora serve a qualcosa!) possiamo pensare di appendere l'intero albero B_1 come figlio sinistro del minimo di B_2. Questo si può sempre fare facilmente perché il minimo di un albero binario di ricerca è, per definizione, il nodo più a sinistra che non possiede figlio sinistro, pertanto è sufficiente settare un puntatore sulla radice di B_2, scendere verso il figlio sinistro finché esso esista e, giunti al nodo che non ha figlio sinistro (che è il minimo), agganciargli a sinistra la radice di B_1. Osserviamo che questo procedimento è corretto perché un albero binario di ricerca ***non*** è necessariamente bilanciato, e quindi poco importa che l'altezza dell'albero risultante possa diventare $O(h_1 + h_2)$. Il costo di questo approccio è dominato dal costo della ricerca del minimo, cioè da $O(h_1)$ ed è, pertanto, da preferirsi al primo metodo proposto.

Esercizio 21.

Riferimento ai capitoli: 9. Grafi

Sia G un grafo con n nodi ed m spigoli. Dire se le seguenti affermazioni sono vere o false e giustificare la propria risposta:

1) tutte le foreste generate da differenti visite in profondità hanno lo stesso numero di alberi;
2) tutte le foreste generate da differenti visite in profondità hanno lo stesso numero di spigoli dell'albero e lo stesso numero di spigoli all'indietro.

Soluzione.

1. Si ricordi che, dato un grafo *G* con *k* componenti connesse, qualsiasi foresta ricoprente (che sia generata da una visita in profondità o no) è formata esattamente da *k* alberi, uno per ciascuna componente connessa. Da questa semplice osservazione si deduce che la risposta al primo quesito è affermativa.

2. Anche la risposta al secondo quesito è affermativa; infatti ogni albero con a nodi ha $a - 1$ spigoli, ed ogni foresta di k alberi con n nodi ha $n - k$ spigoli. Pertanto il numero di spigoli dell'albero è uguale per tutte le foreste (sia quelle generate da visita in profondità che le altre).

Per quanto riguarda gli spigoli all'indietro, si ricordi che una visita in profondità non produce spigoli di attraversamento, e quindi tutti gli spigoli non dell'albero sono all'indietro; ne consegue che tutte le foreste generate da differenti visite in profondità hanno $m - (n - k)$ spigoli all'indietro.

Concludendo, mentre le proprietà descritte dalla prima affermazione e dalla parte della seconda affermazione riguardante gli spigoli dell'albero sono proprietà valide per tutti gli alberi ricoprenti, indipendentemente dal fatto che essi scaturiscano da una visita in profondità o no, la proprietà sugli archi all'indietro relativa alla seconda affermazione si basa pesantemente sulle proprietà degli alberi generati da visita in profondità.

Esercizio 22.

Riferimento ai capitoli: 9. Grafi

Si progetti un algoritmo che, dato un grafo semplice tramite la sua matrice di adiacenza, restituisca 1 se esistono due nodi con lo stesso grado e 0 altrimenti.

Dell'algoritmo proposto:

- Si dia la descrizione a parole, dettagliando eventuali strutture dati ausiliarie utilizzate;
- Si calcoli il costo computazionale;
- Si discuta come cambia il costo computazionale se il grafo in input è dato tramite le altre memorizzazioni viste a lezione.

Soluzione.

E' chiaro che la prima idea che viene in mente è quella di scorrere la struttura dati in input per valutare il grado di ciascun nodo e poi di verificare se ci siano due gradi uguali.

Per quanto riguarda la prima operazione, ricordiamo che sulla matrice di adiacenza, per ogni nodo, valutarne il grado ha un costo computazionale dell'ordine di $\Theta(n)$; pertanto, valutare i gradi di tutti i nodi costerà $\Theta(n^2)$. Non sembra possibile fare meglio di così.

Dobbiamo però ora decidere come fare a confrontare i gradi per stabilire se ve ne siano due uguali. Una possibile soluzione consiste nel memorizzare i gradi in un vettore *Deg* di *n* elementi e poi, per ciascun grado, di cercare nel resto del vettore se vi sia un valore uguale. Questo modo di procedere costa ovviamente $\Theta(n^2)$, che si va ad aggiungere al $\Theta(n^2)$ della prima parte producendo un algoritmo di costo globale $\Theta(n^2)$.

