Soluzioni scelte

per affiancare
Introduzione agli Algoritmi e Strutture Dati
quarta edizione
di
Thomas H. Cormen
Charles E. Leiserson
Ronald L. Rivest
Clifford Stein

Thomas H. Cormen
Department of
Computer Science
Dartmouth College

INTRODUZIONE AGLI ALGORITMI E STRUTTURE DATI 4/ED
di Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein

Soluzioni scelte per il Capitolo 2: Per incominciare

Soluzione dell'Esercizio 2.2-2

SELECTION-SORT(A, n)
    for $i = 1$ **to** $n - 1$
        $smallest = i$
        for $j = i + 1$ **to** n
            if $A[j] < A[smallest]$
                $smallest = j$
        scambia $A[i]$ con $A[smallest]$

L'algoritmo conserva questa invariante di ciclo: all'inizio di ogni iterazione del ciclo **for** esterno, il sottoarray $A[1 : i - 1]$ è ordinato e contiene gli $i - 1$ elementi più piccoli dell'array $A[1 : n]$. Dopo $n - 1$ iterazioni, il sottoarray $A[1 : n - 1]$ contiene gli $n - 1$ elementi più piccoli in ordine, e quindi l'elemento $A[n]$ deve essere l'elemento più grande.
Il tempo di esecuzione dell'algoritmo è $O(n^2)$ in ogni caso.

Soluzione dell'Esercizio 2.2-4
Basta modificare l'algoritmo in modo che prima verifichi se l'array di input è già ordinato: per far questo serve un tempo $O(n)$ su un array di n elementi. Se l'array è già ordinato, l'algoritmo termina. Altrimenti, ordina l'array come prevede l'algoritmo non modificato. Ecco perché il tempo di esecuzione nel caso migliore non è generalmente una buona misura dell'efficienza di un algoritmo.

Soluzione dell'Esercizio 2.3-6

La procedura BINARY-SEARCH prende un array ordinato A, un valore x e un intervallo $[low : high]$ dell'array in cui cercare il valore x. La procedura confronta x con l'elemento dell'array nel punto medio dell'intervallo e per eliminare metà dell'intervallo da ulteriori considerazioni. Presentiamo sia una versione iterativa sia una ricorsiva, ognuna delle quali restituisce un indice i tale che $A[i] = x$, oppure NIL se nessun elemento del sottoarray $A[low : high]$ è uguale a x. La chiamata iniziale in entrambi i casi deve passare i parametri A, x, 1, n.

```
ITERATIVE-BINARY-SEARCH(A, x, low, high)
    while low ≤ high
        mid = ⌊(low + high)/2⌋
        if x == A[mid]
            return mid
        elseif x > A[mid]
            low = mid + 1
        else high = mid − 1
    return NIL
```

```
RECURSIVE-BINARY-SEARCH(A, x, low, high)
    if low > high
        return NIL
    mid = ⌊(low + high)/2⌋
    if x == A[mid]
        return mid
    elseif x > A[mid]
        return RECURSIVE-BINARY-SEARCH(A, x, mid + 1, high)
    return RECURSIVE-BINARY-SEARCH(A, x, low, mid − 1)
```

Entrambe le procedure terminano la ricerca senza successo quando l'intervallo è vuoto (cioè, se $low > high$) e la terminano con successo se il x è stato trovato. In base al confronto di x con l'elemento centrale dell'intervallo in cui lo si sta cercando, la ricerca prosegue su un intervallo di metà elementi. La ricorrenza per queste procedure è quindi $T(n) = T(n/2) + \Theta(1)$, la cui soluzione è $T(n) = \Theta(\lg n)$.

Soluzione del Problema 2-4

a. Le inversioni sono $(1,5)$, $(2,5)$, $(3,4)$, $(3,5)$, $(4,5)$ (tenete a mente che le inversioni sono individuate tramite gli indici degli elementi e non con i loro valori).

b. L'array con più inversioni i cui elementi sono quelli dell'insieme $\{1,2,\ldots,n\}$ è $\langle n, n-1, n-2, \ldots, 2, 1\rangle$. Per $1 \le i < j \le n$ c'è sempre l'inversione (i,j). Il numero totale di inversioni è dunque $\binom{n}{2} = n(n-1)/2$.

c. Supponiamo che l'array A inizi con un'inversione (k,i). Nel momento in cui il ciclo **for** esterno delle righe 1-8 pone $key = A[i]$, il valore che inizialmente era $A[k]$ si trova ancora da qualche parte a sinistra di $A[i]$. Si trova cioè in $A[j]$, con $1 \le j < i$, e quindi l'inversione è diventata (j,i). Ogni iterazione del ciclo **while** delle righe 5-7 sposta $A[j]$ di una posizione a destra. La riga 8 prima o poi farà in modo che il valore key finisca a sinistra di questo elemento, eliminando così l'inversione. Poiché la riga 5 sposta solo gli elementi maggiori di key, sposta solo gli elementi che corrispondono alle inversioni. In altre parole, ogni iterazione del ciclo **while** delle righe 5-7 corrisponde all'eliminazione di un'inversione.

d. Seguiamo il suggerimento e modifichiamo il merge sort in modo da contare il numero di inversioni in tempo $\Theta(n \lg n)$.

Per cominciare, diciamo che una situazione all'interno dell'esecuzione del merge sort in cui la procedura MERGE, dopo aver copiato $A[p:q]$ in L e $A[q+1:r]$ in R, fa si che ci siano un valore x in L e y in R tali che $x > y$ è una ***merge-inversione***. Consideriamo un'inversione (i,j), e siano $x = A[i]$ e $y = A[j]$ tali che $i < j$ e $x > y$. Sosteniamo che se dovessimo eseguire il merge sort, allora ci sarebbe esattamente una merge-inversione che coinvolge x e y. Per capire perché, osserviamo che l'unico modo in cui gli elementi dell'array cambiano la loro posizione è all'interno della procedura MERGE. Inoltre, poiché MERGE mantiene gli elementi all'interno di L nello stesso ordine relativo tra loro e analogamente per R, l'unico modo in cui due elementi possono cambiare il loro ordine relativo è che il maggiore stia in L e il minore in R. Quindi, c'è almeno una merge-inversione che coinvolge x e y. Per provare che c'è esattamente una merge-inversione di questo tipo, osserviamo che

dopo ogni chiamata di MERGE che coinvolge sia x sia y, questi si troveranno nello stesso sottoarray ordinato e quindi staranno entrambi in L o entrambi in R in ogni chiamata successiva. Abbiamo quindi dimostrato l'affermazione.

Abbiamo dimostrato che per ogni inversione si ha una merge-inversione. In realtà, la corrispondenza tra inversioni e merge-inversioni è biunivoca. Supponiamo di avere una merge-inversione che coinvolge i valori x e y, dove x era originariamente $A[i]$ e y era originariamente $A[j]$. Poiché abbiamo una merge-inversione, si ha $x > y$. E poiché x è in L e y è in R, x deve trovarsi all'interno di un sottoarray che precede il sottoarray che contiene y. Pertanto x era inizialmente in una posizione i che precede la posizione originale j di y, e quindi (i, j) è un'inversione.

Avendo provato che inversioni e merge-inversioni sono in corrispondenza biunivoca, ci basta contare le merge-inversioni.

Consideriamo una merge-inversione che coinvolge y in R. Sia z il valore più piccolo in L che è maggiore di y. A un certo punto del processo di fusione, z e y saranno i valori "scoperti" in L e R, cioè avremo $z = L[i]$ e $y = R[j]$ nella riga 13 di MERGE. In quel momento, ci saranno delle merge-inversioni coinvolgeranno y e $L[i]$, $L[i + 1]$, $L[i + 2]$,..., $L[n_L - 1]$, e queste $n_L - i$ merge-inversioni saranno le uniche a coinvolgere y. Pertanto, dobbiamo individuare la prima volta che z e y vengono scoperti durante la procedura MERGE e aggiungere il valore di $n_L - i$ in quel dato momento al contatore delle merge-inversioni.

Il seguente pseudocodice, modellato sul merge sort, funziona come abbiamo appena descritto. Inoltre ordina l'array A.

MERGE-INVERSION(A, p, q, r)

$n_L = q - p + 1$
$n_R = r - q$
crea due nuovi array $L[0 : n_L + 1]$ e $R[0 : n_R - 1]$
for $i = 0$ **to** $n_L - 1$
    $L[i] = A[p + i - 1]$
for $j = 0$ **to** $n_R - 1$
    $R[j] = A[q + j]$
$i = 0$
$j = 1$
$k = p$
$inversions = 0$
while $i < n_L$ e $j < n_R$
    if $L[i] \leq R[j]$
        $inversions = inversions + n_L - i$
        $A[k] = L[i]$
        $i = i + 1$
    else $A[k] = R[j]$
        $j = j + 1$
    $k = k + 1$
while $i < n_L$
    $A[k] = L[i]$
    $i = i + 1$
    $k = k + 1$
while $j < n_R$
    $A[k] = R[j]$
    $j = j + 1$
    $k = k + 1$
return $inversions$

COUNT-INVERSIONS(A, p, r)

$inversions = 0$
if $p < r$
    $q = \lfloor (p + r)/2 \rfloor$
    $inversions = inversions +$ COUNT-INVERSIONS(A, p, q)
    $inversions = inversions +$ COUNT-INVERSIONS$(A, q + 1, r)$
    $inversions = inversions +$ MERGE-INVERSIONS(A, p, q, r)
return $inversions$

La chiamata iniziale è COUNT-INVERSIONS$(A, 1, n)$.

In MERGE-INVERSION, ogni volta che $R[j]$ viene scoperto e un valore maggiore di $R[j]$ viene scoperto nell'array L, aumentiamo $inversions$ del numero di elementi restanti in L. Quindi, poiché al passo successivo viene scoperto $R[j+1]$, $R[j]$ non potrà più esserlo.

Poiché abbiamo aggiunto solo una quantità costante di lavoro a ogni chiamata e a ogni iterazione dell'ultimo ciclo **for** della procedura di fusione, il tempo di esecuzione totale dello pseudocodice qua sopra è lo stesso del merge sort, ovvero $\Theta(n \lg n)$.

Soluzioni scelte per il Capitolo 3: Descrivere i tempi di esecuzione

Soluzione dell'Esercizio 3.2-2
Poiché la notazione O fornisce solo un limite superiore, e non un limite stretto, l'affermazione dice che il tempo di esecuzione dell'algoritmo A è almeno una funzione il cui tasso di crescita è al massimo n^2.

Soluzione dell'Esercizio 3.2-3
$2^{n+1} = O(2^n)$, ma $2^{2n} \neq O(2^n)$.
Per provare che $2^{n+1} = O(2^n)$, dobbiamo trovare delle costanti positive c e n_0 tali che per ogni $n \geq n_0$ si abbia $0 \leq 2^{n+1} \leq c \cdot 2^n$.
Poiché per ogni n si ha $2^{n+1} = 2 \cdot 2^n$, basta considerare $c = 2$ e $n_0 = 1$.
Per mostrare che $2^{2n} \neq O(2^n)$, assumiamo che esistano delle costanti positive c e n_0 tali che per ogni $n \geq n_0$ si abbia $0 \leq 2^{2n} \leq c \cdot 2^n$.
Allora avremmo $2^{2n} = 2^n \cdot 2^n \leq c \cdot 2^n$, da cui $2^n \leq c$; ma nessuna costante è maggiore di 2^n per ogni n, e quindi la nostra assunzione è assurda.

Soluzione dell'Esercizio 3.3-5
$\lceil \lg n \rceil !$ non è polinomialmente limitata, ma $\lceil \lg \lg n \rceil !$ sì.
Proviamo che affermare che una funzione $f(n)$ è polinomialmente limitata equivale a dimostrare che $\lg f(n) = O(\lg n)$, come segue dalle seguenti due osservazioni.

- Se $f(n)$ è polinomialmente limitata, allora esistono delle costanti positive c, k e n_0 tali che per ogni $n \geq n_0$ si ha $0 \leq f(n) \leq cn^k$. Senza perdita di generalità, supponiamo che sia $c \geq 1$, poiché se fosse $c < 1$ allora $f(n) \leq cn^k$ implicherebbe $f(n) \leq n^k$. Assumiamo anche che sia $n_0 \geq 2$, di modo che

$n \geq n_0$ implica $\lg c \leq (\lg c)(\lg n)$. Quindi, abbiamo

$$\begin{aligned} \lg f(n) &\leq \lg c + k \lg n \\ &\leq (\lg c + k) \lg n \ , \end{aligned}$$

che, poiché c e k sono costanti, significa $\lg f(n) = O(\lg n)$.

- Supponiamo ora che sia $\lg f(n) = O(\lg n)$. Allora esistono delle costanti positive c e n_0 tali che $0 \leq \lg f(n) \leq c \lg n$ per ogni $n \geq n_0$. Allora, abbiamo

$$0 \leq f(n) = 2^{\lg f(n)} \leq 2^{clgn} = (2^{\lg n})^c = n^c$$

per ogni $n \geq n_0$, e dunque $f(n)$ è polinomialmente limitata.

In quel che segue, faremo uso dei seguenti fatti:

1. $\lg(n!) = \Theta(n \lg n)$ (per l'equazione (3.28))
2. $\lceil \lg n \rceil = \Theta(\lg n)$, poiché
   - $\lceil \lg n \rceil \geq \lg n$,
   - $\lceil \lg n \rceil < \lg n + 1 \leq 2 \lg n$, per ogni $n \geq 2$.

Abbiamo

$$\begin{aligned} \lg(\lceil \lg n \rceil !) &= \Theta(\lceil \lg n \rceil \lg \lceil \lg n \rceil) \\ &= \Theta((\lg n)(\lg \lg n)) \\ &= \omega(\lg n) \ . \end{aligned}$$

Di conseguenza, $\lg(\lceil \lg n \rceil !)$ non è $O(\lg n)$, e dunque $\lceil \lg n \rceil !$ non è polinomialmente limitata.
Abbiamo anche

$$\begin{aligned} \lg(\lceil \lg \lg n \rceil !) &= \Theta(\lceil \lg \lg n \rceil \lg \lceil \lg \lg n \rceil) \\ &= \Theta((\lg \lg n)(\lg \lg \lg n)) \\ &= o((\lg \lg n)^2) \\ &= o(\lg^2(lgn)) \\ &= o(\lg n) \ . \end{aligned}$$

L'ultima uguaglianza segue dal fatto che ogni funzione polilogaritmica cresce più lentamente di ogni funzione polinomiale, ovvero che per ogni costante $a, b > 0$ si ha $\lg^b n = o(n^a)$. Sostituendo $\lg n$ a n, 2 a b e 1 ad a, si ottiene $\lg^2(lgn) = o(\lg n)$. Di conseguenza, $\lg(\lceil \lg \lg n \rceil !) = O(\lg n)$ e dunque $\lceil \lg \lg n \rceil !$ è polinomialmente limitata.

12

Soluzioni scelte per il Capitolo 4: Divide et impera

Soluzione dell'Esercizio 4.2-3

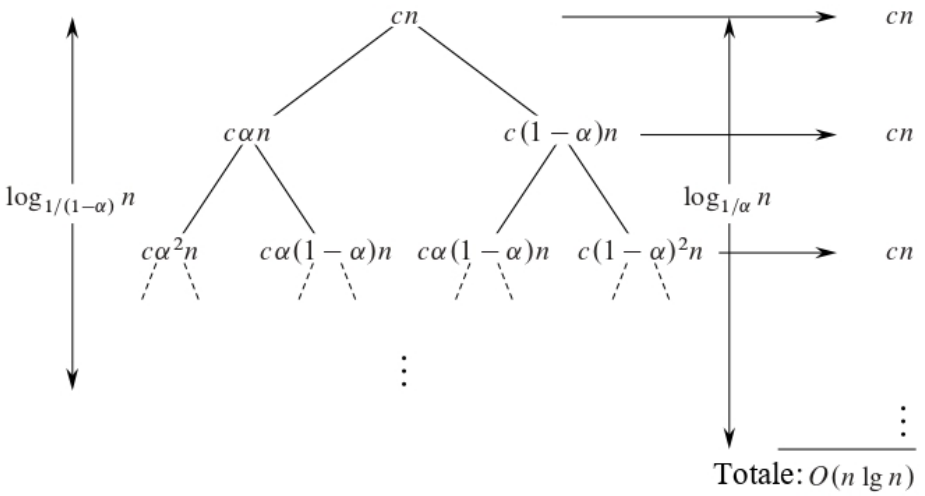
Se fosse possibile calcolare il prodotto tra matrici 3×3 effettuando k moltiplicazioni, allora potremmo calcolare il prodotto tra matrici $n \times n$ moltiplicando ricorsivamente delle matrici $n/3 \times n/3$ in tempo $T(n) = kT(n/3) + \Theta(n^2)$.
Per risolvere questa ricorrenza con il metodo principale, consideriamo il rapporto tra $n^{\log_3 k}$ e n^2.

- Se $\log_3 k = 2$, allora siamo nel caso 2 e otteniamo $T(n) = \Theta(n^2 \lg n)$. In questo caso $k = 9$ e $T(n) = o(n^{\lg 7})$.
- Se $\log_3 k < 2$, allora siamo nel caso 3 e otteniamo $T(n) = \Theta(n^2)$. In questo caso $k < 9$ e $T(n) = o(n^{\lg 7})$.
- Se $\log_3 k > 2$, allora siamo nel caso 1 e otteniamo $T(n) = \Theta(n^{\log_3 k})$. In questo caso $k > 9$. $T(n) = o(n^{\lg 7})$ quando $\log_3 k < \lg 7$, cioè per $k < 3^{\lg 7} \approx 21,85$, e visto che k intero, il valore massimo è 21.

Quindi $k = 21$ e il tempo di esecuzione è $\Theta(n^{\log_3 k}) = \Theta(n^{\log 321}) = O(n^{2,80})$, visto che $\log_3 21 \approx 2,77$.

Soluzione dell'Esercizio 4.4-4

$T(n) = T(\alpha n) + T((1 - \alpha)n) + cn$.
Nel libro abbiamo visto la soluzione della ricorrenza $T(n) = T(n/3) + T(2n/3)$; questa può essere risolta in modo analogo.
Senza perdità di generalità, sia $\alpha \geq 1 - \alpha$, di modo che $0 < 1 - \alpha \leq 1/2$ e $1/2 \leq \alpha < 1$.
Allora l'albero di ricorsione è pieno per $\log_{1/(1-\alpha)} n$ livelli, ciscuno dei quali contribuisce per cn, e quindi possiamo formulare l'ipotesi $T(n) = \Omega(n \log_{1/(1-\alpha)} n) = \Omega(n \lg n)$.



Mostriamo ora che $T(n) = \Theta(n \lg n)$ con il metodo di sostituzione. Per provare il limite superiore, dobbiamo mostrare che $T(n) \leq dn \lg n$ per un'opportuna costante d:

$$\begin{aligned} T(n) &= T(\alpha n) + T((1-\alpha)n) + cn \\ &\leq d\alpha n \lg(\alpha n) + d(1-\alpha)n \lg((1-\alpha)n) + cn \\ &= d\alpha n \lg \alpha + d\alpha n \lg n + d(1-\alpha)n \lg(1-\alpha) + d(1-\alpha)n \lg n + cn \\ &= dn \lg n + dn(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) + cn \\ &\leq dn \lg n \,, \end{aligned}$$

purché sia $dn(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) + cn \leq 0$. Questa condizione equivale a

$$d(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) \leq -c \,.$$

Poiché $1/2 \leq \alpha < 1$ e $0 < 1 - \alpha \leq 1/2$, abbiamo $\lg \alpha < 0$ e $\lg(1-\alpha) < 0$. Quindi $\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha) < 0$, da cui, dividendo i due membri della precedente disequazione per questo fattore e facendo attenzione al segno della disuguaglianza, abbiamo

$$d \geq \frac{-c}{\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)}$$

ovvero

$$d \geq \frac{c}{-\alpha \lg \alpha - (1-\alpha)\lg(1-\alpha)} \,.$$

La frazione a secondo membro è una costante positiva, quindi basta scegliere un valore di d che sia maggiore o uguale di questa frazione.
Per dimostrare il limite inferiore, dobbiamo mostrare che $T(n) \geq dn \lg n$ per un'opportuna costante $d > 0$. Possiamo usare gli stessi calcoli visti sopra, scambiando il verso delle disuguaglianze per ottenere il vincolo

$$0 \leq d \leq \frac{c}{-\alpha \lg \alpha - (1-\alpha)\lg(1-\alpha)} .$$

In conclusione, $T(n) = \Theta(n \lg n)$.

16

Soluzioni scelte per il Capitolo 5: Analisi probabilistica e algoritmi randomizzati

Soluzione dell'Esercizio 5.2-1

Poiché la procedura HIRE-ASSISTANT assume sempre il primo candidato, assume esattamente una volta se e solo se nessun candidato oltre al primo viene assunto. Questo evento si verifica quando il primo candidato è il miglior tra tutti gli n candidati, il che avviene con probabilità $1/n$.

La procedura HIRE-ASSISTANT compie n assunzioni solo se ogni candidato è migliore di tutti quelli che sono stati intervistati (e assunti) precedentemente. Questo evento si verifica proprio quando l'elenco dei ranghi fornito all'algoritmo è $\langle 1, 2, \ldots, n\rangle$, il che si verifica con probabilità $1/n!$.

Soluzione dell'Esercizio 5.2-5

Il problema dei cappelli equivale a determinare il numero atteso di punti fissi in una permutazione casuale (un ***punto fisso*** di una permutazione π è un valore i tale che $\pi(i) = i$). Potremmo enumerare tutte le $n!$ permutazioni, contare il numero totale di punti fissi e dividere per $n!$ per determinare il numero medio di punti fissi per permutazione. Si tratterebbe di un processo assai lungo e complicato, e alla fine la risposta sarebbe 1. Tuttavia, possiamo utilizzare le variabili casuali indicatrici per ottenere la stessa risposta in modo molto più semplice.

Definiamo una variabile casuale X uguale al numero di clienti che recuperano il proprio cappello, dunqoe vogliamo calcolare $\mathrm{E}[X]$.

Per $i = 1, 2, \ldots, n$, definiamo la variabile casuale indicatrice

$$X_i = \mathrm{I}\{\text{il cliente } i\text{-esimo riceve indietro il suo cappello}\} .$$

Allora $X = X_1 + X_2 + \cdots + X_n$.

Poiché l'ordine dei cappelli è casuale, ogni cliente ha probabilità $1/n$ di riavere il proprio cappello. In altre parole, $\Pr\{X_i = 1\} = 1/n$, che, per il Lemma 5.1, implica che $\mathrm{E}[X_i] = 1/n$.
Dunque

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] \qquad \text{per la linearità del valore atteso} \\
&= \sum_{i=1}^{n} 1/n \\
&= 1 \,,
\end{aligned}$$

e quindi ci aspettiamo che esattamente un cliente riceva indietro il suo cappello.
Notate che questa è una situazione in cui le variabili casuali indicatrici non sono indipendenti. Ad esempio, se $n = 2$ e $X_1 = 1$, allora anche X_2 deve essere uguale a 1. Al contrario, se $n = 2$ e $X_1 = 0$, allora anche X_2 deve essere uguale a 0. Nonostante la dipendenza, $\Pr\{X_i = 1\} = 1/n$ per ogni i e vale la linearità del valore atteso. Pertanto, possiamo utilizzare la tecnica delle variabili casuali indicatrici anche in presenza di dipendenza, come in questo caso.

Soluzione dell'Esercizio 5.2-6
Sia X_{ij} la variabile casuale indicatrice che indica se la coppia $(A[i], A[j])$ con $i < j$ è u'inversione, cioè se $A[i] > A[j]$. Più precisamente, definiamo $X_{ij} = \mathrm{I}\{A[i] > A[j]\}$ per $1 \le i < j \le n$. Abbiamo $\Pr\{X_{ij} = 1\} = 1/2$,poiché dati due numeri diversi scelti a caso, la probabilità che il primo sia maggiore del secondo è $1/2$. Per il Lemma 5.1, $\mathrm{E}[X_{ij}] = 1/2$.
Sia X la variabile casuale che conta il numero totale di inversioni, ovvero

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \,.$$

Vogliamo trovare il numero atteso di inversioni, quindi calcoliamo il valore atteso dei due membri dell'equazione sopra, da cuio

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \,.$$

Sfruttando la linearità del valore atteso otteniamo

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \mathrm{E}[X_i] \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 1/2 \\
&= \binom{n}{2}\frac{1}{2} \\
&= \frac{n(n-1)}{2}\cdot\frac{1}{2} \\
&= \frac{n(n-1)}{4}\,.
\end{aligned}$$

Quindi il numero atteso di inversioni è $n(n-1)/4$.

Soluzione dell'Esercizio 5.3-2

Oltre alla permutazione identità, ci sono altre permutazioni che la procedura PERMUTE-WITHOUT-IDENTITY non riesce a produrre. Ad esempio, consideriamo il suo funzionamento quando $n = 3$, quando dovrebbe essere in grado di produrre le $n! - 1 = 5$ permutazioni che non sono l'identità. Il ciclo **for** itera per $i = 1$ e $i = 2$. Quando $i = 1$, la chiamata a RANDOM restituisce uno dei due valori possibili (2 o 3), mentre quando $i = 2$, la chiamata a RANDOM restituisce un solo valore (3). Pertanto, la procedura PERMUTE-WITHOUT-IDENTITY può produrre solo $2 \cdot 1 = 2$ permutazioni, invece delle 5 richieste.

Soluzione dell'Esercizio 5.3-4

La procedura PERMUTE-BY-CYCLE sceglie il valore dell'intero offset in modo casuale nell'intervallo da 1 a n, e poi esegue una rotazione ciclica dell'array. Ovvero, $B[((i + \mathit{offset} - 1) \mod n) + 1] = A[i]$ per $i = 1, 2, \ldots, n$ (la sottrazione e l'aggiunta di 1 nel calcolo dell'indice è dovuta all'indicizzazione con origine 1; se avessimo usato l'indicizzazione con origine a 0 avremmo semplicemente $B[(i + \mathit{offset})\ \mathit{mod} n] = A[i]$ per $i = 1, 2, \ldots, n$).
Quindi, una volta determinato il valore di offset, lo è anche l'intera permutazione. Poiché ogni valore di offset viene assunto con probabilità $1/n$, ogni elemento $A[i]$ ha probabilità di finire nella posizione $B[j]$ con una probabilità di $1/n$.

Questa procedura, tuttavia, non produce una permutazione casuale uniforme, poiché può produrre solo n permutazioni diverse. Pertanto, n permutazioni si verificano con probabilità $1/n$, mentre le restanti $n! - n$ permutazioni si verificano con probabilità 0.

Soluzioni scelte per il Capitolo 6: Heapsort

Soluzione dell'Esercizio 6.1-1

Poiché un heap è un albero binario quasi completo (ovvero completo a tutti i livelli con la possibile eccezione di quello inferiore), contiene al più $2^{h+1} - 1$ elementi (se è completo) e almeno $2^h - 1 + 1 = 2^h$ (se il livello inferiore contiene un solo elemento mentre tutti gli altri sono completi).

Soluzione dell'Esercizio 6.1-2

Dato un heap di n elementi e di altezza h, sappiamo dall'Esercizio 6.1-1 che

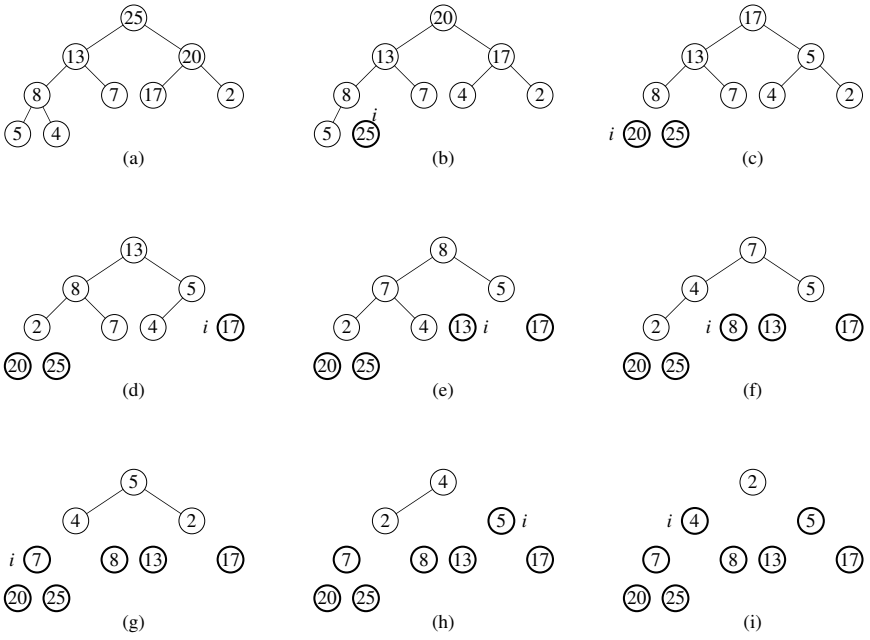
$$2^h \leq n \leq 2^{h+1} - 1 < 2^{h+1} .$$

Dunque $h \leq \lg n < h+1$, e poiché h è intero, abbiamo $h = \lfloor \lg n \rfloor$ (per definizione di $\lfloor\ \rfloor$).

Soluzione dell'Esercizio 6.2-7

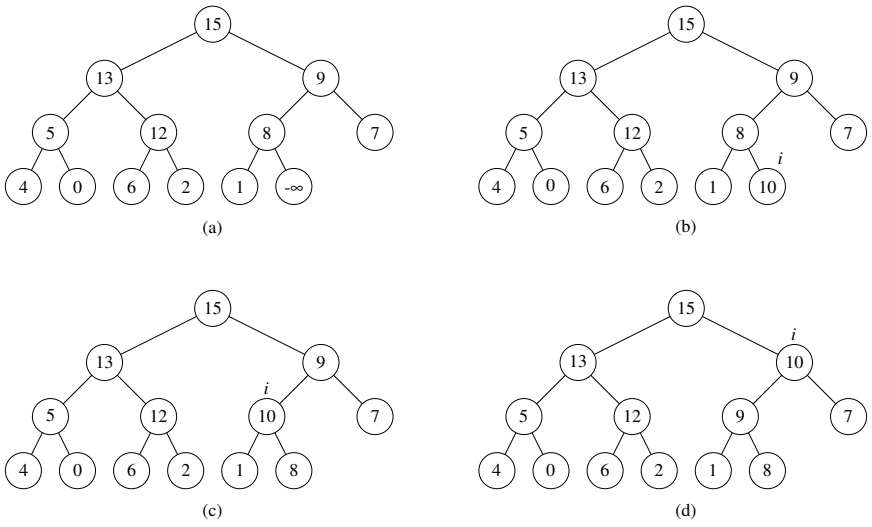
Se nella radice c'è un valore minore di tutti i valori nei sottoalberi di sinistra e di destra, la procedura MAX-HEAPIFY verrà richiamata in modo ricorsivo fino a raggiungere una foglia. Per fare in modo che le chiamate ricorsive percorrano il percorso più lungo fino a una foglia, basta scegliere dei valori che facciano in modo che MAH-HEAPIFY esegua la ricorrenza sempre sul figlio sinistro. Poiché segue il ramo sinistro quando il figlio sinistro è maggiore o uguale al figlio destro, mettendo 0 nella radice e 1 in tutti gli altri nodi, si otterrà questo risultato. Con questi valori, MAX-HEAPIFY verrà chiamata h volte (dove h è l'altezza dell'heap, ovvero il numero di archi nel percorso più lungo dalla radice a una foglia), quindi il suo tempo di esecuzione sarà $\Theta(h)$ (poiché ogni chiamata esegue un lavoro $\Theta(1)$), ovvero $\Theta(\lg n)$. Poiché abbiamo un caso in cui il tempo di esecuzione di MAX-HEAPIFY è proprio $\Theta(\lg n)$, il suo tempo di esecuzione nel caso peggiore è $\Omega(\lg n)$.

Soluzione dell'Esercizio 6.4-1



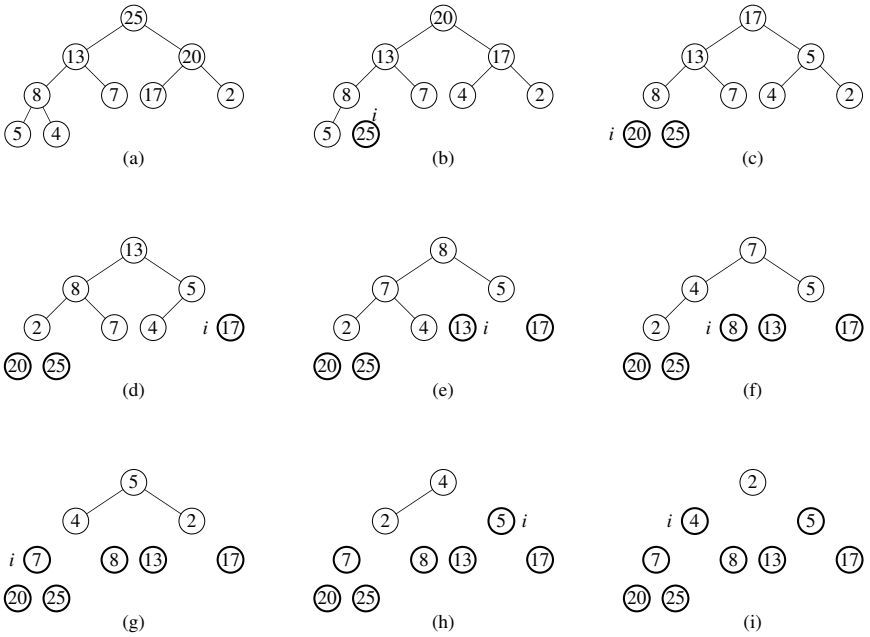
A	2	4	5	7	8	13	17	20	25

Soluzione dell'Esercizio 6.5-2



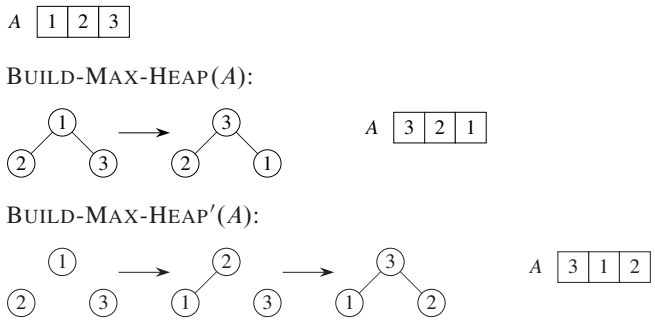
Il tempo di esecuzione è $O(\lg n)$, più il costo della corrispondenza tra gli oggetti della coda di priorità e gli indici dell'array.

Soluzione dell'Esercizio 6.4-1



Soluzione del Problema 6-1

a. Le procedure BUILD-MAX-HEAP e BUILD-MAX-HEAP′ non producono sempre lo stesso heap quando vengono eseguite sullo stesso array di input. Considerate il seguente controesempio con array di input $A = \langle 1, 2, 3 \rangle$.



b. Un limite superiore di $O(n \lg n)$ segue immediatamente dal fatto che ci sono $n - 1$ chiamate a MAX-HEAP-INSERT, ognuna delle quali richiede un tempo $O(\lg n)$. Per un limite inferiore di $\Omega(n \lg n)$, considerate il caso in cui l'array di input sia dato in ordine strettamente crescente. Allora ogni chiamata a MAX-HEAP-INSERT fa sì che la procedura HEAP-INCREASE-KEY salga fino alla radice. Poiché la profondità del nodo i è $\lfloor \lg i \rfloor$, il tempo totale è

$$\begin{aligned}
\sum_{i=1}^{n} \Theta(\lfloor \lg i \rfloor) &\geq \sum_{i=\lceil n/2 \rceil}^{n} \Theta(\lfloor \lg \lceil n/2 \rceil \rfloor) \\
&\geq \sum_{i=\lceil n/2 \rceil}^{n} \Theta(\lfloor \lg(n/2) \rfloor) \\
&= \sum_{i=\lceil n/2 \rceil}^{n} \Theta(\lfloor \lg n - 1 \rfloor) \\
&\geq (n/2) \cdot \Theta(\lg n) \\
&= \frac{n(n-1)}{2} \cdot \frac{1}{2} \\
&= \Omega(n \lg n) \, .
\end{aligned}$$

Nel caso peggiore, quindi, la procedura BUILD-MAX-HEAP′ richiede un tempo $\Theta(\lg n)$ per costruire un heap di n elementi.

26

Soluzioni scelte per il Capitolo 7: Quicksort

Soluzione dell'Esercizio 7.2-3

Supponiamo che la procedura PARTITION venga chiamata su un sottoarray $A[p:r]$ i cui elementi sono tutti diversi tra loro e in ordine decrescente. PARTITION sceglie l'elemento più piccolo, cioè $A[r]$, come pivot. Ogni test della riga 4 risulta falso, quindi nessun elemento viene scambiato durante l'esecuzione del ciclo **for**. Prima che PARTITION restituisca un valore, la riga 6 trova che $i = p-1$ e quindi scambia $A[p]$ con $A[r]$. PARTITION restituisce p come posizione del pivot. Il sottoarray contenente gli elementi minori o uguali al pivot è vuoto. Il sottoarray contenente elementi maggiori del pivot, ovvero $A[p+1:r]$, contiene tutti gli elementi tranne il pivot ed è in ordine decrescente, tranne per il fatto che l'elemento massimo di questa sottoarray si trova in $A[r]$.

Quando la procedura QUICKSORT chiama PARTITION su $A[p:q-1]$, non cambia nulla, poiché questo sottoarray è vuoto. Quando QUICKSORT chiama PARTITION su $A[q+1:r]$ il pivot è l'elemento più grande del sottoarray. Sebbene ogni test della riga 4 risulti vero, nella riga 6 gli indici i e j sono sempre uguali, per cui, proprio come nel caso in cui il pivot è l'elemento più piccolo, non viene scambiato nessun elemento durante l'esecuzione del ciclo **for**. Prima che la procedura PARTITION termini, la riga 6 trova che $i = r-1$, quindi lo scambio nella riga 6 lascia il pivot in $A[r]$. PARTITION restituisce r come posizione del pivot. Ora il sottoarray contenente gli elementi minori o uguali al pivot contiene tutti gli elementi tranne il pivot ed è in ordine decrescente, mentre il sottoarray che contiene gli elementi maggiori del pivot è vuoto. La chiamata successiva a PARTITION, quindi, è su un sottoarray in ordine decrescente e si ricade nel primo caso visto sopra.

Pertanto, ogni chiamata ricorsiva viene fatta su un sottoarray che ha un solo elemento in meno rispetto alla chiamata precedente, è perciò otteniamo la ricorren-

za per il tempo di esecuzione dell'algoritmo $T(n) = T(n-1) + \Theta(n)$, la cui soluzione è $\Theta(n^2)$.

Soluzione dell'Esercizio 7.2-5

La profondità minima segue un percorso che prende sempre la parte più piccola della partizione, cioè che moltiplica il numero di elementi per α. Un livello della ricorsione riduce il numero di elementi da n ad αn, e i livelli della ricorsione riducono il numero di elementi a $\alpha^i n$. In una foglia c'è un solo elemento rimanente e quindi in una foglia di profondità minima m si ha $\alpha^m n = 1$. Quindi, $\alpha^m = 1/n$. Passando ai logaritmi, si ottiene $m \lg \alpha = -\lg n$, ovvero $m = -\lg n / \lg \alpha$ (questo numero è positivo perché $0 < \alpha < 1$ implica $\lg \alpha < 0$).

Analogamente, il percorso di massima profondità corrisponde a prendere sempre la parte più grande della partizione, cioè a mantenere ogni volta una frazione β di elementi. La profondità massima M viene raggiunta quando rimane un elemento, cioè quando $\beta^M n = 1$. Quindi $M = -\lg n / \lg \beta$ (anche qui questo numero è positivo, perché $0 < \beta < 1$ implica $\lg \beta < 0$).

Tutti i calcoli sono approssimati, perché ignoriamo sempre le parti intere inferiori e superiori.

Soluzioni scelte per il Capitolo 8: Ordinamento in tempo lineare

Soluzione dell'Esercizio 8.1-3

Se l'ordinamento viene eseguito in tempo lineare per m permutazioni dell'input, allora l'altezza h della porzione dell'albero di decisione formata dalle m foglie corrispondenti e da tutti i loro antenati è lineare.

Seguiamo lo stesso ragionamento nella dimostrazione del Teorema 8.1 per provare che questo è impossibile se $m = n!/2$, $m = n!/n$ e $m = n!/2^n$.

Abbiamo $2^h \geq m$, da cui $h \geq \lg m$. Per tutti i possibili valori di m indicati sopra si ha $m = \Omega(n \lg n)$, e quindi $h = \Omega(n \log n)$.

In particolare, dall'Equazione (3.25), segue

$$\begin{aligned}
\lg \frac{n!}{2} &= \lg n! - 1 \geq n \lg n - n \lg e - 1 \,, \\
\lg \frac{n!}{n} &= \lg n! - \lg n \geq n \lg n - n \lg e - \lg n \,, \\
\lg \frac{n!}{2^n} &= \lg n! - n \geq n \lg n - n \lg e - n \,.
\end{aligned}$$

Soluzione dell'Esercizio 8.2-3

La seguente soluzione risponde anche all'Esercizio 8.2-2.

Notate che la giustificazione della correttezza riportato nel libro non dipende dall'ordine in cui A viene elaborato. L'algoritmo è corretto sia che A venga analizzato da sinistra a destra che da destra a sinistra.

Ma l'algoritmo modificato non è stabile. Come in precedenza, nell'ultimo ciclo **for** un elemento uguale a quello preso in precedenza da A viene collocato prima di quello precedente (cioè in una posizione di indice inferiore) nell'array di output B. L'algoritmo originale era stabile perché un elemento che veniva preso da A dopo

un altro partiva da un indice inferiore rispetto a a quello dell'elemento precedente. Nell'algoritmo modificato, invece, un elemento che viene preso da A dopo un altro inizia con un indice più alto rispetto a quello dell'elemento precedente.
In particolare, l'algoritmo colloca ancora gli elementi con valore k nelle posizioni da $C[k-1]+1$ a $C[k]$, ma in ordine inverso rispetto a come appaiono in A. Riscriviamo ora la procedura COUNTING-SORT mantenendo la modifica, ma facendo in modo che gli elementi con lo stesso valore vengano scritti nell'array di output in ordine di indice crescente, e quindi che l'algoritmo sia stabile.

```
COUNTING-SORT(A, n, k)
    siano B[1 : n], C[0 : k] e L[0 : k] tre nuovi array
    for i = 0 to k
        C[i] = 0
    for j = 1 to n
        C[A[j]] = C[A[j]] + 1
    // Ora in C[i] c'è il numero di elementi uguali a i.
    L[0] = 1
    for i = 1 to k
        L[i] = L[i − 1] + C[i − 1]
    // Ora in L[i] c'è l'indice del primo elemento di A uguale a a i.
    for j = 1 to n
        B[L[A[j]]] = A[j]
        L[A[j]] = L[A[j]] + 1
```

Soluzione dell'Esercizio 8.3-3
Caso base: se $d = 1$, c'è una sola cifra, quindi l'ordinamento su quella cifra ordina l'array.
Passo induttivo: supponendo che il radix sort funzioni per $d-1$ cifre, dimostriamo che funziona per d cifre.
Il radix sort ordina separatamente su ogni cifra, a partire dalla prima cifra. Pertanto, il radix sort su d cifre, che ordina sulle cifre $1, \ldots, d$, è equivalente al radix sort sulle $d-1$ cifre di ordine inferiore seguito da un ordinamento sulla cifra d. Per l'ipotesi di induzione, l'ordinamento delle $d-1$ cifre di ordine inferiore funziona, quindi appena prima dell'ordinamento sulla cifra d, gli elementi sono ordinati secondo le loro $d-1$ cifre di ordine inferiore.
L'ordinamento sulla cifra d ordinerà gli elementi in base alla loro cifra d-esima. Consideriamo due elementi, a e b, la cui d-esima cifra è rispettivamente a_d e b_d.