Con l'idea di rispondere poi in modo efficiente all'ultimo quesito, possiamo cercare di migliorare il costo computazionale di almeno questo secondo passo dell'algoritmo. In effetti, è facile vedere che se prima ordiniamo gli elementi di *Deg* (con un algoritmo efficiente), basterà scorrere il vettore ordinato per individuare eventuali occorrenze multiple. Con questo approccio, la seconda parte dell'algoritmo costa $\Theta(n \log n)+ \Theta(n)= \Theta(n \log n)$. Tale costo può essere ulteriormente decrementato, osservando che ogni elemento di *Deg* è un grado, e quindi è un valore compreso tra 0 e $n-1$; possiamo quindi applicare l'algoritmo di Counting sort, dando luogo ad un costo per questa seconda parte pari a $\Theta(n)$.

Analogamente, avremmo potuto ottenere lo stesso costo utilizzando un vettore *NumDeg* come contatore delle molteplicità dei gradi: se per un nodo *v* si trova che esso ha grado $deg(v)=i$, allora si incrementa il valore di *NumDeg[i]*; dopo aver perlustrato tutti i nodi, si hanno due nodi con lo stesso grado se esiste una cella di *NumDeg* con valore >1.

Concludiamo la trattazione di questo primo quesito asserendo che il costo computazionale per risolvere il problema su matrice di adiacenza è $\Theta(n^2)$, indipendentemente dalla soluzione utilizzata per confrontare i gradi.

Non rimane ora che passare in rassegna le altre strutture dati, le quali modificheranno il costo computazionale della prima parte, lasciando invece inalterato il costo della seconda.

Liste di adiacenza. Per calcolare il grado di un nodo *v* è necessario scorrere la *v*-esima lista, con un costo di $\Theta(1+deg(v))$; sommando su tutti i nodi, il costo della prima parte dell'algoritmo è $\Theta(n+m)$ che, sommato al costo della seconda parte ($\Theta(n)$) dà luogo ad un costo totale di $\Theta(n+m)$.

Matrice di incidenza. Per calcolare il grado di un nodo *v* è necessario scorrere la riga *v*-esima, con un costo di $\Theta(m)$; sommando su tutti i nodi, il costo della prima parte dell'algoritmo è $\Theta(n\ m)$ che, sommato al costo della seconda parte dà luogo ad un costo totale di $\Theta(n\ m)$.

Lista di archi. Per calcolare il grado di un nodo *v* è necessario scorrere l'intero vettore, con un costo per tutti i nodi di $\Theta(n^2)$. Tuttavia, se usiamo per la seconda parte dell'algoritmo il vettore

Deg, è sufficiente scorrere il vettore che memorizza la lista di archi una volta sola, ed ogni volta che si incontra l'occorrenza di un arco $e=(u,v)$, si incrementa il valore di *Deg[u]* e di *Deg[v].* In totale, il costo dell'algoritmo sarà di $\Theta(n)$.

Vogliamo concludere questa trattazione con una semplice considerazione: il grafo in input è semplice quindi, come abbiamo già avuto modo di osservare, i gradi degli n nodi sono valori tra 0 ed n-1. Se i gradi sono tutti diversi, allora tutti i valori da 0 ad n-1 devono essere usati. Ma se esiste un nodo con grado n-1, allora esso è adiacente a tutti gli altri nodi, e quindi non potrà esistere un nodo con grado 0. Segue che non è possibile che i nodi abbiano tutti gradi diversi.

Un semplice algoritmo che risolve il problema dato, indipendentemente dalla struttura dati usata per memorizzare il grafo è allora il seguente:

```
int VerificaSeGradiUguali()

return 1
```

che ha, evidentemente, costo costante.

Simboli

O

Ω

Θ

$\boldsymbol{\varepsilon}$

$\supseteq \subseteq \subset \supset \Rightarrow$

$\in$

$\leftarrow$