- Se $a_d < b_d$, l'ordinamento metterà a prima di b, il che è corretto, poiché $a < b$ indipendentemente dalle cifre di ordine inferiore.

- Se $a_d > b_d$, l'ordinamento metterà a dopo b, il che è corretto, poiché $a > b$ indipendentemente dalle cifre di ordine inferiore.

- Se $a_d = b_d$, l'ordinamento lascerà a e b nello stesso ordine in cui erano, perché è stabile. Ma questo ordine è già corretto, poiché, quando la d-esima cifra è uguale, l'ordine corretto di a e b è determinato dalle $d - 1$ cifre di ordine inferiore, e gli elementi sono già ordinati in base alle loro $d - 1$ cifre di ordine inferiore.

Se l'ordinamento intermedio non fosse stabile, potrebbe rimescolare gli elementi le cui d-esime cifre sono uguali, elementi che *erano* nell'ordine giusto dopo l'ordinamento sulle loro cifre di ordine inferiore.

Soluzione dell'Esercizio 8.3-5

Basta considerare i numeri come numeri di 3 cifre in base n, quindi ogni cifra varia da 0 a $n - 1$. Ordiniamo questi numeri di 3 cifre con il radix sort.
Ci sono 3 chiamate del counting sort, ognuna delle quali richiede un tempo $\Theta(n + n) = \Theta(n)$, e dunque il tempo totale è ancora $\Theta(n)$.

Soluzione del Problema 8-1

a. Affinché un algoritmo di ordinamento per confronti A possa ordinare correttamente, non ci possono essere due permutazioni in input che raggiungono la stessa foglia dell'albero di decisione, e quindi ci devono essere almeno $n!$ foglie raggiunte in T_A, una per ogni possibile permutazione in input. Poiché A è un algoritmo deterministico, deve raggiungere sempre la stessa foglia quando gli viene data in ingresso una particolare permutazione, quindi vengono raggiunte al massimo $n!$ foglie (una per ogni permutazione). Dunque vengono raggiunte esattamente $n!$ foglie, una per ogni permutazione in input.

Queste $n!$ foglie avranno ciascuna una probabilità $1/n!$ di venire raggiunte, poiché ognuna delle $n!$ possibili permutazioni è l'input con probabilità $1/n!$. Le foglie rimanenti avranno probabilità 0, poiché non vengono raggiunte da nessun input.

Senza perdita di generalità, nel il resto del problema possiamo assumere che i percorsi che portano solo a foglie con probabilità 0 non siano presenti nell'albero, poiché non possono influire sul tempo di esecuzione dell'ordinamento. In altre parole, possiamo assumere che l'albero A_T consista solo delle $n!$ foglie etichettate con $1/n!$ e dei loro antenati.

b. Se $k > 1$, allora la radice di T non è una foglia. Tutte le foglie di T devono essere foglie di TL o di TR. Poiché ogni foglia a profondità h in LT o in RT ha profondità $h + 1$ in T, $D(T)$ deve essere la somma di $D(LT)$, $D(RT)$ e k, che è il numero di tutte le foglie. Per provarlo, sia $d_T(x)$ la profondità del nodo x in T e $\text{leaves}(T')$ l'insieme di tutte le foglie di un albero T', allora

$$\begin{aligned}
D(T) &= \sum_{x \in \text{leaves}(T)} d_T(x) \\
&= \sum_{x \in \text{leaves}(LT)} d_T(x) + \sum_{x \in \text{leaves}(RT)} d_T(x) \\
&= \sum_{x \in \text{leaves}(LT)} (d_{LT}(x) + 1) + \sum_{x \in \text{leaves}(RT)} (d_{RT}(x) + 1) \\
&= \sum_{x \in \text{leaves}(LT)} d_{LT}(x) + \sum_{x \in \text{leaves}(RT)} d_{RT}(x) + \sum_{x \in \text{leaves}(T)} 1 \\
&= D(LT) + D(RT) + k \,.
\end{aligned}$$

c. Per dimostrare che $d(k) = \min\{d(i) + d(k-i) + k \ : \ 1 \le i \le k-1\}$, proveremo separatamente che $d(k) \le \min\{d(i)+d(k-i)+k \ : \ 1 \le i \le k-1\}$ e che $d(k) \ge \min\{d(i) + d(k-i) + k \ : \ 1 \le i \le k-1\}$

- Dimostriamo che $d(k) \le \min\{d(i) + d(k-i) + k \ : \ 1 \le i \le k-1\}$ provando che $d(k) \le d(i) + d(k-i) + k$ per $i = 1, 2, \ldots, k-1$. Per l'Esercizio B.5-4, c'è un albero binario pieno con i foglie per ogni i da 1 a $k-1$. Quindi, possiamo costruire un albero di decisione LT con i foglie e un albero di decisione RT con $k-i$ foglie tali che $D(LT) = d(i)$ e $D(RT) = d(k-i)$. Possiamo poi costruire l'albero T in modo che LT e RT siano i suoi sottoalberi sinistro e destro. Allora

$$\begin{aligned}
d(T) &\le D(T) && \text{per definizione di } d \text{ come minimo di } D(T) \\
&= D(LT) + D(RT) + k && \text{per il punto (b)} \\
&= d(i) + d(k-i) + k && \text{per come sono stati scelti } LT \text{ e } RT \,.
\end{aligned}$$

- Dimostriamo che $d(k) \geq \min\{d(i)+d(k-i)+k \,:\, 1 \leq i \leq k-1\}$ provando che esiste un i in $\{1, 2, \ldots, k-1\}$ tale che $d(k) \geq d(i) + d(k-i) + k$. Consideriamo l'albero T con k foglie tale che $D(T) = d(k)$, siano LT e RT i suoi sottoalberi sinistro e destro e sia i il numero di foglie di LT. Allora $k - i$ è il numero di foglie di RT e

  $$\begin{aligned} d(T) &= D(T) && \text{per definizione di } T \\ &= D(LT) + D(RT) + k && \text{per il punto (b)} \\ &\geq d(i) + d(k-i) + k && \text{per definizione di } d \text{ come minimo di } D(T)\ . \end{aligned}$$

  Né i né $k - i$ può essere 0 (e quindi $1 \leq 1 \leq k - 1$), poiché se uno di questi fosse 0, allora o LT o RT conterrebbero tutte le k foglie di T. In tal caso, la radice di T avrebbe un solo figlio, e dunque T non sarebbe un albero binario pieno e quindi non sarebbe un albero di decisione.

d. Sia $f_k(i) = i \lg i + (k-i)\lg(k-i)$. Per trovare il valore di i che minimizza f_k, troviamo il valore di i per il quale la derivata di f_k rispetto a i è 0:

$$\begin{aligned} f'_k(i) &= \frac{d}{di}\left(\frac{i\ln i + (k-i)\ln(k-i)}{\ln 2}\right) \\ &= \frac{\ln i + 1 - \ln(k-i) - 1}{\ln 2} \\ &= \frac{\ln i - \ln(k-i)}{\ln 2} \end{aligned}$$

è 0 per $i = k/2$. Per verificare che si tratta effettivamente di un punto di minimo (e non di massimo) verifichiamo che la derivata seconda di f_k in $i = k/2$ sia positiva:

$$\begin{aligned} f''_k(i) &= \frac{d}{di}\left(\frac{\ln i - \ln(k-i)}{\ln 2}\right) \\ &= \frac{1}{\ln 2}\left(\frac{1}{i} + \frac{1}{k-i}\right)\ , \end{aligned}$$

da cui

$$\begin{aligned} f''_k(k/2) &= \frac{1}{\ln 2}\left(\frac{2}{k} + \frac{2}{k}\right) \\ &= \frac{1}{\ln 2}\cdot\frac{4}{k} \\ &> 0 \qquad \text{poiché } k > 1\ . \end{aligned}$$

Ora facciamo una sostituzione per provare che $d(k) = \Omega(k \lg k)$. Il caso base dell'induzione è soddisfatto, poiché per ogni costante c si ha $d(1) \geq 0 = c \cdot 1 \cdot \lg 1$. Per il passo induttivo, supponiamo che sia $d(i) \geq ci \lg i$ per $1 \leq i \leq k-1$, dove c è una costante da determinare. Allora

$$\begin{aligned}
d(k) &= \min\{d(i) + d(k-i) + k \ : \ 1 \leq i \leq k-1\} \\
&\geq \min\{c(i \lg i + (k-i)\lg(k-i)) + k \ : \ 1 \leq i \leq k-1\} \\
&= c\left(\frac{k}{2}\lg\frac{k}{2} + \left(k - \frac{k}{2}\right)\lg\left(k - \frac{k}{2}\right)\right) + k \\
&= ck \lg\left(\frac{k}{2}\right) + k \\
&= c(k \lg k - k) + k \\
&= ck \lg k + (k - ck) \\
&\geq cl \lg k \qquad \text{se } c \leq 1
\end{aligned}$$

e dunque $d(k) = \Omega(k \lg k)$.

e. Sfruttando il risultato della parte (d) e il fatto che T_A (come modificato nella nostra soluzione della parte (a)) ha $n!$ foglie, possiamo concludere che

$$D(T_A) \geq d(n!) = \Omega(n! \lg(n!)) \ .$$

$D(T_A)$ è la somma delle lunghezze dei percorsi dell'albero di decisione per l'ordinamento di tutte le permutazioni in input, e le lunghezze dei percorsi sono proporzionali al tempo di esecuzione. Poiché le $n!$ permutazioni hanno la stessa probabilità $1/n!$, il tempo atteso per ordinare n elementi scelti a caso (cioè una permutazione dell'input) è il tempo totale per tutte le permutazioni diviso per $n!$, ovvero

$$\frac{\Omega(n! \lg(n!))}{n!} = \Omega(\lg(n!)) = \Omega(n \lg n) \ .$$

f. Mostreremo come modificare un albero di decisione (o un algoritmo) randomizzato per definire un albero di decisione (o un algoritmo) deterministico che sia almeno altrettanto efficiente di quello randomizzato in termini di numero medio di confronti.

In ogni nodo randomizzato, scegliamo il figlio con il sottoalbero più piccolo (il sottoalbero con il minor numero medio di confronti su un percorso verso

una foglia). Eliminiamo tutti gli altri figli del nodo randomizzato e dividiamo il nodo randomizzato stesso.

L'algoritmo deterministico corrispondente a questo albero modificato funziona ancora, perché l'algoritmo randomizzato funzionava indipendentemente dal percorso seguito da ciascun nodo randomizzato.

Il numero medio di confronti per l'algoritmo modificato non è superiore al numero medio per l'albero randomizzato originale, poiché abbiamo scartato i sottoalberi con la media più alta in ogni caso. In particolare, ogni volta che scartiamo un nodo randomizzato, la media complessiva resta minore o uguale a quella che era prima, poiché

- lo stesso insieme di permutazioni in ingresso raggiunge il sottoalbero modificato come prima, ma tali input vengono gestiti in un tempo inferiore o uguale alla media di prima, e
- il resto dell'albero non viene modificato.

L'algoritmo randomizzato richiede quindi in media almeno lo stesso tempo del corrispondente algoritmo deterministico (abbiamo dimostrato che il tempo medio di esecuzione di un ordinamento per confronto deterministico è $\Omega(n \lg n)$, quindi il tempo atteso per un ordinamento per confronto randomizzato è $\Omega(n \lg n)$).

36

Soluzioni scelte per il Capitolo 9: Mediane e statistiche d'ordine

Soluzione dell'Esercizio 9.3-1

Per gruppi di 7, l'algoritmo funziona ancora in tempo lineare. Il numero g di gruppi è al massimo $n/7$. Ci sono almeno $4(\lfloor n/2 \rfloor + 1) \geq 2g$ elementi maggiori o uguali al pivot e almeno $4\lceil n/2 \rceil \geq 2g$ elementi minori o uguali al pivot. Rimangono quindi al massimo $7g - 2g = 5g \leq 5n/7$ elementi nella chiamata ricorsiva. La ricorrenza diventa $T(n) \leq T(n/7) + T(5n/7) + O(n)$, che ha soluzione $T(n) = O(n)$, come si dimostra facilmente per sostituzione seguendo il metodo visto nel libro.

In realtà, qualunque raggruppamento con un numero dispari e maggiore o uguale a 5 di elementi continua a funzionare, producendo un algoritmo con complessità lineare.

Soluzione dell'Esercizio 9.3-3

Una modifica al quicksort che consente di eseguirlo in tempo $O(n \lg n)$ nel caso peggiore utilizza la procedura deterministica PARTITION-AROUND che prende come parametro di input un elemento intorno a cui eseguire la partizione.

La procedura SELECT prende un array A, gli estremi p e r del sottoarray in A e il rango i di una statistica d'ordine, e in tempo lineare rispetto alla dimensione del sottoarray $A[p:r]$ restituisce l'i-esimo elemento più piccolo in $A[p:r]$.

```
BEST-CASE-QUICKSORT(A, p, r)
  if p < r
      i = ⌊(r − p + 1)/2⌋
      x = SELECT(A, p, r, i)
      q = PARTITION-AROUND(A, p, r, x)
      BEST-CASE-QUICKSORT(A, p, q − 1)
      BEST-CASE-QUICKSORT(A, q + 1, r)
```

Per un array di n elementi, il sottoarray più grande su cui BEST-CASE-QUICKSORT esegue la ricorrenza ha $n/2$ elementi. Questa situazione si verifica quando $n = r - p + 1$ è pari: in tal caso il sottoarray $A[q+1:r]$ ha $n/2$ elementi e $A[p:q-1]$ ha $n/2 - 1$ elementi.
Poiché BEST-CASE-QUICKSORT fa sempre delle chiamate ricorsive solo su sottoarray che hanno al massimo dimensione uguale a metà di quello di partenza, la ricorrenza per il tempo di esecuzione nel caso peggiore è $T(n) \leq 2T(n/2) + \Theta(n) = O(n \lg n)$.

Soluzione dell'Esercizio 9.3-6
Supponiamo di avere una procedura MEDIAN che prende come parametri un array A e gli indici p e r e restituire il valore dell'elemento mediano di $A[p:r]$ in tempo $O(n)$ nel caso peggiore.
Avendo a disposizione MEDIAN, quello che segue è la procedura SELECT$'$ che in tempo lineare trova l'i-esimo elemento più piccolo di $A[p:r]$. Questo algoritmo utilizza la procedura deterministica PARTITION-AROUND che tra i parametri di input prende anche un elemento intorno a cui eseguire la partizione.

```
SELECT'(A, p, r, i)
    if p == r
        return A[p]
    x = MEDIAN(A, p, r)
    q = PARTITION-AROUND(A, p, r, x)
    k = q − p + 1
    if i == k
        return A[q]
    elseif i < k
        return SELECT'(A, p, q − 1, i)
    else return SELECT'(A, q + 1, r, i − k)
```

Poiché x è la mediana di $A[p:r]$, i sottoarray $A[p:q-1]$ e $A[q+1:r]$ hanno un numero di elementi che è al massimo la metà di quelli di $A[p:r]$. La ricorrenza per il tempo di esecuzione di SELECT$'$ nel caso peggiore è quindi $T(n) \leq T(n/2) + O(n) = O(n)$.

Soluzione del Problema 9-1
Assumiamo che gli n numeri si trovino già in un array.

a. Ordinate i numeri utilizzando il merge sort o l'heapsort, che richiedono un tempo $\Theta(n \lg n)$ nel caso peggiore (non usate il quicksort o l'insertion sort, che possono richiedere un tempo $\Theta(n^2)$). Mettere gli i elementi più grandi (direttamente accessibili nell'array ordinato) nell'array di output: questo richiede un tempo $\Theta(i)$.

Quindi il tempo di esecuzione totale nel caso peggiore è $\Theta(n \lg n + i) = \Theta(n \lg n)$ (poiché $i \leq n$).

b. Potete implementare la coda di priorità come un heap. Costruite l'heap usando BUILD-HEAP, che richiede un tempo $\Theta(n)$, quindi chiamate i volte la procedura HEAP-EXTRACT-MAX per ottenere gli i elementi più grandi, questo in tempo $\Theta(i \lg n)$ nel caso peggiore, e infine li memorizzate nell'array di output in ordine inverso di estrazione. Il tempo richiesto per estrarre gli i elementi più grandi è davvero $\Theta(i \lg n)$ poiché

- i estrazioni da un heap con $O(n)$ elementi richiedono un tempo $i \cdot O(\lg n) = O(i \lg n)$, e
- metà delle i estrazioni provengono da un heap con almeno $n/2$ elementi, quindi queste $i/2$ estrazioni richiedono un tempo $(i/2) \cdot \Omega(\lg(n/2)) = \Omega(i \lg n)$ nel caso peggiore.

Di conseguenza, il tempo di esecuzione totale nel caso peggiore è $\Theta(n+i \lg n)$.

c. Usate la procedura SELECT del Paragrafo 9.3 per trovare l'i-esimo numero più grande in tempo $\Theta(n)$. Partizionate intorno a questo numero in tempo $\Theta(n)$. Ordinate gli i numeri più grandi in tempo $\Theta(i \lg i)$ nel caso peggiore (con il merge sort o l'heapsort).

Di conseguenza, il tempo di esecuzione totale nel caso peggiore è $\Theta(n+i \lg i)$.

Notate che il metodo (c) è sempre asintoticamente buono almeno quanto gli altri due metodi e che il metodo (b) è asintoticamente buono almeno quanto (a).

40

Soluzioni scelte per il Capitolo 11: Hashing

Soluzione dell'Esercizio 11.2-1

Per ogni coppia di chiavi k ed l, con $k \neq l$, definiamo la variabile casuale indicatrice $X_{kl} = \mathrm{I}\{h(k) = h(l)\}$. Poiché stiamo assumendo un hashing uniforme e indipendente, abbiamo $\Pr\{X_{kl} = 1\} = \Pr\{h(k) = h(l)\} = 1/m$, e dunque $\mathrm{E}[X_{kl}] = 1/m$.

Definiamo ora la variabile casuale $Y = \sum_{k \neq l} X_{kl}$; il numero atteso di collisioni è

$$\begin{aligned}
\mathrm{E}[Y] &= \mathrm{E}\left[\sum_{k \neq l} X_{kl}\right] \\
&= \sum_{k \neq l} \mathrm{E}[X_{kl}] \qquad \text{(per linearità del valore atteso)} \\
&= \binom{n}{2}\frac{1}{m} \\
&= \frac{n(n-1)}{2} \cdot \frac{1}{m} \\
&= \frac{n(n-1)}{2m} .
\end{aligned}$$

Soluzione dell'Esercizio 11.2-4

La marcatura di ogni cella indica se la cella è libero.

- Una cella libera si trova nella free list, che è una lista doppiamente concatenata contenente tutte le celle non utilizzate. La cella quindi contiene due puntatori.
- Una cella usata contiene un elemento e un puntatore (eventualmente NIL) all'elemento successivo che ha viene mandato in questa cella (naturalmente, il puntatore punta a un altra cella della tabella).

Funzionamento.

- **Inserimento.**
  - Se l'elemento viene mandato in una cella libera, è sufficiente rimuovere la cella dalla free list e memorizzarvi l'elemento (con un puntatore NIL). La free list deve essere doppiamente concatenata per poter eseguire questa eliminazione in tempo $O(1)$.
  - Se l'elemento viene mandato in una cella j già in uso, verifichiamo se l'elemento x già presente vi "appartiene" (cioè, se anche la sua chiave viene mandata nella cella j).
    * In caso affermativo, aggiungiamo il nuovo elemento alla catena di elementi in questa cella. Per fare ciò, allochiamo una cella libera (ad esempio, prendendo la cima della free list) per il nuovo elemento e mettiamo questa nuova cella in cima alla lista a cui punta la cella a cui deve essere associata (cioè j).
    * In caso contrario, x fa parte della catena di un'altra cella. Lo spostiamo in una nuova cella allocandone una dalla free list, copiando il contenuto della vecchia cella (cioè j), ovvero l'elemento x e il puntatore, nella nuova cella e aggiornando il puntatore nella cella che puntava a j per puntare alla nuova cella. Quindi inseriamo il nuovo elemento nella cella ora vuota come di consueto. Per aggiornare il puntatore a j, è necessario trovarlo cercando nella catena di elementi a partire dalla cella a cui punta x.
- **Eliminazione.** Sia j la cella in cui viene mandato l'elemento x da cancellare.
  - Se x è l'unico elemento in j (cioè, se j non punta ad altri elementi), basta liberare la cella, inserendola in cima alla free list.
  - Se x è in j ma c'è un puntatore a una catena di altri elementi, spostiamo il primo elemento puntato nella cella j e liberiamo la cella in cui si trovava.
  - Se x viene trovato seguendo un puntatore da j, basta liberare la cella di x e separarla dalla catena (cioè aggiornare la cella che puntava a x per puntare al suo successore).
- **Ricerca.** Controlliamo la cella a cui punta la chiave e, se non è l'elemento desiderato, seguiamo la catena di puntatori dalla cella.

Tutte queste le operazioni richiedono un tempo atteso $O(1)$ per lo stesso motivo riportato nel libro: il tempo atteso per la ricerca delle catene è $O(1+\alpha)$, indipendentemente da dove sono memorizzate le catene, e il fatto che tutti gli elementi siano memorizzati nella tabella significa che $\alpha \leq 1$. Se la free list fosse singolarmente concatenata, le operazioni che comportano la rimozione di una cella dalla free list non verrebbero più eseguite in tempo $O(1)$.

Soluzione del Problema 11-3

a. Una chiave data viene assegnata a una cella data con probabilità $1/n$. Supponiamo di prendere un insieme di k chiavi. La probabilità che queste k chiavi siano inserite nella cella in questione e che tutte le altre chiavi siano inserite altrove è

$$\left(\frac{1}{n}\right)^k \left(1-\frac{1}{n}\right)^{n-k} .$$

Poiché ci sono $\binom{n}{k}$ modi diversi di scegliere le nostre k chiavi, otteniamo

$$Q_k = \left(\frac{1}{n}\right)^k \left(1-\frac{1}{n}\right)^{n-k} \binom{n}{k} .$$

b. Per $i = 1, 2, \ldots, n$, sia X_i la variabile casuale che indica il numero di chiavi che vengono assegnate alla cella i, e sia A_i l'evento che corrisponde a $X_i = k$, ovvero al fatto che esattamente k chiavi vengono associate alla cella i. Dalla parte (a), segue che $\Pr\{A\} = Q_k$, e quindi

$$\begin{aligned}
P_k &= \Pr\{M = k\} \\
&= \Pr\{\max\{X_i : 1 \leq i \leq n\} = k\} \\
&= \Pr\{\text{esiste } i \text{ tale che } X_i = k \text{ e } X_i \leq k \text{ per } i = 1, 2, \ldots, n\} \\
&\leq \Pr\{\text{esiste } i \text{ tale che } X_i = k\} \\
&= \Pr\{A_1 \cup A_2 \cdots \cup A_n\} \\
&\leq \Pr\{A_1\} + \Pr\{A_2\} + \cdots + \Pr\{A_n\} \qquad \text{(per la disuguaglianza (C.21))} \\
&= nQ_k .
\end{aligned}$$

c. Iniziamo osservando due semplici fatti. Il primo è che $1 - 1/n < 1$ e che $n - k \geq 0$ implica $(1 - 1/n)^{n-k} \leq 1$. Il secondo è che $n!/(n-k)! = n \cdot (n-1) \cdot (n-2) \ldots (n-k+1) < n^k$. Da queste semplici osservazioni, insieme alla disequazione (3.25) che afferma che $k! > (k/e)^k$, segue che

$$\begin{aligned} Q_k &= \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \frac{n!}{k!(n-k)!} \\ &\leq \frac{n!}{n^k k!(n-k)!} \qquad \text{(poiché } (1-1/n)^{n-k} \leq 1\text{)} \\ &< \frac{1}{k!} \qquad \text{(poiché } n!/(n-k)! < n^k\text{)} \\ &< \frac{e^k}{k^k} \qquad \text{(poiché } k! > (k/e)^k\text{)} . \end{aligned}$$

d. Notate che quando $n = 2$, si ha $\lg \lg n = 0$, quindi, per essere precisi, dobbiamo assumere che sia $n \geq 3$.

Nella parte (c) abbiamo dimostrato che per ogni k si ha $Q_k < e^k/k^k$; in particolare, questa disuguaglianza vale per k_0. Di conseguenza, basta dimostrare che $e^{k_0}/k_0^{k_0} < 1/n^3$ o, equivalentemente, che $n^3 < k_0^{k_0}/e^{k_0}$.

Passando ai logaritmi a entrambi i membri, otteniamo la condizione equivalente

$$\begin{aligned} 3 \lg n &< k_0(\lg k_0 - \lg e) \\ &= \frac{c \lg n}{\lg \lg n}(\lg c + \lg \lg n - \lg \lg \lg n - \lg e) . \end{aligned}$$

Dividendo ambo i membri per $\lg n$, otteniamo la condizione

$$\begin{aligned} 3 &< \frac{c}{\lg \lg n}(\lg c + \lg \lg n - \lg \lg \lg n - \lg e) \\ &= c\left(1 + \frac{\lg c - \lg e}{\lg \lg n} - \frac{\lg \lg \lg n}{\lg \lg n}\right) . \end{aligned}$$

Sia x l'ultima espressione tra parentesi, ovvero

$$x = \left(1 + \frac{\lg c - \lg e}{\lg \lg n} - \frac{\lg \lg \lg n}{\lg \lg n}\right) .$$

Allora dobbiamo dimostrare che esiste una costante $c > 1$ tale che $3 < cx$.

Notiamo che $\lim_{n\to\infty} x = 1$, e quindi esiste n_0 tale che $x \geq 1/2$ per ogni $n \geq n_0$. Quindi, ogni costante $c > 6$ fa al caso nostro per $n \geq n_0$.

Ora dobbiamo gestire i valori di n più piccoli, cioè quelli tali che $3 \leq n < n_0$. Per farlo, iniziamo osservando che n è un intero, e quindi ci sono solo un numero finito di valori che può assumere n tra 3 e n_0. Possiamo calcolare l'espressione x per ciascuno di questi valori di n e determinare un valore di c tale che per ognuno di essi si abbia $3 < cx$. In coclusione, il valore di c di cui abbiamo bisogno sarà il massimo tra

- 6, che va bene per tutti gli $n \geq n_0$, e
- $\max\{c : 3 < cx \text{ e } 3 \leq n < n_0\}$, cioè il più grande valore di c che va bene per ogni n tra 3 e n_0.

Abbiamo dunque dimostrato che $Q_k < 1/n^3$, come richiesto.

Per provare che $P_k < 1/n^2$ per ogni $k \geq k_0$, osserviamo che dalla parte (b) segue che $P_k \leq Q_k$ per ogni $k > k_0$. Prendendo $k = k_0$ troviamo $P_{k_0} \leq nQ_{k_0} < n \cdot (1/n^3) = 1/n^2$. Per $k > k_0$, proviamo ora che possiamo trovare una costante c tale che $Q_k < 1/n^3$ per ogni $k \geq k_0$, per poter quindi concludere che $P_k < 1/n^2$ per ogni $k \geq k_0$.

Per trovare c come ci serve, sia c grande abbastanza perché sia $k_0 > 3 > e$. Allora $e/k < 1$ per ogni $k \geq k_0$, e dunque l'espressione e^k/k^k decresce al crescere di k. Di conseguenza

$$\begin{aligned} Q_k &< e^k/k^k \\ &\leq e^{k_0}/k^{k_0} \\ &= Q_{k_0} \\ &< 1/n^3 \end{aligned}$$

per ogni $k \geq k_0$.

e. Il valore atteso di M è

$$\begin{aligned}
\mathrm{E}[M] &= \sum_{k=0}^{n} k \cdot \Pr\{M = k\} \\
&= \sum_{k=0}^{k_0} k \cdot \Pr\{M = k\} + \sum_{k=k_0+1}^{n} k \cdot \Pr\{M = k\} \\
&\leq \sum_{k=0}^{k_0} k_0 \cdot \Pr\{M = k\} + \sum_{k=k_0+1}^{n} n \cdot \Pr\{M = k\} \\
&= k_0 \cdot \Pr\{M \leq k_0\} + n \cdot \Pr\{M > k_0\} \ ,
\end{aligned}$$

che è proprio quello che dovevamo dimostrare, visto che $k_0 = c \lg n / \lg \lg n$.

Per dimostrare che $\mathrm{E}[M] = O(\lg n / \lg \lg n)$, osserviamo che $\Pr\{M \leq k_0\} \leq 1$ e che

$$\begin{aligned}
\Pr\{M > k_0\} &= \sum_{k=k_0+1}^{n} \Pr\{M = k\} \\
&= \sum_{k=k_0+1}^{n} P_k \\
&< \sum_{k=k_0+1}^{n} 1/n^2 \qquad \text{(per la parte (d))} \\
&= 1/n \ .
\end{aligned}$$

Ne concludiamo che

$$\begin{aligned}
\mathrm{E}[M] &\leq k_0 \cdot 1 + n \cdot (1/n) \\
&= k_0 + 1 \\
&= O(\lg n / \lg \lg n) \ .
\end{aligned}$$

Soluzioni scelte per il Capitolo 12: Alberi binari di ricerca

Soluzione dell'Esercizio 12.1-2

In un heap, la chiave di un nodo è maggiore o uguale a entrambe le chiavi dei suoi figli. In un albero binario di ricerca, la chiave di un nodo è maggiore o uguale alla chiave del suo figlio sinistro, ma minore o uguale alla chiave del suo figlio destro. La proprietà di min-heap, a differenza della proprietà degli alberi binari di ricerca, non permette di stampare direttamente i nodi in ordine crescente, perché non dice quale sottoalbero di un nodo contiene l'elemento da stampare prima di quel nodo. In un min-heap, il più grande elemento che è più piccolo del nodo potrebbe trovarsi in uno qualunque dei due sottoalberi.

Notate che se la proprietà di min-heap potesse essere utilizzata per stampare le chiavi in ordine crescente in tempo $O(n)$, avremmo un algoritmo con complessità al più lineare per risolvere il problema dell'ordinamento, perché la costruzione dell'heap richiede solo un tempo $O(n)$. Ma sappiamo dal Teorema 8.1 che un ordinamento per confronti richiede un tempo $\Omega(n \lg n)$.

Soluzione dell'Esercizio 12.2-7

Notate che una chiamata a TREE-MINIMUM seguita da $n - 1$ chiamate a TREE-SUCCESSOR esegue esattamente lo stesso attraversamento dell'albero compiuto dalla procedura INORDER-TREE-WALK. INORDER-TREE-WALK stampa prima il TREE-MINIMUM e, per definizione, il TREE-SUCCESSOR di un nodo è il nodo successivo nell'ordinamento determinato da un attraversamento simmetrico.

Questo algoritmo richiede un tempo $\Theta(n)$ perché:

- richiede un tempo $\Omega(n)$ per eseguire le n chiamate,
- attraversa ciascuno degli $n - 1$ archi dell'albero al massimo due volte, il che richiede un tempo pari $O(n)$.

Per vedere che ogni arco viene percorso al massimo due volte (una volta scendendo e una volta salendo), consideriamo l'arco tra un nodo u e uno dei suoi figli, il nodo v. Partendo dalla radice, il percorso deve attraversare l'arco (u, v) scendendo da u a v, prima di percorrerlo nuovamente verso l'alto da v a u. L'unica volta che l'albero viene percorso verso il basso è nella chiamata di TREE-MINIMUM, e l'unica volta che viene percorso verso l'alto è in TREE-SUCCESSOR quando si cerca il successore di un nodo che non ha un sottoalbero destro.
Supponiamo che v sia il figlio sinistro di u.

- Prima di stampare u, vanno stampati tutti i nodi del suo sottoalbero sinistro, che ha radice in v, garantendo l'attraversamento verso il basso dell'arco (u, v).

- Dopo che tutti i nodi del sottoalbero sinistro di u sono stati stampati, deve venire stampato u. La procedura TREE-SUCCESSOR segue un percorso che sale verso u a partire dall'elemento massimo (che non ha un sottoalbero destro) nel sottoalbero con radice in v. Questo percorso include chiaramente l'arco (u, v) e, poiché tutti i nodi del sottoalbero sinistro di u sono già stati stampati, lo spigolo l'arco (u, v) non viene più attraversato.

Supponiamo ora che v sia il figlio destro di u.

- Dopo che u è stato stampato, viene chiamata TREE-SUCCESSOR(u). Per raggiungere l'elemento minimo nel sottoalbero destro di u (la cui radice è v), l'arco (u, v) deve essere attraversato verso il basso.

- Dopo che tutti i valori nel sottoalbero destro di u sono stati stampati, viene chiamata TREE-SUCCESSOR sull'elemento massimo (di nuovo, che non ha un sottoalbero destro) nel sottoalbero con radice in v. La procedura TREE-SUCCESSOR segue un percorso verso l'alto nell'albero fino a un elemento successivo a u, poiché u è già stato stampato. In questo percorso l'arco (u, v) deve essere percorso verso l'alto e, poiché tutti i nodi del sottoalbero destro di u sono stati stampati, l'arco (u, v) non viene più percorso.

Quindi, nessuno arco viene percorso due volte nella stessa direzione.
Pertanto, l'algoritmo richiede un tempo $\Theta(n)$

Soluzione dell'Esercizio 12.3-3
Ecco la procedura.

TREE-SORT(A)
    T è un albero binario di ricerca vuoto
    for $i = 1$ **to** n
        TREE-INSERT$(T, A[i])$
    INORDER-TREE-WALK$(T.root)$

Il caso peggiore ha complessità $\Theta(n^2)$, questo accade quando le chiamate a TREE-INSERT producono una catena lineare.
Il caso migliore ha complessità $\Theta(n \lg n)$, questo accade quando le chiamate a TREE-INSERT producono un albero binario di altezza $\Theta(\lg n)$.

Soluzione dell'Esercizio 12.3-6
Rispetto alla procedura TREE-INSERT riportata nel libro, questa versione omette l'assegnazione a $z.p$, ma deve mantenere correttamente gli attributi $succ$. Il nuovo nodo z diventa figlio del nodo y. Se z diventa figlio sinistro di y, allora y deve essere il successore di z. Il codice deve anche trovare il predecessore w di y e porre z come successore di w. Basta imporre che il successore di z sia il successore di y e poi fare in modo che il successore di y sia z.
La procedura TRANSPLANT sostituisce i valori dell'attributo p con il nodo restituito dalla chiamata TREE-PARENT.

TRANSPLANT(T, u, v)
    $z =$ TREE-PARENT(T, u)
    if z == NIL
        $T.root = v$
    elseif u == $z.left$
        $z.left = v$
    else $z.right = v$

Infine, la procedura TREE-DELETE evita di riferirsi all'attributo p e inoltre fa in modo che il successore del predecessore del nodo z che viene rimosso diventi il successore di z

```
TREE-DELETE(T, z)
    x = TREE-PREDECESSOR(T, z)
    if x ≠ NIL
        x.succ = z.succ
    if z.left == NIL
        TRANSPLANT(T, z, z.right)
    elseif z.right == NIL
        TRANSPLANT(T, z, z.left)
    else y = TREE-MINIMUM(z.right)
        if y ≠ z.right
            TRANSPLANT(T, y, y.right)
            y.right = z.right
        TRANSPLANT(T, z, y)
        y.left = z.left
```

Poiché ogni chiamata a TREE-PREDECESSOR e a TREE-PARENT richiede un tempo $O(h)$, sia TREE-INSERT sia TREE-DELETE richiedono un tempo $O(h)$.

Soluzione del Problema 12-2

Per ordinare le stringhe di S, bisogna prima inserirle in un radix tree e poi fare un attraversamento anticipato per estrarle in ordine lessicografico. L'attraversamento dell'albero produce stringhe solo per i nodi che indicano l'esistenza di una stringa (cioè quelli che corrispondono ai nodi più scuri nella Figura 12.5 del libro).

Correttezza

L'attraversamento anticipato è quello corretto perché:

- la stringa di un nodo è un prefisso di tutte le stringhe dei suoi discendenti e quindi li precede nell'ordinamento (regola 2);
- i discendenti di sinistra di un nodo vengono prima dei suoi discendenti di destra perché le stringhe corrispondenti sono uguali fino a quel nodo, e nella posizione successiva le stringhe del sottoalbero di sinistra hanno 0 mentre quelle del sottoalbero di destra hanno 1 (regola 1).

Tempo Il tempo richiesto è $\Theta(n)$.

- L'inserimento richiede un tempo $\Theta(n)$, poiché l'inserimento di ogni stringa richiede un tempo proporzionale alla sua lunghezza (bisogna percorrere una porzione dell'albero la cui lunghezza è la lunghezza della stringa), e la somma di tutte le lunghezze delle stringhe è n.

- L'attraversamento anticipato richiede un tempo $O(n)$. Infatti stampa il nodo corrente e richiama se stesso ricorsivamente sui sottoalberi di sinistra e di destra, in modo da impiegare un tempo proporzionale al numero di nodi dell'albero. Il numero di nodi è al massimo 1 più la somma (cioè n) delle lunghezze delle stringhe binarie dell'albero, perché una stringa di lunghezza i corrisponde a un percorso dalla radice attraverso altri i nodi, ma un singolo nodo può essere condiviso tra molti percorsi di stringhe.

Ecco lo pseudocodice dell'attraversamento anticipato. Si assume che ogni nodo abbia due attributi *left* e *right* che puntano ai suoi figli (NIL per i figli non presenti) e un attributo booleano *string* per indicare se il nodo corrisponde a una stringa vera e propria (ad esempio, se è un nodo più scuro nella Figura 12.5 del libro). La chiamata iniziale è PREORDER-RADIX-TREE-WALK$(T.root, \varepsilon)$, dove ε indica la stringa nulla. Il simbolo $\|$ indica la concatenazione di stringhe.

```
PREORDER-RADIX-TREE-WALK(x, stringa-corrente)
    if x.string == TRUE
        stampa stringa-corrente
    if x.left ≠ NIL
        PREORDER-RADIX-TREE-WALK(x.left, stringa-corrente || 0)
    if x.right ≠ NIL
        PREORDER-RADIX-TREE-WALK(x.left, stringa-corrente || 1)
```

Soluzioni scelte per il Capitolo 13: Alberi rosso-neri

Soluzione dell'Esercizio 13.1-4
Dopo aver assorbito ogni nodo rosso nel suo padre nero, il grado di ogni nodo nero è

- 2, se entrambi i figli erano neri,
- 3, se un figlio era nero e l'altro rosso,
- 4, se entrambi i figli erano rossi.

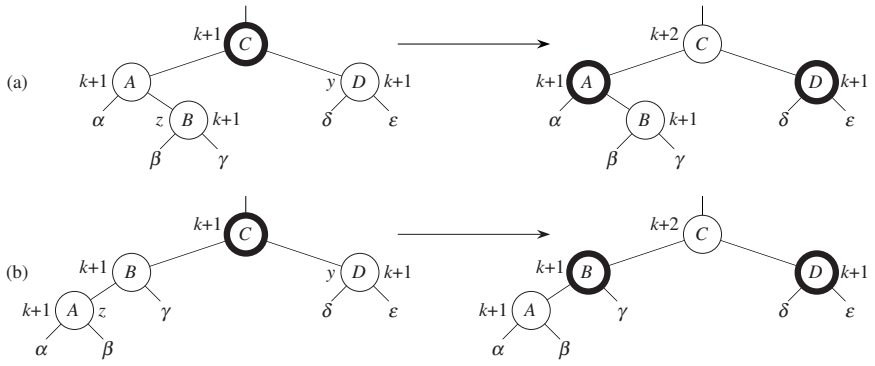
Tutte le foglie dell'albero risultante avranno la stessa profondità.

Soluzione dell'Esercizio 13.1-5
Nel percorso più lungo, almeno ogni altro nodo è nero. Nel percorso più breve, al massimo ogni nodo è nero. Poiché i due percorsi contengono lo stesso numero di nodi neri, la lunghezza del percorso più lungo è al massimo il doppio di quella del percorso più breve.
Possiamo provarlo in modo più preciso, così.
Poiché ogni percorso contiene $\mathrm{bh}(x)$ nodi neri, anche il percorso più breve da x a una foglia sua discendente ha lunghezza almeno $\mathrm{bh}(x)$. Per definizione, il percorso più lungo da x a una foglia sua discendente ha lunghezza $\mathrm{height}(x)$. Poiché il percorso più lungo ha $\mathrm{bh}(x)$ nodi neri e almeno la metà dei nodi del percorso più lungo sono neri (per la proprietà 4), $\mathrm{bh}(x) \geq \mathrm{height}(x)/2$, per cui la lunghezza del cammino più lungo è uguale a $\mathrm{height}(x)$, che è minore o uguale di $2 \cdot \mathrm{bh}(x)$, a sua volta minore o uguale della lunghezza del cammino più breve.

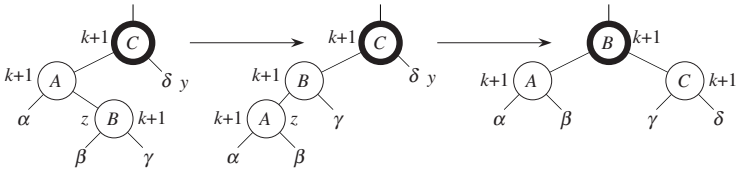
Soluzione dell'Esercizio 13.3-3

Attenzione: nelle figure seguenti, i nodi con un bordo più spesso sono neri e quelli con un bordo più fine sono rossi.

Nella Figura 13.5 del libro, i nodi A, B e D hanno sempre altezza nera $k + 1$, perché ciascuno dei loro sottoalberi ha altezza nera k e una radice nera. Il nodo C ha altezza nera $k + 1$ a sinistra (perché i suoi figli rossi hanno altezza nera $k + 1$) e altezza nera $k + 2$ a destra (perché i suoi figli neri hanno altezza nera $k + 1$).



Nella Figura 13.6 del libro, i nodi A, B e C hanno sempre altezza nera $k + 1$. A sinistra e al centro, ciascuno dei sottoalberi di A e B ha altezza nera k e una radice nera, mentre C ha uno di questi sottoalberi e un figlio rosso con altezza nera $k+1$. A destra, ciascuno dei sottoalberi di A e C ha altezza nera k e una radice nera, mentre i figli rossi di B hanno ciascuno altezza nera $k + 1$.



La proprietà 5 è conservata dalle trasformazioni. Abbiamo dimostrato in precedenza che l'altezza nera è ben definita all'interno dei sottoalberi raffigurati, quindi la proprietà 5 è conservata all'interno di questi sottoalberi. La proprietà 5 è conservata per l'albero contenente i sottoalberi mostrati, perché ogni percorso attraverso questi sottoalberi verso una foglia ha $k + 2$ nodi neri.

Soluzione del Problema 13-1

a. Quando si inserisce un nodo, tutti i nodi del percorso dalla radice al nodo aggiunto (una nuova foglia) devono cambiare, poiché la necessità di un nuovo puntatore ai figli si propaga dal nuovo nodo a tutti i suoi antenati.

Quando si elimina un nodo z, si possono verificare tre possibilità.

- Se z ha al più un figlio, allora z verrà eliminato e tutti gli antenati di z dovranno essere modificati (come nel caso dell'inserimento, la necessità di un nuovo puntatore ai figli si propaga a partire dal nodo rimosso).

- Se z ha due figli e il suo successore y è il figlio destro di z, allora si sostituisce z con y, e così tutti gli antenati di z devono essere modificati (cioè, come nel caso in cui z ha al più un figlio).

- Se z ha due figli e il suo successore y non è figlio destro di z, allora si sostituisce z con y ec$y$ con il figlio destro x di y. Poiché y e z sono antenati di x, tutti gli antenati di y devono essere modificati.

Poiché non esiste un attributo $parent$, non è necessario modificare altri nodi.

b. Quelle che seguono sono due possibili implementazioni della procedura PERSISTENT-TREE-INSERT. La prima è una versione di TREE-INSERT modificata in modo che crei dei nuovi nodi lungo il percorso dove deve essere inserito il nuovo nodo senza bisogno di usare un attributo $parent$.

```
PERSISTENT-TREE-INSERT(T, z)
    crea un nuovo albero binario di ricerca persistente T'
    T'.root = COPY-NODE(T.root)
    y = NIL
    x = T'.root
    while x ≠ NIL
        y = x
        if z.key < x.key
            x = COPY-NODE(x.left)
            y.left = x
        else x = COPY-NODE(x.right)
            y.right = x
    if y == NIL
        new-root = z
    elseif z.key < y.key
        y.left = z
    else y.right = z
    return T'
```

Questa seconda versione usa una sottoprocedura ricorsiva, PERSISTENT-SUBTREE-INSERT(r, z), che inserisce il nodo z nel sottoalbero di T radicato in r, copiando i nodi che devono esserlo, e restituendo il nodo z o la copia di r nell'albero T'.

```
PERSISTENT-TREE-INSERT(T, z)
    crea un nuovo albero binario di ricerca persistente T'
    T'.root = PERSISTENT-SUBTREE-INSERT(r, z)
    return T'

PERSISTENT-SUBTREE-INSERT(r, z)
    if r == NIL
        x = z
    else x = COPY-NODE(r)
        if z.key < r.key
            x.left = PERSISTENT-SUBTREE-INSERT(r.left, z)
        else x.right = PERSISTENT-SUBTREE-INSERT(r.right, z)
        return x
```

c. Proprio come TREE-INSERT, anche la procedura PERSISTENT-TREE-INSERT esegue una quantità costante di lavoro in ogni nodo lungo il percorso dalla radice al nuovo nodo. Poiché la lunghezza del percorso è al massimo h, richiede un tempo $O(h)$.

Poiché alloca un nuovo nodo (una quantità costante di spazio) per ogni antenato del nodo inserito, ha bisogno di uno spazio $O(h)$.

d. Se ogni nodo avesse un attributo p, allora, a causa della nuova radice, ogni nodo dell'albero dovrebbe essere copiato quando viene inserito un nuovo nodo. Per capire perché, notate che i figli della radice dovrebbero venire modificati per puntare alla nuova radice, poi i loro figli dovrebbero per puntare a loro e così via. Poiché ci sono n nodi, questo cambiamento farebbe sì che la procedura di inserimento crei $\Omega(n)$ nuovi nodi, e dunque richiederebbe un tempo $\Omega(n)$.

e. Dalle parti (a) e (c), sappiamo che l'inserimento in un albero di ricerca binario persistente di altezza h, come l'inserimento in un albero di ricerca binario ordinario, richiede nel caso peggiore un tempo $O(h)$. In un albero rosso-nero ha $h = O(\lg n)$, quindi l'inserimento in un albero rosso-nero ordinario richiede un tempo $O(\lg n)$. Dobbiamo dimostrare che se l'albero rosso-nero è persistente, l'inserimento può comunque essere effettuato in tempo $O(\lg n)$ (vedremo la cancellazione dopo). Per fare questo, dobbiamo dimostrare due cose.

- Come trovare i puntatori al padre necessari per il funzionamento della procedura in tempo $O(1)$ senza utilizzare un attributo p per il padre. Non possiamo usare un attributo di questo tipo perché un albero persistente con attributi p richiede un tempo $\Omega(n)$ per l'inserimento (come visto nella parte (d)).
- Che le modifiche aggiuntive ai nodi effettuate durante le operazioni sull'albero rosso-nero (per rotazione e ricolorazione) non causano più di $O(\lg n)$ modifiche aggiuntive ai nodi.

Ecco come trovare ogni puntatore al padre richiesto durante l'inserimento in tempo $O(1)$ senza bisogno dell'attributo p. Per fare un inserimento in un albero rosso-nero, chiamiamo la procedura RB-INSERT, che a sua volta chiama RB-INSERT-FIXUP. Apportiamo a RB-INSERT le stesse modifiche che abbiamo apportato a TREE-INSERT per la persistenza. Inoltre, mentre RB-INSERT percorre l'albero per trovare il punto in cui inserire il nuovo nodo,

dobbiamo mantenere in una pila i nodi che attraversa e passare questa pila a RB-INSERT-FIXUP. La procedura RB-INSERT-FIXUP ha bisogno dei puntatori al padre per risalire lo stesso percorso, e in ogni momento ha bisogno dei puntatori al padre solo per trovare il padre e il nonno del nodo su cui sta lavorando. Man mano che RB-INSERT-FIXUP sale nella pila dei padri, ha bisogno solo dei puntatori al padre che si trovano in posizioni note a distanza costante nella pila. In questo modo, le informazioni sui padri possono essere trovate in tempo $O(1)$, proprio come se fossero memorizzate in un attributo p.

Una rotazione e una ricolorazione modificano i nodi in questo modo.

- RB-INSERT-FIXUP esegue al massimo due rotazioni e ogni rotazione aggiorna i puntatori dei figli in tre nodi (il nodo su cui avviene la rotazione, suo padre e uno dei figli del nodo su cui avviene la rotazione). Pertanto, la rotazione modifica direttamente al massimo sei nodi durante RB-INSERT-FIXUP. In un albero persistente, tutti gli antenati di un nodo modificato vengono copiati, quindi le rotazioni di RB-INSERT-FIXUP impiegano un tempo $O(\lg n)$ per modificare i nodi a causa della rotazione (in realtà, in questo caso i nodi modificati condividono un unico percorso di antenati di lunghezza $O(\lg n)$).
- RB-INSERT-FIXUP ricolora alcuni degli antenati del nodo inserito, che vengono comunque modificati nell'inserimento persistente, e alcuni figli degli antenati (gli "zii" a cui si fa riferimento nella descrizione dell'algoritmo). Ci sono $O(\lg n)$ antenati, quindi $O(\lg n)$ cambi di colore degli zii. La ricolorazione degli zii non causa alcuna modifica aggiuntiva dei nodi a causa della persistenza, perché gli antenati degli zii sono gli stessi nodi (cioè gli antenati del nodo inserito) che vengono modificati comunque a causa della persistenza. Pertanto, la ricolorazione non influisce sul tempo di esecuzione che è $O(\lg n)$, anche con la persistenza.

Possiamo dimostrare in modo analogo che anche la cancellazione in un albero persistente richiede il tempo $O(h)$ nel caso peggiore.

- Abbiamo già visto nella parte (a) che vengono modificati $O(h)$ nodi.
- Potremmo scrivere una procedura RB-DELETE persistente che viene eseguita in tempo $O(h)$, modificando quella non persistente in modo analogo a quanto fatto per l'inserimento. Ma per farlo senza usare i puntatori al padre, la procedura deve percorrere l'albero fino al nodo più profondo

che viene modificato e costruire una pila dei padri come discusso sopra per l'inserimento. Questo si basa sul fatto che le chiavi sono tutte diverse.

Il problema di dimostrare che la cancellazione richiede un tempo che è solo $O(\lg n)$ in un albero persistente rosso-nero è lo stesso dell'inserimento.

- Come per l'inserimento, possiamo dimostrare che i padri necessari a RB-DELETE-FIXUP possono essere trovati in tempo $O(1)$ (usando la stessa tecnica dell'inserimento).
- Inoltre, RB-DELETE-FIXUP esegue al massimo tre rotazioni, che, come discusso sopra per l'inserimento, richiedono un tempo $O(\lg n)$ per cambiare i nodi a causa della persistenza. Effettua anche $O(\lg n)$ ricolorazioni, che (come per l'inserimento) richiedono un tempo che è solo $O(\lg n)$ per cambiare gli antenati a causa della persistenza, perché il numero di nodi copiati è $O(\lg n)$.

60

Soluzioni scelte per il Capitolo 14: Programmazione dinamica

Soluzione dell'Esercizio 14.2-5

Ogni volta che viene eseguito il ciclo indicizzato da l, quello indicizzato da i viene eseguito $n - l + 1$ volte. Ogni volta che viene eseguito il ciclo indicizzato da i, quello indicizzato da k viene eseguito $j - i = l - 1$ volte, e ogni volta fa riferimento a m due volte. Quindi il numero totale di volte a cui si fa riferimento ad un elemento di m mentre se ne sta calcolando un altro è $\sum_{l=2}^{n} 2(n-l+1)(l-1)$. Di conseguenza

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=1}^{n} R(i,j) &= \sum_{l=2}^{n} 2(n-l+1)(l-1) \\
&= 2\sum_{l=1}^{n-1} 2(n-l)l \\
&= 2\sum_{l=1}^{n-1} nl - 2\sum_{l=1}^{n-1} l^2 \\
&= 2\frac{n(n-1)n}{2} - 2\frac{(n-1)n(2n-1)}{6} \\
&= n^3 - n^2 - \frac{2n^3 - 3n^2 + n}{3} \\
&= \frac{n^3 - n}{3} .
\end{aligned}$$

Soluzione dell'Esercizio 14.3-1

L'esecuzione di RECURSIVE-MATRIX-CHAIN è asintoticamente più efficiente che enumerare tutti gli schemi di parentesizzazione del prodotto e calcolare il numero di moltiplicazioni tra scalari per ciascuno di essi.

Vediamo come nei due approcci vengono trattati i sottoproblemi.

- Per ogni possibile punto in cui suddividere la catena di matrici, l'approccio enumerativo trova tutti gli schemi di parentesizzazione per la metà di sinistra e per la metà destra ed esamina tutte le possibili combinazioni della metà sinistra con la metà destra. La quantità di lavoro richiesta per esaminare ogni combinazione dei sottoproblemi di sinistra e di destra è quindi il prodotto del numero di modi per calcolare la metà sinistra e il numero di modi per calcolare la metà destra.
- Per ogni possibile punto in cui suddividere la catena di matrici, RECURSIVE-MATRIX-CHAIN trova il modo migliore per parentesizzare la metà sinistra, trova il modo migliore per parentesizzare la metà destra e combina solo questi due risultati. Pertanto, la quantità di lavoro richiesta per combinare i risultati dei sottoproblemi della metà sinistra e della metà destra è $O(1)$.

Nel Paragrafo 14.2 abbiamo detto che il tempo di esecuzione nel caso dell'enumerazione è $\Omega(4^n/n^{3/2})$. Qui dimostreremo che il tempo di esecuzione di RECURSIVE-MATRIX-CHAIN è $O(n3^{n-1})$.
Per ottenere un limite superiore sul tempo di esecuzione di RECURSIVE-MATRIX-CHAIN, utilizziamo lo stesso approccio usato nel Paragrafo 14.2 per ottenere un limite inferiore: trovare una disequazione in forma di ricorrenza e risolverla per sostituzione. Per il limite inferiore, il libro ipotizza che l'esecuzione delle righe 1-2 e 6-7 richieda almeno un tempo unitario. Per la ricorrenza del limite superiore, assumeremo che queste coppie di righe richiedano ciascuna al massimo un tempo costante c. Abbiamo quindi la ricorrenza

$$T(n) \leq \begin{cases} c & \text{se } n = 1 \,, \\ c + \displaystyle\sum_{k=1}^{n-1} (T(k) + T(n-k) + c) & \text{se } n \geq 2 \,. \end{cases}$$

Questa non è altro che la ricorrenza del libro per il limite inferiore, con la sola differenza che c'è c al posto di 1, quindi la possiamo scrivere nella forma

$$T(n) \leq 2 \sum_{i=1}^{n-1} T(i) + cn \,.$$

Proviamo ora che $T(n) = O(n3^{n-1})$ usando il metodo di sostituzione (osservate che ogni limite superiore per $T(n)$ che sia $o(4^n/n^{3/2})$ andrebbe bene, potreste

provare a dimostrarne da soli un altro che abbia una forma più semplice, come $T(n) = O(3, 5^n)$, ma $T(n) = O(n3^{n-1})$ è un limite più stretto). Più precisamente, proviamo che $T(n) \leq cn3^{n-1}$ per tutti gli $n \geq 1$. Il caso base è facile, visto che $T(1) \leq c = c \cdot 1 \cdot 3^{1-1}$. Assumendo l'ipotesi induttiva per $n-1$ con $n \geq 2$, abbiamo

$$\begin{aligned}
T(n) &\leq 2\sum_{i=1}^{n-1} T(i) + cn \\
&\leq 2\sum_{i=1}^{n-1} ci3^{i-1} + cn \\
&= c \cdot \left(2\sum_{i=1}^{n-1} i3^{i-1} + n\right) \\
&= c \cdot \left(2\left(\frac{n3^{n-1}}{3-1} + \frac{1-3^n}{(3-1)^2}\right) + n\right) \qquad \text{(vedi sotto)} \\
&= cn3^{n-1} + c \cdot \left(\frac{1-3^n}{2} + n\right) \\
&= cn3^{n-1} + \frac{c}{2}(2n + 1 - 3^n) \\
&\leq cn3^{n-1}
\end{aligned}$$

per ogni $c > 0$ e $n \geq 1$.
L'esecuzione di RECURSIVE-MATRIX-CHAIN richiede un tempo $O(n3^{n-1})$, mentre l'enumerazione di tutti gli schemi di parentesizzazione richiede un tempo $\Omega(4^n/n^{3/2})$, dunque RECURSIVE-MATRIX-CHAIN è molto più efficiente dell'enumerazione completa.
Per concludere, osservate che nel metodo di sostituzione abbiamo utilizzato l'uguaglianza

$$\sum_{i=1}^{n-1} ix^{i-1} = \frac{nx^{n-1}}{x-1} + \frac{1-x^n}{(x-1)^2} \, .$$

Questa uguaglianza segue dall'equazione (A.6) facendo la derivata dei due membri. Infatti, se

$$f(x) = \sum_{i=1}^{n-1} x^i = \frac{x^n - 1}{x-1} - 1 \, ,$$

allora

$$\sum_{i=1}^{n-1} ix^{i-1} = f'(x) = \frac{nx^{n-1}}{x-1} + \frac{1-x^n}{(x-1)^2} .$$

Soluzione dell'Esercizio 14.4-4

Quando si calcola una particolare riga della tabella c, non sono necessarie altre righe oltre alla precedente. Pertanto, è necessario tenere in memoria solo due righe, e dunque $2n$ elementi, alla volta (osservate che ogni riga di c ha in realtà $n+1$ elementi, ma non è necessario memorizzare la colonna degli 0; possiamo invece far sì che il programma "sappia" che quegli elementi sono 0). Con questa idea, abbiamo bisogno solo di $2 \cdot \min\{m, n\}$ elementi se chiamiamo sempre LCS-LENGHT con la sequenza più corta come argomento Y.
Possiamo quindi fare a meno della tabella c così.

- Utilizziamo due array di lunghezza $\min\{m, n\}$ che chiamiamo *riga-precedente* e *riga-corrente*, che conterranno le righe di c che ci servono.
- Inizializziamo *riga-precedente* a 0 e calcoliamo i valori da inserire in *riga-corrente* da sinistra a destra.
- Quando *riga-corrente* è piena, se ci sono ancora righe da calcolare, copiamo *riga-corrente* in *riga-precedente* e calcoliamo la nuova *riga-corrente*.

In realtà, durante il calcolo sono necessari solo poco più degli elementi di una riga di c, per la precisione $\min\{m, n\} + 1$. Gli unici elementi della tabella che servono quando è il momento di calcolare $c[i, j]$ sono $c[i, k]$ per $k \leq j - 1$ (cioè quelli che lo precedono nella riga corrente, che serviranno per calcolare la riga successiva) e $c[i-1, k]$ per $k \geq j - 1$ (cioè quelli nella riga precedente che servono ancora per calcolare il resto della riga corrente). Si tratta di un elemento per ogni k da 1 a $\min\{m, n\}$, tranne per il fatto che ci sono due elementi con $k = j - 1$, da cui l'elemento aggiuntivo necessario oltre a quelli che compongono un'intera riga.
Possiamo quindi fare a meno della tabella c così.

- Usiamo un array a di lunghezza $\min\{m, n\} + 1$ per contenere gli elementi di c che ci servono. Nel momento in cui deve essere calcolato $c[i, j]$, l'array a contiene questi elementi:
  - $a[k] = c[i, k]$ per $1 \leq k \leq j - 1$ (cioè gli elementi che precedono nella riga corrente),

  - $a[k] = c[i-1, k]$ per $k \geq j-1$ (cioè elementi nella riga precedenti),
  - $a[0] = c[i, j-1]$ (cioè l'elemento appena calcolato, che non può essere inserito nel posto giusto in a senza cancellare l'elemento $c[i-1, j-1]$, che però ci serve ancora).

- Inizializziamo a a 0 e calcoliamo gli elementi da sinistra a destra
  - Osserviamo che i tre valori che servono per calcolare $c[i, j]$ per $j > i$ si trovano in $a[0] = ci, j-1$, $a[j-1] = c[i-1, j-1]$ e $a[j] = c[i-1, j]$.
  - Quando $c[i, j]$ è stato calcolato, spostiamo $a[0]$ (cioè $c[i, j-1]$) al posto giusto, cioè in $a[j-1]$, e mettiamo $c[i, j]$ in $a[0]$.

Soluzione del Problema 14-4

Iniziamo definendo alcune quantità in modo da poter enunciare il problema in modo più uniforme. I casi speciali relativi all'ultima riga e al fatto se una sequenza di parole può stare in una riga saranno gestiti in queste definizioni, in modo da non dovercene occupare quando definiremo la nostra strategia generale.

- Definiamo $extras[i, j] = M - j + 1 - \sum_{k=i}^{j} l_k$, ovvero il numero di spazi in più alla fine di una riga contenente le parole da i a j. Notate che $extras$ può assumere valori negativi.

- Ora definiamo il costo di avere una riga contenente le parole da i a j nella somma che vogliamo minimizzare:

$$lc[i, j] = \begin{cases} \infty & \text{se } extras[i, j] < 0 \text{ (cioè se le parole } i, \ldots, j \text{ non ci stanno)}, \\ 0 & \text{se } j = n \text{ e } extras[i, j] \geq 0 \text{ (l'ultima riga costa 0)}, \\ (extras[i, j])^3 & \text{altrimenti}. \end{cases}$$

  Rendendo infinito il costo della riga quando le parole non vi si adattano, impediamo che tale disposizione faccia parte di una somma minima; rendendo 0 il costo per l'ultima riga (se le parole si adattano), impediamo che la disposizione dell'ultima riga influisca sulla somma da minimizzare.

Vogliamo minimizzare la somma di lc fatta su tutte le righe del paragrafo.
I nostri sottoproblemi riguardano come disporre in modo ottimale le parole $1, \ldots, j$, per j da 1 a n.
Consideriamo una disposizione ottimale delle parole $1, \ldots, j$. Supponiamo di sapere che l'ultima riga, che termina con la parola j, inizia con la parola i. Le

righe precedenti, quindi, contengono le parole $1, \ldots, i-1$. Più precisamente, devono contenere una disposizione ottimale delle parole $1, \ldots, i-1$ (si applica il solito ragionamento del tipo taglia e incolla).

Sia $c[j]$ il costo di una disposizione ottimale delle parole $1, \ldots, j$. Se sappiamo che l'ultima riga contiene le parole $i, \ldots, j$, allora $c[j] = c[i-1] + lc[i, j]$. Come caso base, quando calcoliamo $c[1]$, ci serve $c[0]$. Se poniamo $c[0] = 0$, allora $c[1] = lc[1, 1]$, che è quello che vogliamo.

Ma naturalmente dobbiamo capire con quale parola inizia l'ultima riga per il sottoproblema relativo alle parole $1, \ldots, j$. Quindi proviamo tutte le possibili parole i e scegliamo quella che dà il costo minore. In questo caso, i va da 1 a j. Pertanto, possiamo definire $c[j]$ in modo ricorsivo così:

$$c[j] = \begin{cases} 0 & \text{se } j = 0\,, \\ \min\{c[i-1] + lc[i, j] : 1 \le i \le j\} & \text{se } j > 0\,. \end{cases}$$

Notate che il modo in cui abbiamo definito lc garantisce che

- tutte le scelte fatte rientrano nella riga (poiché una disposizione con $lc = \infty$ non può essere scelta come minima), e

- il costo di mettere le parole $i, \ldots, j$ sull'ultima riga non può essere 0, a meno che questa non sia davvero l'ultima riga del paragrafo ($j = n$) o che le parole $i, \ldots, j$ riempiano l'intera riga.

Possiamo calcolare una tabella dei valori di c da sinistra a destra, poiché ogni valore dipende solo dai valori precedenti.

Per tenere traccia di quali parole vanno su quali righe, possiamo tenere in parallelo una tabella p che indica la provenienza di ciascun valore di c. Quando calcoliamo $c[j]$, se $c[j]$ richiede il valore di $c[k-1]$, poniamo $p[j] = k$. Quindi, dopo che è stato calcolato $c[n]$, possiamo seguire i puntatori per vedere dove interrompere le righe. L'ultima riga inizia dalla parola $p[p[n]]$ e va fino alla parola $p[n] - 1$, e così via.

Ecco come costruire le tabelle in pseudocodice.

```
PRINT-NEATLY(l, n, M)
    crea delle nuove tabelle extras[1:n, 1:n], lc[1:n, 1:n], c[0:n] e p[1:n]
    // Calcola extras[i, j] per 1 ≤ i ≤ j ≤ n.
    for i = 1 to n
        extras[i, j] = M − l_i
        for j = i + 1 to n
            extras[i, j] = extras[i, j − 1] − l_i − 1
    // Calcola lc[i, j] per 1 ≤ i ≤ j ≤ n.
    for i = 1 to n
        for j = i to n
            if extras[i, j] < 0
                lc[i, j] = ∞
            elseif i == n e extras[i, j] ≥ 0
                lc[i, j] = 0
            else lc[i, j] = (extras[i, j])^3
    // Calcola c[j] per 0 ≤ j ≤ n e p[j] per 1 ≤ j ≤ n.
    c[0] = 0
    for j = 1 to n
        c[j] = ∞
        for i = 1 to j
            if c[i − 1] + lc[i, j] < c[j]
                c[j] = c[i − 1] + lc[i, j]
                p[j] = i
    return c e p
```

É abbastanza chiaro che la complessità temporale e quella spaziale sono entrambe $\Theta(n^2)$.

In realtà, possiamo fare un po' meglio: possiamo ridurre sia il tempo che lo spazio a $\Theta(nM)$. L'osservazione chiave è che su una riga possono stare al massimo $\lceil M/2 \rceil$ parole (infatti ogni parola è lunga almeno un carattere e c'è uno spazio tra le parole). Poiché una riga con le parole $i, \ldots, j$ contiene $j - i + 1$ parole, se $j - i + 1 > \lceil M/2 \rceil$ allora sappiamo che $lc[i, j] = \infty$. Quindi dobbiamo calcolare e memorizzare solo $extras[i, j]$ e $lc[i, j]$ per $j - i + 1 \leq \lceil M/2 \rceil$. Inoltre possiamo anche fare in modo che il contatore dei cicli **for** più interni che servono per calcolare c e lc vada da $\max\{1, j - \lceil M/2 \rceil + 1\}$ a j.

Possiamo ridurre ulteriormente lo spazio a $\Theta(n)$. Per farlo, non memorizziamo le tabelle lc ed $extras$ e invece calcoliamo il valore di $lc[i, j]$ quando richiesto nell'ultimo ciclo. L'idea è che possiamo calcolare $lc[i, j]$ in tempo $O(1)$ se conosciamo

il valore di $extras[i, j]$. E se cerchiamo il valore minimo in ordine *decrescente* rispetto a i, possiamo calcolarlo come $extras[i, j] = extras[i + i, j] - l_i - 1$ (inizialmente $extras[i, j] = M - l_i$) Questo miglioramento riduce lo spazio a $\Theta(n)$, dal momento che ora le uniche tabelle memorizzate sono c e p.
Ecco come stampare l'output. La chiamata PRINT-LINES(p, j) stampa tutte le parole dalla parola 1 alla parola j.

```
PRINT-LINES(p, j)
    if j > 0
        i = p[j]
        PRINT-LINES(p, i − 1)
        stampa la riga contenente le parole da i a j
          con uno spazio tra ogni parola
```

La chiamata iniziale è PRINT-LINES(p, n). Poiché il valore di j diminuisce a ogni chiamata ricorsiva, PRINT-LINES impiega un tempo totale $O(n + k)$ per stampare tutte le n parole, dove k è la lunghezza totale di tutte le parole (notate che, poiché ogni parola contiene almeno un carattere, anche contando gli spazi e i margini di riga come caratteri stampati, il numero totale di caratteri stampati è al massimo $2k$).

Soluzioni scelte per il Capitolo 15: Algoritmi avidi

Soluzione dell'Esercizio 15.1-4

Sia S l'insieme delle n attività.

La soluzione "ovvia", che consiste nell'utilizzare la procedura GREEDY-ACTIVITY-SELECTOR per trovare un insieme massimale S_1 di attività di S compatibili la prima aula, poi di nuovo per trovare un insieme massimale S_2 per la seconda aula, e così via fino a quando tutte le attività sono assegnate, richiede un tempo $\Theta(n^2)$ nel caso peggiore. Inoltre, può produrre un risultato che utilizza più aule del necessario. Consideriamo quattro attività con tempi di inizio e di fine $[1, 4)$, $[2, 5)$, $[6, 7)$ e $[4, 8)$. La procedura GREEDY-ACTIVITY-SELECTOR sceglierebbe le attività con tempi di inizio e di fine rappresentati dagli intervalli $[1, 4)$ e $[6, 7)$ intervalli per la prima aula, e poi ciascuna delle attività rimanenti avrebbe bisogno di un'aula tutta per sé, per un totale di tre aule utilizzate. Una soluzione ottimale metterebbe le attività rappresentate dagli intervalli $[1, 4)$ e $[4, 8)$ in un'aula e quelle rappresentate dagli intervalli $[2, 5)$ e $[6, 7)$ in un'altra, per un totale di sole due aule utilizzate.

Esiste tuttavia un algoritmo corretto, la cui complessità temporale asintotica è data soltanto dal tempo necessario per ordinare le attività in base ai loro tempi di inizio e di fine, e quindi che richiede in generale un tempo $O(n \lg n)$, o addirittura $O(n)$ se possiamo applicare le considerazioni del Capitolo 8, ad esempio se i tempi sono numeri interi piccoli.

L'idea è quella di scorrere le attività in ordine di tempo di inizio, assegnando ciascuna a una qualunque delle sale disponibili in quel dato momento. Per far questo, dobbiamo scorrere l'insieme formato da tutti gli eventi che corrispondono al fatto che un'attività sta iniziando oppure che sta finendo, ordinato in ordine crescente di tempo. Dobbiamo poi mantenere due elenchi di aule: quelle che sono occupate al tempo corrente dell'evento t (perché è stata assegnata loro un'attività i che è iniziata al tempo $s_i \leq t$ ma che non finirà prima di $f_i > t$) e le aule

che sono libere al tempo t (come abbiamo fatto per il problema della selezione di attività nel Paragrafo 15.1, assumiamo che gli intervalli di tempo delle attività siano semiaperti, cioè che se $s_i \geq f_j$, allora le attività i e j sono compatibili). Quando t è il tempo di inizio di un'attività, assegniamo l'attività a una sala libera e spostiamo la sala dall'elenco di quelle libere e la inseriamo in quello di quelle occupate. Quando t è il tempo di fine di un'attività, spostiamo la sala dell'attività dall'elenco di quelle occupate a quello di quelle libere (l'attività è sicuramente in una sala, perché gli orari degli eventi vengono elaborati in ordine e l'attività deve essere iniziata prima del suo tempo di fine t, quindi deve essere stata assegnata a una sala).
Per evitare di utilizzare più sale del necessario, scegliamo sempre una sala a cui è già stata assegnata un'attività, se possibile, prima di scegliere una sala mai utilizzata (questo può essere fatto considerando sempre l'inizio dell'elenco delle sale libere, cioè inserendo le sale liberate in cima all'elenco e togliendo una sala che viene occupata dalla cima dell'elenco, in modo che una nuova sala non si trovi in cima e venga scelta quando ci sono sale disponibili che sono già state utilizzate). Questo garantisce che l'algoritmo utilizzi il minor numero possibile di sale. Infatti supponiamo che occupi $m \geq n$ sale e che i sia la prima attività programmata nell'aula m, allora questo vuol dire che le prime $m-1$ sale erano occupate al tempo s_i. Quindi in quel momento c'erano m attività in corso contemporaneamente. Pertanto, qualsiasi pianificazione deve utilizzare almeno m aule, e quella prodotta dall'algoritmo è ottimale.
Il tempo di esecuzione è dato da

- quello necessario a ordinare i $2n$ tempi di inizio e di fine (nell'ordinamento, un tempo di fine attività deve precedere il tempo di inizio di un'attività che inizia in quello stesso momento), questo può essere fatto in tempo $O(n \lg n)$ in generale, o addirittura in tempo $O(n)$ se possiamo applicare le considerazioni del Capitolo 8, per esempio se i tempi sono numeri interi piccoli;

- più quello necessario a elaborare i $2n$ eventi, questo richiede un tempo $O(1)$ per ciascuno, perché si deve spostare una sala da una lista ad un'altra ed eventualmente associarvi un'attività, e dunque $O(n)$ in totale.

In conclusione, il tempo totale per questo algoritmo è $O(n+\text{costo dell'ordinamento}) = O(n \lg n)$.

Soluzione dell'Esercizio 15.2-2
La soluzione si basa sull'osservazione fatta relativamente alla sottostruttura ottima

riportata nel libro. Se i è l'ultimo articolo, nell'ordine che viene fornito, contenuto in una soluzione ottimale S per W chili e n articoli, allora $S' = S - \{i\}$ deve essere una soluzione ottimale per $W - w_i$ chili e articoli $1, \ldots, i-1$. Quindi il valore della soluzione S è v_i più il valore della soluzione S' del sottoproblema. Se $c[i, w]$ indica il valore di una soluzione per gli articoli $1, \ldots, i$ e peso massimo w, allora possiamo esprimere l'osservazione precedente così:

$$c[i,w] = \begin{cases} 0 & \text{se } i = 0 \text{ o } w = 0\ , \\ c[i-1,w] & \text{se } w_i > w\ , \\ \max\{v_i + c[i-1, w-w_i], c[i-1,w]\} & \text{se } i > 0 \text{ e } w \geq w_i\ . \end{cases}$$

L'ultimo caso dice che il valore di una soluzione per i articoli o include l'articolo i, nel qual caso è il valore è v_i più quello di una soluzione del sottoproblema per $i-1$ articoli e peso ridotto di w_i, o non include l'articolo i, nel qual caso il valore è quello di una soluzione del sottoproblema per $i-1$ articoli con lo stesso peso. In altre parole, se il ladro sceglie l'articolo i, allora sommiamo il valore v_i e il ladro può scegliere tra gli elementi $1, \ldots, i-1$ fino al limite di peso $w - w_i$, guadagnando un valore aggiuntivo $c[i-1, w-w_i]$. Se invece il ladro decide di non prendere l'articolo i, può scegliere tra gli oggetti $1, \ldots, i-1$ fino al limite di peso w, con un guadagno $c[i-1, w]$. Ovviamente il ladro deve fare la scelta più vantaggiosa tra queste due.

L'algoritmo prende come input il peso massimo W, il numero n di articoli e le successioni $v = \langle v_1, v_2, \ldots, v_n \rangle$ e $w = \langle w_1, w_2, \ldots, w_n \rangle$. Memorizza i valori $c[i, j]$ in una tabella $c[0:n, 0:W]$ i cui elementi sono calcolati per righe (cioè, la prima riga di c viene calcolata da sinistra a destra, poi si passa alla seconda e così via). Alla fine del calcolo, c contiene il valore massimo che il ladro può ottenere.

DYNAMIC-0-1-KNAPSACK(v, w, n, W)
    crea una nuova tabella $c[0:n, 0:W]$
    for $w = 0$ **to** W
        $c[0, w] =$
    for $i = 1$ **to** n
        $c[i, 0] = 0$
        for $w = 1$ **to** W
            if $w_i \leq w$ e $v_i + c[i-1, w-w_i] > c[i-1, w]$
                $c[i, w] = v_i + c[i-1, w-w_i]$
            else $c[i, w] = c[i-1, w]$

Possiamo usare la tabella c per capire quali articoli da prendere partendo da $c[n, W]$ e risalendo ai valori ottimali. Se $c[i, w] = c[i-1, w]$, allora l'articolo i non fa parte

della soluzione e continuiamo con $c[i-1, w]$, altrimenti i fa parte della soluzione(e lo teniamo da parte) e continuiamo con $c[i-1, w-w_i]$.
Il nostro algoritmo richiede in totale un tempo $\Theta(nW)$, infatti

- serve un tempo $\Theta(nW)$ per riempire la tabella c, poiché ci sono $(n+1)\cdot(W+1)$ elementi e ciascuno richiede un tempo $\Theta(1)$ per essere calcolato, e
- serve un ulteriore tempo $O(n)$ per ricostruire la soluzione, poiché si parte dalla riga n e si sale di una riga ad ogni passo.

Soluzione dell'Esercizio 15.2-7
Basta ordinare A e B in ordine decrescente.
Ecco perché questo metodo produce una soluzione ottimale. Consideriamo due indici i e j tali che $i < j$, e i termini corrispondenti $a_i^{b_i}$ e $a_j^{b_j}$. Vogliamo dimostrare che non si fa peggio prendendo questi due termini rispetto a prendere $a_i^{b_j}$ e $a_j^{b_i}$, cioè che $a_i^{b_i} a_j^{b_j} \geq a_i^{b_j} a_j^{b_i}$. Dal momento che A e B sono ordinati in ordine decrescente e che $i < j$, abbiamo $a_i \geq a_j$ e $b_i \geq b_j$. Poiché a_i e a_j sono positivi e $b_i - b_j$ è non negativo, abbiamo $a_i^{b_i - b_j} \geq a_j^{b_i - b_j}$. Moltiplicando entrambi i membri per $a_i^{b_j} a_j^{b_j}$ otteniamo proprio $a_i^{b_i} a_j^{b_j} \geq a_i^{b_j} a_j^{b_i}$.
Poiché l'ordine dei prodotti non conta, si possono ordinare A e B anche in ordine crescente.

Soluzioni scelte per il Capitolo 16: Analisi ammortizzata

Soluzione dell'Esercizio 16.1-3
Sia c_i il costo della i-esima operazione, cioè c_i è i se i è una potenza esatta di 2, 1 altrimenti.
Allora il costo delle operazioni da c_1 a c_n è

$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lg n} 2^j = n + (2n - 1) < 3n \ .$$

Quindi il costo medio per operazione (cioè il costo totale diviso per il numero di operazioni) è minore di 3 e, per l'analisi aggregata, il costo ammortizzato per operazione è $O(1)$.

Soluzione dell'Esercizio 16.2-2
Ripetiamo l'Esercizi 16.1-3 utilizzando il metodo degli accantonamenti. Come in quel caso, c_i è il costo della i-esima operazione, cioè c_i è i se i è una potenza esatta di 2, 1 altrimenti.
Ad ogni operazione assegniamo un costo ammortizzato $\hat{c}_i$ di 3 euro:

- Se i non è una potenza esatta di 2, paghiamo un euro e ne accantoniamo 2 come credito,
- se i pè una potenza esatta di 2, paghiamo i euro usando il credito accantonato.

Otteniamo una tabella dei costi di questo tipo:

operazione	costo ammortizzato	costo reale	credito restante
1	3	1	2
2	3	2	3
3	3	1	5
4	3	4	4
5	3	1	6
6	3	1	8
7	3	1	10
8	3	8	5
9	3	1	7
10	3	1	9
...	...	...	...

Poiché il costo ammortizzato è di 3 euro per operazione, abbiamo $\sum_{i=1}^{n} \hat{c}_i = 3n$. Sappiamo dall'Esercizio 16.1-3 che $\sum_{i=1}^{n} c_i < 3n$.
Poiché $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0$, cioè il credito disponibile non è mai negativo, e il costo ammortizzato per operazione è $O(1)$, ne segue che il costo totale delle n operazioni è $O(n)$.

Soluzione dell'Esercizio 16.2-3

Introduciamo un nuovo campo $A.max$ per memorizzare la posizione del bit 1 più significativo. Inizialmente $A.max = -1$, poiché il bit meno significativo è 0 e non ci sono 1 in A. Il valore di $A.max$ viene aggiornato in modo appropriato quando il contatore viene incrementato o azzerato, e questo valore limita la porzione del contatore A che va analizzata per azzerarlo. Poiché teniamo sotto controllo il costo dell'operazione RESET in questo modo, possiamo limitarlo con un importo che può essere pagato prendendo del credito accantonato dalle precedenti operazioni INCREMENT.

DYNAMIC-0-1-KNAPSACK(v, w, n, W)
    crea una nuova tabella $c[0:n, 0:W]$
    for $w = 0$ **to** W
        $c[0, w] =$
    for $i = 1$ **to** n
        $c[i, 0] = 0$
        for $w = 1$ **to** W
            if $w_i \leq w$ e $v_i + c[i-1, w-w_i] > c[i-1, w]$
                $c[i, w] = v_i + c[i-1, w-w_i]$
            else $c[i, w] = c[i-1, w]$

Come nell'analisi del contatore che abbiamo svolto nel libro, supponiamo che il costo per cambiare il valore di un bit sia di un euro. Inoltre, supponiamo che l'aggiornamento di $A.max$ costi un euro.
Assegnare e resettare i bit con INCREMENT funziona esattamente come per il contatore originale del libro: paghiamo un euro per assegnare 1 a un bit, mettiamo un euro sul bit che viene assegnato come credito e usiamo questo credito per riportare a 0 il bit durante un incremento.
Inoltre, paghiamo un euro per aggiornare max, e se max aumenta, e mettiamo un altro euro di credito sul nuovo bit 1 più significativo (se max non aumenta, stiamo sprecando un euro, ma non ci servirà). Poiché RESET manipola solo i bit fino alla posizione $A.max$, e poiché ogni bit fino a quella posizione deve essere stato il bit 1 più significativo in un qualche momento prima che il bit 1 più significativo si trovasse in posizione$A.max$, ogni bit analizzato da RESET ha un euro di credito su di esso. Quindi l'azzeramento dei bit di A da parte di RESET può essere completamente pagato dal credito memorizzato sui bit. Ci serve solo un euro per pagare l'azzeramento di max.
È quindi sufficiente pagare 4 euro per ogni operazione INCREMENT e un euro per ogni operazione RESET, e di conseguenza una sequenza di n operazioni INCREMENT e RESET richiede un tempo $O(n)$.

Soluzioni scelte per il Capitolo 17: Arricchire le strutture dati

Soluzione dell'Esercizio 17.1-7

Sia $A[1:n]$ un array formato da n elementi diversi.

Un modo per contare le inversioni consiste nel mantenere un contatore inizialmente a 0 e per ogni elemento incrementare il contatore del numero di elementi maggiori di quello considerato che lo precedono nell'array, dunque il numero delle inversioni è

$$\sum_{j=1}^{n} |\operatorname{Inv}(j)| ,$$

dove $\operatorname{Inv}(j) = \{i : i < j \text{ e } A[i] > A[j]\}$.

Osservate che $|\operatorname{Inv}(j)|$ si riferisce al rango di $A[j]$ nel sottoarray $A[1:j]$, poiché gli elementi in $\operatorname{Inv}(j)$ sono il motivo per cui $A[j]$ non si trova dove dovrebbe secondo il suo rango. Sia $r(j)$ il rango di $A[j]$ in $A[1:j]$, allora $j = r(j) + |\operatorname{Inv}(j)$, e possiamo calcolare

$$|\operatorname{Inv}(j) = j - r(j)$$

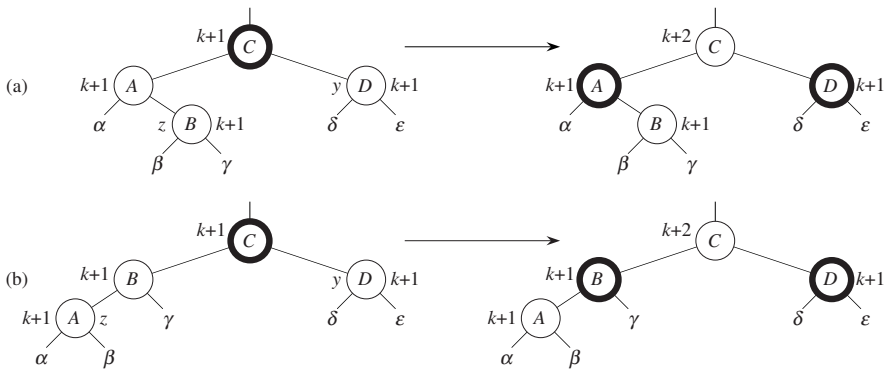
inserendo $A[1], \ldots, A[n]$ in un albero per statistiche d'ordine è usando OS-RANK per cercare il rango di ogni $A[j]$ nell'albero immediatamente dopo il suo inserimento in esso (in tal modo OS-RANK restituisce proprio $r(j)$).

Un inserimento e una chiamata di OS-RANK richiedono ciascuno un tempo $O(\lg n)$, e visto che gli elementi sono in tutto n, il tempo totale è $O(n \lg n)$.

Soluzione dell'Esercizio 17.2-2

Sì, è possibile mantenere le altezze nere come attributi nei nodi di un albero rosso-nero senza influire sulle prestazioni asintotiche delle operazioni dell'albero rosso-nero. Possiamo applicare il Teorema 17.1, perché l'altezza nera di un nodo può

essere calcolata a partire dalle informazioni presenti nel nodo e nei suoi due figli. In realtà, l'altezza nera può essere calcolata a partire dalle informazioni contenute in un solo figlio: l'altezza nera di un nodo è l'altezza nera di un figlio rosso o l'altezza nera di un figlio nero più uno. Il secondo figlio non deve essere controllato, grazie alla proprietà 5 degli alberi rosso-neri.
Le procedure RB-INSERT-FIXUP e RB-DELETE-FIXUP cambiano i colori dei nodi e ogni cambiamento di colore può potenzialmente causare $O(\lg n)$ cambiamenti di altezze nere. Dimostreremo che i cambiamenti di colore indotte da queste due procedure causano solo cambiamenti locali dell'altezza nera e quindi sono operazioni che possono essere eseguite in tempo costante. Supponiamo che l'altezza nera di ogni nodo x sia conservata nell'attributo $x.bh$.
Per RB-INSERT-FIXUP, ci sono tre casi da esaminare.
Caso 1: lo zio di z è rosso.

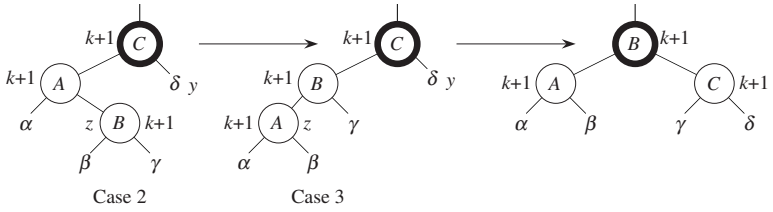


- Prima del cambio di colore, supponiamo che tutti i sottoalberi α, β, γ, δ ed ϵ abbiano la stessa altezza nera k e una radice rossa, e dunque che i nodi A, B, C e D abbiano altezza nera $k + 1$.

- Dopo la ricolorazione, l'unico nodo la cui altezza nera è cambiata è C. Per sistemarla, basta aggiungere l'istruzione $z.p.p.bh = z.p.p.bh + 1$ dopo le righe 7 e 21 nella procedura RB-INSERT-FIXUP.

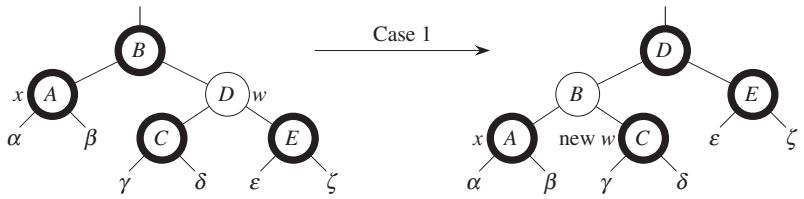
- Poiché il numero di nodi neri tra $z.p.p$ e z resta lo stesso, i nodi sopra $z.p.p$ non sono influenzati dal cambio di colore.

Caso 2: lo zio y di z è nero e z è un figlio destro.
Caso 3: lo zio y di z è nero e z è un figlio sinistro.



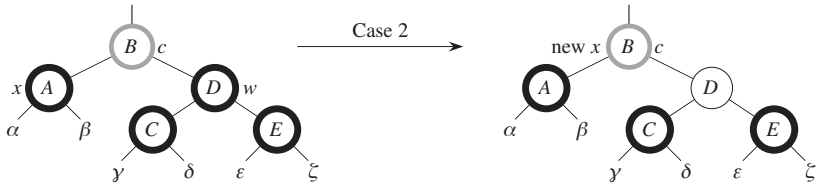
- Visto che i sottoalberi α, β, γ, δ ed ϵ hanno la stessa altezza nera k, anche dopo le ricolorazioni e le rotazioni, l'altezza nera dei nodi A, B e C rimane $k + 1$.

Quindi la procedura RB-INSERT-FIXUP conserva il tempo di esecuzione originale $O(\lg n)$.
Per RB-DELETET-FIXUP, ci sono quattro casi da esaminare.
Caso 1: il fratello di x è rosso.



- Anche se nel caso 1 avvengono dei cambi di colori e una rotazione, l'altezza nera non cambia.
- Il caso 1 cambia la struttura dell'albero, ma aspetta che si applichino i casi 2, 3 e 4 per gestire il colore nero extra su x.

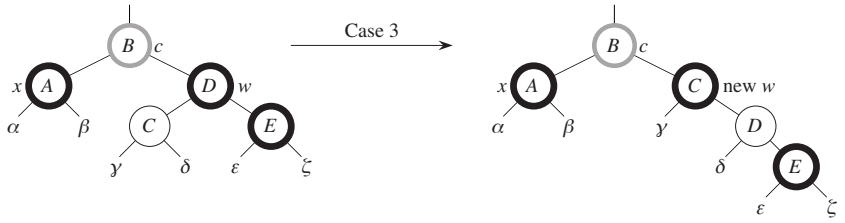
Caso 2: il fratello w di x è nero e entrambi i figli di w sono neri.



- Il nodo w diventa rosso, è il colore nero extra su x viene spostato su $x.p$.

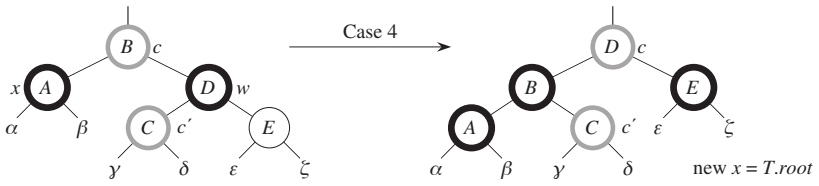
- Bisogna aggiungere l'istruzione $x.p.bh = x.bh$ dopo le righe 10 e 31 in RB-DELETET-FIXUP.
- Questo aggiornamento richiede un tempo costante, poi bisogna iterare per gestite il colore nero extra su $x.p$.

Caso 3: il fratello w di x è nero, il figlio sinistro di w è rosso e quello destro è nero.



- Indipendentemente dai cambi di colore e dalla rotazione, le altezze nere non cambiano.
- Il caso 3 modifica solo la struttura dell'albero, in modo da poter poi applicare in modo corretto il caso 4.

Caso 4: il fratello w di x è nero e il figlio destro di w è rosso.



- I nodi A, C ed E conservano i loro sottoalberi, quindi la loro altezza nera non cambia.
- Bisogna aggiungere le istruzioni

$$x.p.bh = x.bh + 1$$
$$x.p.p.bh = x.p.bh + 1$$

dopo le righe 21 e 42 di RB-DELETET-FIXUP, entrambe richiedono solo un costo costante.

- Il colore nero extra vene sistemato e il ciclo termina.

Quindi la procedura RB-DELETET-FIXUP conserva il tempo di esecuzione originale $O(\lg n)$.
Di conseguenza, possiamo concludere affermando che è possibile mantenere le altezze nere dei nodi come attributi negli alberi rosso-neri senza influire sulle prestazioni asintotiche delle operazioni sugli alberi rosso-neri.
Per quanto riguarda la seconda parte della domanda la risposta è no: non possiamo mantenere le profondità dei nodi senza influire sulle prestazioni asintotiche delle operazioni degli alberi rosso-neri. La profondità di un nodo dipende dalla profondità del padre. Quando la profondità di un nodo cambia, devono essere aggiornate le profondità di tutti i nodi sotto di esso nell'albero. L'aggiornamento della radice causa l'aggiornamento di $n - 1$ altri nodi, il che significa che le operazioni sull'albero che modificano la profondità dei nodi potrebbero non essere più eseguite in tempo $O(n \lg n)$.

Soluzione dell'Esercizio 17.3-6
L'idea consiste nel far passare una linea di scansione da sinistra a destra, mantenendo l'insieme dei rettangoli attualmente intersecati dalla linea in un albero di intervalli. L'albero di intervalli organizzerà tutti i rettangoli il cui lato orizzontale include la posizione corrente della linea di scansione e si baserà sui lati verticali dei rettangoli, in modo che gli intervalli sovrapposti nell'albero di intervalli corrispondano a dei rettangoli che si sovrappongono.
Pià precisamente, possiamo fare così.

1. Ordiniamo i rettangoli in base alle ascisse dei loro lati orizzontali (in realtà, ogni rettangolo deve comparire due volte nell'elenco ordinato: una per la sua ascissa minima e una per quella massima).

2. Ispezioniamo l'elenco ordinato (dalla coordinata minore alla maggiore).

   - Quando troviamo una ascissa che corrisponde al vertice sinistro di un rettangolo, controlliamo se il lato verticale del rettangolo si sovrappone a un lato di un rettangolo contenuto nell'albero di intervalli e inseriamo il rettangolo (in base all'intervallo che rappresenta il lato verticale) nell'albero.

- Quando troviamo una ascissa che corrisponde al vertice destrodi un rettangolo, eliminiamo il rettangolo dall'albero di intervalli.

L'albero di intervalli contiene sempre l'insieme dei rettangoli "aperti" intersecati dalla linea di scansione. Se troviamo una sovrapposizione nell'albero di intervalli, significa che ci sono rettangoli sovrapposti.

Il tempo di esecuzione è $O(n \lg n)$, poiché

- serve un tempo $O(n \lg n)$ per ordinare i rettangoli (usando il merge sort o l'heapsort), e
- serve un tempo $O(n \lg n)$ per le operazioni sull'albero di intervalli.(inserimento, cancellazione e verifica delle sovrapposizioni).

Soluzioni scelte per il Capitolo 19: Strutture dati per insiemi disgiunti

Soluzione dell'Esercizio 19.2-3

Vogliamo mostrare come assegnare un costo $O(1)$ a MAKE-SET e FIND-SET e un costo $O(\lg n)$ a UNION, in modo che il costo di una sequenza di queste operazioni sia sufficiente a coprire il costo della sequenza che, secondo il teorema, è $O(m + n \lg n)$. Oltre a poter parlare del costo di ogni tipo di operazione, ci serve anche poter parlare del numero di ogni tipo di operazione.

Consideriamo la solita sequenza di m operazioni MAKE-SET, UNION e FIND-SET, di cui n sono operazioni MAKE-SET, e sia $u < n$ il numero di operazioni UNION (ricordiamo il ragionamento fatto nel Paragrafo 19.1 riguardo al fatto che ci possono essere al più $n - 1$ operazioni UNION). Quindi abbiamo n operazioni MAKE-SET, u operazioni UNION e $m - n - u$ operazioni FIND-SET.

Il teorema non considera separatamente il numero u di operazioni UNION, ma semplicemente osservava che il loro numero è al più n. Se si ripete la dimostrazione del teorema considerando u operazioni UNION, si ottiene un limite di tempo $O(m - u + u \lg u) = O(m + u \lg u)$ per la sequenza di operazioni. Cioè, il tempo effettivo impiegato dalla sequenza di operazioni è al massimo $c(m + u \lg u)$, per una costante c.

Pertanto, vogliamo assegnare i costi delle operazioni in modo tale che il costo di un'operazione MAKE-SET moltiplicato per n sommato al costo di un'operazione FIND-SET per $m - n - u$ sommato al costo di un'operazione UNION per u sia maggiore o uguale a $c(m + u \lg u)$, così che i costi ammortizzati forniscano effettivamente una stima superiore del costo reale.

Se $c' \geq c$ è una costante qualunque, allora possiamo assegnare i seguenti costi ammortizzati:

- c' a MAKE-SET,

- c' a FIND-SET,
- $c'(\lg n + 1)$ a UNION.

Infatti in tal modo, sostituendo nella somma precedente, otteniamo

$$\begin{aligned} c'n + c'(m - n - u) + c'(\lg n + 1)u &= c'm + c'u \lg n \\ &= c'(m + u \lg u) \\ &> c(m + u \lg u) \, . \end{aligned}$$

Soluzione dell'Esercizio 19.2-6

Siano A e B le due liste e supponiamo che il rappresentante della nuova lista sia il rappresentante di A. Al posto di aggiungere B in coda ad A, inseriamo B in A subito dopo il primo elemento di A. Poiché dobbiamo comunque attraversare B per aggiornare i puntatori all'oggetto dell'insieme, alla fine basta solo fare in modo che l'ultimo elemento di B punti al secondo elemento di A.

Soluzioni scelte per il Capitolo 20: Algoritmi elementari per grafi

Soluzione dell'Esercizio 20.1-7
Per definizione, si ha

$$(BB^T)_{i,j} = \sum_{e \in E} b_{ie} b_{je} \ .$$

Abbiamo:

- se $i = j$, allora $b_{ie} b_{je} = 1$ (poiché in tal caso è $1 \cdot 1$ oppure $(-1) \cdot (-1)$) quando e entra o esce dal vertice i, ed è 0 altrimenti;
- se $i \neq j$, allora $b_{ie} b_{je} = -1$ $e = (i, j)$ o $e = (j, i)$, ed è 0 altrimenti.

Di conseguenza

$$(BB^T)_{i,j} = \begin{cases} \text{grado entrante di } i + \text{grado uscente di } i & \text{se } i = j \ , \\ -(\text{numero di archi che collegano } i \text{ e } j) & \text{se } i \neq j \ . \end{cases}$$

Soluzione dell'Esercizio 20.2-5
Dalla dimostrazione della correttezza dell'algoritmo di visita in ampiezza segue che $u.d = \delta(s, u)$, e inoltre l'algoritmo non assume che le liste di adiacenza siano in un ordine particolare.
Prendendo a modella la Figura 20.3 del libro, se t precede x in $Adj[w]$, otteniamo proprio l'albero di visita in ampiezza mostrato in figura. Ma se x precede t in $Adj[w]$ e u precede y in $Adj[x]$, allora nell'albero di visita in ampiezza è presente l'arco (x, u), che non è presente nell'altro caso, e dunque gli alberi che si ottengono sono diversi.

Soluzione dell'Esercizio 20.3-12

Le seguenti procedure modificano DFS e DFS-VISIT in modo da assegnare a ciascun vertice l'attributo cc richiesto dall'esercizio.

```
DFS-CC(G)
    for ogni vertice u ∈ G.V
        u.color = WHITE
        u.π = NIL
    time = 0
    for ogni vertice u ∈ G.V
        if u.color == WHITE
            counter = counter + 1
            DFS-VISIT(G, u, counter)

DFS-VISIT-CC(G, u, counter)
    u.cc = counter          // Etichetta il vertice
    time = time + 1
    u.d = time
    u.color = GRAY
    for ogni v ∈ G.Adj[u]
        if v.color == WHITE
            v.π = u
            DFS-VISIT(G, v, counter)
    time = time + 1
    u.f = time
    u.color = BLACK
```

La procedura DFS-CC incrementa un contatore ogni volta che DFS-VISIT-CC viene chiamata per far crescere un nuovo albero foresta di visita in profondità. Ogni vertice visitato (e aggiunto all'albero) da DFS-VISIT-CC è etichettato con lo stesso valore del contatore. Quindi, $u.cc = v.cc$ se e solo se u e v sono visitati nella stessa chiamata a DFS-VISIT-CC da parte di DFS, e il valore finale del contatore è il numero di chiamate fatte a DFS-VISIT-CC da parte di DFS. Inoltre, poiché ogni vertice prima o poi viene effettivamente visitato, ogni vertice viene etichettato.

Pertanto, è sufficiente dimostrare che i vertici visitati da ogni chiamata a DFS-VISIT-CC da parte di DFS corrispondono ai vertici di una certa componente connessa di G. Visto che dobbiamo provare un'uguaglianza tra insiemi, ovvero una doppia inclusione, separiamo il ragionamento in due.

- Tutti i vertici di una componente connessa sono effettivamente visitati da una chiamata di DFS a DFS-VISIT-CC.

  Infatti, sia u il primo vertice della componente C visitata da DFS-VISIT-CC. Poiché un vertice diventa non bianco solo quando viene visitato, al momento della chiamata di DFS-VISIT-CC per u tutti i vertici in C sono bianchi. Quindi, per il teorema del percorso bianco, tutti i vertici in C diventano discendenti di u nella foresta di visita in profondità, il che significa che tutti i vertici in C sono visitati (da chiamate ricorsive a DFS-VISIT-CC) prima che DFS-VISIT-CC ritorni a DFS.

- Viceversa, tutti i vertici visitati da una chiamata di DFS a DFS-VISIT-CC appartengono alla stessa componente connessa.

  Infatti, se due vertici vengono visitati nella stessa chiamata di DFS a DFS-VISIT-CC, allora si trovano nella stessa componente connessa, perché i vertici vengono visitati solo seguendo percorsi in G (cioè seguendo gli archi trovati nelle liste di adiacenza, a partire da un vertice).

Soluzione dell'Esercizio 20.4-3

Un grafo non orientato è aciclico (ovvero, è una foresta) se e solo se una visita in ampiezza non produce archi all'indietro.
Infatti:

- se c'è un arco all'indietro, allora c'è un ciclo;

- se non ci sono archi all'indietro, allora, per il Teorema 20.10, ci sono solo archi d'albero, e dunque il grafo è aciclico.

Quindi, per determinare se un grafo non orientato ha un ciclo, basta eseguire la procedura DFS e classificare gli archi: se uno è all'indietro, allora c'è un ciclo.
Il tempo di esecuzione è $O(V)$, e non $O(V + E)$, come si potrebbe pensare. Infatti, se G ha un ciclo, allora dopo aver visitato $|V|$ archi diversi, almeno uno deve essere un arco all'indietro, poiché in una foresta (non orientata) aciclica, per il Teorema B.2, si ha $|E| \leq |V| - 1$.

Soluzione del Problema 20-1

a. (a) Supponiamo che (u, v) sia un arco all'indietro o in avanti in una visita in ampiezza di un grafo non orientato. Senza perdita di generalità, sia u un antenato proprio di v nell'albero di visita in ampiezza. Poiché tutti gli archi di u vengono esplorati prima di esplorare gli archi di un qualunque discendente di u, l'arco (u, v) deve essere esplorato da u. Ma allora (u, v) deve essere un arco d'albero.

(b) Nella vista in ampiezza, un arco (u, v) è d'albero se la procedura pone $v.\pi = u$. Ma ciò avviene solo quando la procedura pone anche $v.d = u.d + 1$. Poiché né $u.d$ né $v.d$ cambiano in seguito, alla fine della visita avremo $v.d = u.d + 1$.

(c) Consideriamo un arco trasversale (u, v) dove, senza perdita di generalità, u viene visitato prima di v. Quando gli archi incidenti su u vengono esplorati, il vertice v deve essere già nella coda, altrimenti (u, v) sarebbe un arco d'albero. Poiché v è in coda, per il Lemma 20.3 abbiamo $v.d \leq u.d + 1$. Dal Corollario 20.4 segue che $v.d \geq u.d$. Quindi, $v.d = u.d$ oppure $v.d = u.d + 1$.

b. (a) Supponiamo che (u, v) sia un arco in avanti. Allora sarebbe stato esplorato durante l'esplorazione da u e avrebbe dovuto essere un arco d'albero.

(b) Vale lo stesso ragionamento svolto per i grafi non orientati.

(c) Per ogni arco (u, v), indipendentemente dal fatto che si tratti di un arco trasversale oppure no, non si può avere $v.d > u.d + 1$, poiché la visita in ampiezza analizza v v al più tardi quando esplora l'arco (u, v). Quindi, $v.d \leq u.d + 1$.

(d) Chiaramente, per ogni vertice v si ha $v.d \geq 0$. Se (u, v) è un arco all'indietro, allora v è un antenato di u nell'albero di visita in ampiezza, il che significa che $v.d \leq u.d + 1$ (osservate che, poiché i cappi sono considerati archi all'indietro, potremmo avere $u = v$).

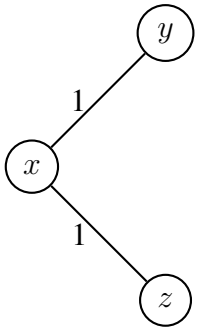
Soluzioni scelte per il Capitolo 21: Alberi di connessione minimi

Soluzione dell'Esercizio 21.1-1
Segue immediatamente dal Teorema 21.1, infatti basta considerare A vuoto e S un qualunque insieme che contiene u ma non v.

Soluzione dell'Esercizio 21.1-4
Un triangolo i cui archi hanno pesi uguali è un grafo in cui ogni arco è leggero e attraversa un taglio. Ma un triangolo è un ciclo, quindi non è un albero di connessione minimo.

Soluzione dell'Esercizio 21.1-6
Supponiamo che per ogni taglio di G esista un unico arco leggero che attraversi il taglio. Consideriamo due diversi alberi di connessione minimi di G, siano T e T'. Poiché T e T' sono diversi, T contiene un arco (u, v) che non sta in T'. Se (u, v) viene rimosso da T, allora T diventa disconnesso, dando luogo a un taglio $(S, V - S)$. L'arco (u, v) è leggero e attraversa il taglio $(S, V - S)$ (segue dall'Esercizio 21.1-3) e, per la nostra ipotesi, è l'unico arco leggero che attraversa questo taglio. Poiché (u, v) è l'unico arco leggero che attraversa $(S, V - S)$ e (u, v) non sta in T', ogni arco di T' che attraversa $(S, V - S)$ deve avere un peso strettamente maggiore di $w(u, v)$. Come nella dimostrazione del Teorema 21.1, possiamo identificare l'unico arco (x, y) in T' che attraversa $(S, V - S)$ e che si trova sul ciclo che risulta se aggiungiamo (u, v) a T'. In base alla nostra ipotesi, sappiamo che $w(u, v) < w(x, y)$. Possiamo quindi rimuovere (x, y) da T' e sostituirlo con (u, v), ottenendo un albero di connessione minimo con peso strettamente minore di $w(T')$. Pertanto, T' non era un albero di connessione minimo, contraddicendo l'ipotesi che il grafo ne avesse due diversi.
Ecco un controesempio per il viceversa.

90



In questo caso, il grafo è il suo stesso albero di connessione minimo, e quindi l'albero di connessione minimo è unico. Consideriamo il taglio $(\{x\}, \{y, z\})$. Gli archi (x, y) e (x, z) sono entrambi archi leggeri che attraversano il taglio, e dunque sono entrambi leggeri.

Soluzioni scelte per il Capitolo 22: Cammini minimi da sorgente unica

Soluzione dell'Esercizio 22.1-3

Se il numero massimo di archi su ogni percorso più breve dall'origine è m, allora la proprietà del rilassamento del cammino ci dice che dopo m iterazioni della procedura BELLMAN-FORD, ogni vertice v ha raggiunto il peso del cammino più breve in $v.d$. Per la proprietà del limite superiore, dopo le iterazioni, nessun valore di d cambierà più. Pertanto, nessun valore d cambierà nella $(m+1)$-esima iterazione. Poiché non conosciamo m in anticipo, non possiamo fare in modo che l'algoritmo iteri esattamente m volte e poi termini. Però, se l'algoritmo si ferma quando non cambia più nulla, si fermerà dopo $m+1$ iterazioni.

```
BELLMAN-FORD-EARLY-TERMINATION(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    repeat
            changes = FALSE for ogni arco (u, v) ∈ G.E
                if RELAX'(u, v, w)
                    changes = TRUE
        until changes == FALSE

RELAX'(u, v, w)
    if v.d > u.d + w(u, v)
        v.d = u.d + w(u, v)
        v.π = u
        return TRUE
    else return FALSE
```

Poiché l'esercizio specifica che G non ha cicli di peso negativo, il test per un ciclo di peso negativo (basato sull'esistenza di un valore d che cambierebbe se venisse

eseguito un altro passo del rilassamento) è stato rimosso. Se ci fosse un ciclo di peso negativo, questa versione dell'algoritmo non uscirebbe mai dal ciclo **repeat** perché un valore d cambierebbe a ogni iterazione.

Soluzione dell'Esercizio 22.3-3

Sì, l'algoritmo continuerebbe a funzionare. Infatti, sia u un vertice avanzato che che non è stato estratto dalla coda di priorità Q. Se u non è raggiungibile da s, allora $u.d = \delta(s,u) = \infty$. Se invece u è raggiungibile da s, allora esiste un percorso più breve $p = s \rightsquigarrow x \rightarrow u$. Quando il vertice x è stato estratto da Q, si ha $x.d = \delta(s,x)$, e quindi l'arco (x,u) era rilassato, di conseguenza $u.d = \delta(s,u)$.

Soluzione dell'Esercizio 22.3-7

Per trovare il percorso più affidabile tra s e t, possiamo eseguire l'algoritmo di Dijkstra assegnando agli archi peso $w(u,v) = \lg r(u,v)$ per trovare i percorsi più brevi da s in tempo $O(E + V \lg V)$. Il percorso più affidabile è il percorso più breve da s a t e l'affidabilità di tale percorso è il prodotto delle affidabilità dei suoi archi.

Ecco perché questo metodo funziona. Poiché le probabilità sono indipendenti, la probabilità che un percorso non fallisca è il prodotto delle probabilità che i suoi archi non falliscano. Vogliamo trovare un percorso $s \rightsquigarrow t$ che massimizzi $\prod_{(u,v)\in p} r(u,v)$. Ciò equivale a massimizzare $\lg\left(\prod_{(u,v)\in p} r(u,v)\right) = \sum_{(u,v)\in p} \lg r(u,v)$, che a sua volta equivale a minimizzare $\sum_{(u,v)\in p} -\lg r(u,v)$ (osservate che $r(u,v)$ può essere 0 e che $\lg 0$ non è definito; perciò, in questo algoritmo, poniamo $\lg 0 = -\infty$). Quindi, se assegniamo a ogni arco un peso $w(u,v) = \lg r(u,v)$, il problema è diventato un problema di cammino minimo.

Poiché $\lg 1 = 0$, $\lg x < 0$ per $0 < x < 1$, e abbiamo definito $\lg 1 = -\infty$, tutti i pesi w sono non negativi e possiamo usare l'algoritmo di Dijkstra per trovare i cammini minimi da s in tempo $O(E + V \lg V)$.

Soluzione alternativa

Possiamo anche tenere le probabilità originali eseguendo una versione modificata dell'algoritmo di Dijkstra che massimizza il prodotto delle affidabilità lungo un cammino invece di minimizzare la somma dei pesi lungo.

In questa variante, usiamo le affidabilità come pesi degli archi e facciamo le seguenti modifiche.

- La riga 2 di INITIALIZE-SINGLE-SOURCE diventa

  $v.d = -\infty$

- La procedura RELAX diventa

  RELAX(u, v, r)
  if $v.d < u.d \cdot r(u,v)$
  $v.d = u.d \cdot r(u,v)$
  $v.\pi = u$

- Nella procedura DIJKSTRA, Q diventa una coda di max-priorità, la riga 7 diventa

  $u =$ EXTRACT-MAX(Q)

  e le righe 11-12 diventano

  if la chiamata di RELAX aumenta il valore di $v.d$
  INCREASE-KEY$(Q, v, v.d)$

Questo algoritmo è isomorfo al precedente: esegue le stesse operazioni con la sola differenza che lavora direttamente con le probabilità originali invece che con quelle modificate.

Soluzione dell'Esercizio 22.4-7

Notiamo che, dopo il primo passaggio, tutti i valori di d sono al massimo 0 e che il rilassamento degli archi (v_0, v_i) non cambierà mai più un valore di d. Pertanto, possiamo eliminare v_0 eseguendo l'algoritmo di Bellman-Ford sul grafo dei vincoli senza il vertice v_0, ma inizializzando tutte le stime dei cammini minimi a 0 invece che a ∞.

Soluzione dell'Esercizio 22.5-4

Ogni volta che la procedura RELAX imposta π per un vertice, riduce anche il valore d del vertice. Pertanto, se $s.\pi$ viene posto a un valore che non è NIL, $s.d$ viene ridotto dal suo valore iniziale, che era 0, a un numero negativo. Ma $s.d$ è il peso di un cammino da s a s, che è un ciclo che contiene s. Pertanto, esiste un ciclo di peso negativo.

Soluzione del Problema 22-3

a. Possiamo usare l'algoritmo di Bellman-Ford sul grafo pesato e orientato $G = (V, E)$ fatto in questo modo. C'è un vertice per ogni valuta, e per ogni coppia di valute c_i e c_j ci sono i due archi diretti (c_i, c_j) e (c_j, c_i) (quindi $|V| = n$ e $|E| = n(n-1)$). Stiamo cercando un ciclo $\langle i_1, i_2, i_3, \ldots, i_k, i_1 \rangle$ tale che

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1 \, .$$

Prendendo i logaritmi dei due membri, otteniamo la disequazione

$$\lg R[i_1, i_2] + \lg R[i_2, i_3] + \cdots + \lg R[i_{k-1}, i_k] + \lg R[i_k, i_1] > 0 \, .$$

Moltiplicando per -1 troviamo

$$(-\lg R[i_1, i_2]) + (-\lg R[i_2, i_3]) + \cdots + (-\lg R[i_{k-1}, i_k]) + (-\lg R[i_k, i_1]) < 0 \, ,$$

e quindi vogliamo stabilire se G contiene un ciclo di peso negativo con questi pesi degli archi. Possiamo determinare se esiste un ciclo di peso negativo in G aggiungendo un vertice extra v_0 e per ogni $v_i \in V$ un arco (v_0, v_i) di peso 0, eseguendo la procedura BELLMAN-FORD da v_0 e usando il risultato booleano di BELLMAN-FORD (che è TRUE se non ci sono cicli di peso negativo e FALSE se ce n'è uno) per trovare la risposta. Più precisamente, invertiamo il risultato di BELLMAN-FORD.

Questo metodo funziona perché l'aggiunta del nuovo vertice v_0 e degli archi di peso 0 uscenti da v_0 non può introdurre nuovi cicli, ma garantisce che tutti i cicli di peso negativo siano raggiungibili da v_0. Ci vuole un tempo $\Theta(n^2)$ per costruire G, che ha $\Theta(n^2)$ archi. Poi ci vuole un tempo $O(n^3)$ tempo per eseguire BELLMAN-FORD. Il tempo totale è quindi $O(n^3)$.

Un altro modo per determinare se esiste un ciclo di peso negativo è quello di costruire G e, senza aggiungere v_0 e i relativi archi, eseguire uno degli algoritmi per la ricerca dei cammini minimi fra tutte le coppie. Se la matrice matrice di pesi di cammini minimi risultante ha un numero negativo sulla diagonale principale, allora c'è un ciclo di peso negativo.

b. Nota bene: la soluzione di questa parte serve anche come soluzione dell'Esercizio 22.1-7.

Supponendo di aver utilizzato la procedura BELLMAN-FORD per risolvere la parte (a), dobbiamo solo trovare i vertici di un ciclo di peso negativo. Possiamo farlo nel modo seguente. Esaminiamo ancora una volta gli archi. Quando

troviamo un arco (u, v) tale che $u.d + w(u, v) < v.d$, sappiamo che il vertice v si trova su un ciclo di peso negativo o è raggiungibile da uno di essi. Possiamo trovare un vertice sul ciclo di peso negativo ripercorrendo i valori di π da v, tenendo traccia dei vertici visitati fino a raggiungere un vertice x già visitato. A questo punto possiamo ripercorrere i valori π da x fino a tornare a x e tutti i vertici intermedi, insieme a x, costituiranno un ciclo di peso negativo. Possiamo usare il metodo ricorsivo dato dalla procedura PRINT-PATH nel Paragrafo 20.2, ma interrompendolo quando torna al vertice x.

Il tempo di esecuzione della procedura BELLMAN-FORD è $O(n^3)$, più $O(m)$ per controllare tutti gli archi e $O(n)$ per stampare i vertici del ciclo, quindi il tempo totale è ancora $O(n^3)$.

96

Soluzioni scelte per il Capitolo 23: Cammini minimi fra tutte le coppie

Soluzione dell'Esercizio 23.1-3
La matrice $L^{(0)}$ corrisponde alla matrice identità, infatti gli elementi ∞ (che è l'elemento neutro dell'operazione $\min$) corrispondono a 0 (l'elemento neutro della somma) e gli 0 (unità della somma) a 1 (unità del prodotto).

Soluzione dell'Esercizio 23.1-5
L'algoritmo per i cammini minimi tra tutte le coppie nel Paragrafo 23.1 calcola

$$L^{(n-1)} = W^{n-1} = L^{(0)} \cdot W^{n-1} \,,$$

dove $l_{ij}^{(n-1)} = \delta(i,j)$ e $L^{(0)}$ è la matrice identità. Quindi, l'elemento nella i-esima riga e j-esima colonna della matrice prodotto è il peso del cammino minimo da i a j e quindi la riga i-esima rappresenta la soluzione del problema del cammino minimo da sorgente unica per il vertice i.
Notiamo che quando si esegue il prodotto $C = A \cdot B$, la i-esima riga di C è la i-esima riga di A moltiplicata per B. Se a noi serve solo la riga i-esima di C, della matrice A non ci serve che la riga i-esima.
Quindi, la soluzione del problema dei cammini minimi da sorgente unica per il vertice i è $L_i^{(0)} \cdot W^{n-1}$, dove $L_i^{(0)}$ è la i-esima riga di $L^{(0)}$, cioè il vettore il cui elemento in posizione i è 0 e tutti gli altri sono ∞.
Eseguire le moltiplicazioni partendo da sinistra non è altro che eseguire l'algoritmo di Bellman-Ford. Il vettore corrisponde ai valori di d nella procedura BELLMAN-FORD, ovvero alle stime dei cammini minimi dalla sorgente per ogni vertice. Infatti:

- il vettore inizialmente vale 0 per la sorgente e ∞ per tutti gli altri vertici, come per i valori di d nella procedura INITIALIZE-SINGLE-SOURCE;

- ogni prodotto tra il vettore corrente e W rilassa ogni vertice esattamente come farebbe BELLMAN-FORD, ovvero la stima del peso in una riga, ad esempio quella relativa a v, viene aggiornata ad una stima migliore, se possibile, ottenuta aggiungendo $w(u, v)$ alla stima corrente relativa a u;
- questi rilassamenti/prodotti vengono eseguiti $n - 1$ volte.

Soluzione dell'Esercizio 23.2-4

Con gli apici, si ha $d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$. Se, eliminati gli apici, la procedura dovesse calcolare e memorizzare d_{ik} o d_{kj} prima di utilizzare questi valori per calcolare d_{ij}, potrebbe calcolare uno dei seguenti valori:

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k)} + d_{kj}^{(k-1)}\} ,$$

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k)}\} ,$$

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k)} + d_{kj}^{(k)}\} .$$

In ciascun caso, il codice calcola il peso di un cammino minimo da i a j i cui vertici intermedi appartengono all'insieme $\{1, 2, \ldots, k\}$. Se usiamo $d_{ik}^{(k)}$ al posto di $d_{ik}^{(k-1)}$, allora stima consideranno un sottocammino da i a k i cui vertici intermedi appartengono all'insieme $\{1, 2, \ldots, k\}$. Ma k non può essere un vertice intermedio di un cammino minimo da i a k, perché altrimenti questo cammino conterrebbe un ciclo su k e non sarebbe minimo. Di conseguenza, $d_{ik}^{(k)} = d_{ik}^{(k-1)}$. Ragionando in modo analogo si trova che anche $d_{kj}^{(k)} = d_{kj}^{(k-1)}$, e quindi nel calcolo si possono eliminare tutti gli apici.

Soluzione dell'Esercizio 23.3-4

Facendo così, si cambiano i cammini minimi. Consideriamo il grafo $G = (V, E)$ con $V = \{s, x, y, z\}$ e quattro archi con pesi $w(s, x) = 2$, $w(x, y) = 2$, $w(s, y) = 5$ e $w(s, z) = -10$. Seguendo le indicazioni del professor Greenstreet, otteniamo $\hat{w}$ sommando 10 ad ogni peso. Con i pesi w, il cammino minimo da s a y è $s \to x \to y$, che ha peso 4. Rispetto a $\hat{w}$ è $s \to y$, che ha peso 15 (infatti il cammino $s \to x \to y$ ha peso 24). Il problema è che, aggiungendo la stessa quantità a ogni arco, si penalizzano i percorsi con più archi, anche se i loro pesi sono bassi.

Soluzioni scelte per il Capitolo 24: Flusso massimo

Soluzione dell'Esercizio 24.2-11

Per ogni coopia di vertici u e v in G, possiamo definire una rete di flusso G_{uv} che consiste nella versione orientata di G con $s = u$, $t = v$ e in cui tutte le capacità degli archi valgono 1. Poiché una rete di flusso non può avere archi antiparalleli, per ogni arco in G, uno degli archi orientati in G_{uv} deve essere spezzato in due archi, con l'aggiunta di un nuovo vertice. Pertanto, G_{uv} ha $|V| + |E|$ vertici e $3|E|$ archi, cioè $O(V + E)$ vertici e $O(E)$, come richiesto. Poniamo tutte le capacità in G_{uv} a 1, in modo che il numero di archi di G che attraversano un taglio sia uguale alla capacità del taglio in G_{uv}. Indichiamo con f_{uv} un flusso massimo in G_{uv}. Allora l'arcoconnettività k è uguale a $\min_{u \in V}\{|f_{uv}| : v \in V - \{u\}\}$. Dimostreremo questo fatto tra poco. Prima, supponendo che l'affermazione sia corretta, mostriamo come calcolare k: basta usare la seguente procedura.

EDGE-CONNETTIVITY(G)
    $k = \infty$
    seleziona un vertice $u \in G.V$
    for ogni vertice $v \in G.V - \{u\}$
        costruisci una rte di flusso G_{uv} come sopra
        trova il flusso massimo f_{uv} su G_{uv}
        $k = \min\{k, |f_{uv}|\}$
    return k

Proviamo ora che effettivamente $k = \min_{u \in V}\{|f_{uv}| : v \in V - \{u\}\}$. L'affermazione segue dal teorema del flusso massimo/taglio minimo e da come abbiamo scelto le capacità in modo che la capacità di un taglio sia il numero di archi che lo attraversano. Dividiamo la dimostrazione in due parti.

- Si ha $k \geq \min_{u \in V}\{|f_{uv}| : v \in V - \{u\}\}$.

  Infatti, sia $m = \min_{u \in V}\{|f_{uv}| : v \in V - \{u\}\}$ e supponiamo di rimuovere solo $m - 1$ archi da G. Per ogni vertice v, per il teorema del flusso massimo/taglio minimo, u e v sono ancora connessi (infatti, il flusso massimo da u a v è almeno m, quindi ogni taglio che separa u da v ha capacità almeno m, il che significa che almeno m archi attraversano tale taglio, e dunque rimane almeno un arco che attraversa il taglio quando rimuoviamo $m - 1$ archi). Di conseguenza, ogni vertice è connesso a u, il che implica che il grafo è ancora connesso. Quindi, per disconnettere il grafo, devono essere rimossi almeno m archi, cioè $k \geq \min_{u \in V}\{|f_{uv}| : v \in V - \{u\}\}$.

- Si ha $k \leq \min_{u \in V}\{|f_{uv}| : v \in V - \{u\}\}$.

  Infatti, consideriamo un vertice v per il quale $|f_{uv}|$ sia minimo. Per il teorema del flusso massimo/taglio minimo, esiste un taglio di capacità $|f_{uv}|$ che separa u e v. Poiché tutte le capacità degli archi sono 1, esattamente $|f_{uv}|$ archi attraversano questo taglio. Se questi archi vengono rimossi, non c'è alcun percorso da u a v e quindi il nostro grafo diventa disconnesso, perciò $k \leq \min_{u \in V}\{|f_{uv}| : v \in V - \{u\}\}$.

Soluzione dell'Esercizio 24.3-3

Per definizione, un cammino aumentante è un cammino semplice $s \rightsquigarrow t$ nella rete residua G'_f. Poiché G non ha archi tra vertici in L e neppure tra quelli di R, lo stesso vale per la rete di flusso G'_f e tantomeno per G'. Inoltre, gli unici archi che coinvolgono s o t collegano s a L e R a t. Notiamo anche che, sebbene gli archi in G' possano andare solo da L a R, quelli in G'_f possono anche andare da R a L. Pertanto, ogni percorso aumentante deve essere del tipo

$$s \to L \to R \to \cdots \to L \to R \to t\ ,$$

e andare avanti e indietro tra L e R al più per il numero di volte che può farlo senza usare un vertice due volte. Contiene s, t e un numero uguale di vertici diversi di L e R, quindi in totale al più $2 + 2 \cdot \min\{|L|, |R|\}$ vertici. La lunghezza di un cammino aumentante (cioè il numero dei suoi archi) è quindi limitata superiormente da $2 + 2 \cdot \min\{|L|, |R|\}$.

Soluzione del Problema 24-4

a. Basta eseguire un'iterazione dell'algoritmo di Ford-Fulkerson. L'arco (u, v) in E la cui capacità è aumentata garantisce che (u, v) sia nella rete residua. Quindi cerchiamo un cammino aumentante e aggiorniamo il flusso se si trova un cammino. Il tempo di esecuzione è $O(V + E) = O(E)$, visto che il cammino aumentante può essere trovato con una visita in ampiezza o in profondità.

Vediamo ora che basta davvero una sola iterazione dell'algoritmo. Analizziamo separatamente i casi in cui (u, v) è o non è un arco che attraversa un taglio minimo. Se (u, v) non attraversa un taglio minimo, l'aumento della sua capacità non modifica la capacità di nessun taglio minimo e quindi il valore del flusso massimo non cambia. Se (u, v) attraversa un taglio minimo, allora aumentando la sua capacità di 1 si aumenta la capacità di quel taglio minimo di 1, e quindi eventualmente il valore del flusso massimo di 1. In questo caso, o non c'è un cammino aumentante (nel qual caso c'era qualche altro taglio minimo che (u, v) non attraversa), o il cammino aumentante incrementa il flusso di 1. In ogni caso, è sufficiente un'iterazione dell'algoritmo di Ford-Fulkerson.

b. Sia f il flusso massimo prima di ridurre $c(u, v)$.

Se $f(u, v) < c(u, v)$, non è necessario fare nulla.

Se $f(u, v) = c(u, v)$, dobbiamo aggiornare il flusso massimo. Poiché $c(u, v)$ è un numero intero che diminuisce, deve essere almeno 1, quindi $f(u, v) = c(u, v) \geq 1$.

Per ogni $x, y \in V$, sia $f'(x, y) = f(x, y)$, con l'eccezione che $f'(u, v) = f(u, v) - 1$. Anche se f' soddisfa tutti i vincoli sulle capacità, anche dopo che $c(u, v)$ è diminuito, non è un flusso accettabile, poiché viola la conservazione del flusso in u (a meno che non sia $u = s$) e in v (a meno che non sia $v = t$). Infatti f' ha un'unità di flusso in più in entrata da u rispetto a quella in uscita da u, e ha un'unità di flusso in più in uscita da v rispetto a quella in entrata.

L'idea è quella di cercare di reindirizzare questa unità di flusso in modo che esca da u ed entri in v attraverso qualche altro cammino. Se ciò non è possibile, dobbiamo ridurre il flusso da s a u e da v a t di 1 unità. Ecco come fare.

Cerchiamo un cammino aumentante da u a v (purché non sia da s a t).

- Se c'è, aumentiamo il flusso lungo tale cammino.

- Se non c'è, riduciamo il flusso da s a u aumentando il flusso da u a s. Cioè, cerchiamo un cammino aumentante $u \rightsquigarrow s$ in G_f e aumentiamo il suo flusso di 1 (sicuramente esiste un cammino di questo tipo, perché c'è flusso da s a u). Analogamente, riduciamo il flusso da v a t cercando un cammino aumentante $t \rightsquigarrow v$ in G_f e aumentando il suo flusso di 1.

Il tempo di esecuzione è ancora $O(V + E) = O(E)$, visto che il cammino aumentante può essere trovato con una visita in ampiezza o in profondità.

Indice