

# Esercizi Capitolo 5 - Alberi

Alberto Montresor

19 Agosto, 2014

Alcuni degli esercizi che seguono sono associati alle rispettive soluzioni. Se il vostro lettore PDF lo consente, è possibile saltare alle rispettive soluzioni tramite collegamenti ipertestuali. Altrimenti, fate riferimento ai titoli degli esercizi. Ovviamente, si consiglia di provare a risolvere gli esercizi personalmente, prima di guardare la soluzione.

Per molti di questi esercizi l'ispirazione è stata presa dal web. In alcuni casi non è possibile risalire alla fonte originale. Gli autori originali possono richiedere la rimozione di un esercizio o l'aggiunta di una nota di riconoscimento scrivendo ad `alberto.montresor@unitn.it`.

## 1 Problemi

### 1.1 Larghezza (Esercizio 5.4 del libro)

La larghezza di un albero ordinato è il numero massimo di nodi che stanno tutti al medesimo livello. Si fornisca una funzione che calcoli in tempo ottimo la larghezza di un albero ordinato  $T$  di  $n$  nodi.

**Soluzione:** Sezione [2.1](#)

### 1.2 Visite (Esercizio 5.5 del libro)

Gli ordini di visita di un albero binario di 9 nodi sono i seguenti:

- A, E, B, F, G, C, D, I, H (anticipato)
- B, G, C, F, E, H, I, D, A (posticipato)
- B, E, G, F, C, A, D, H, I (simmetrico).

Si ricostruisca l'albero binario e si illustri *brevemente* il ragionamento.

**Soluzione:** Sezione [2.2](#)

### 1.3 Albero inverso (Esercizio 5.9 del libro)

Dato un albero binario, i cui nodi contengono elementi interi, si scriva una procedura di complessità ottima per ottenere l'albero inverso, ovvero un albero in cui il figlio destro (con relativo sottoalbero) è scambiato con il figlio sinistro (con relativo sottoalbero).

**Soluzione:** Sezione [2.3](#)

### 1.4 Aggiungi un figlio (Esercizio 5.10 del libro)

Dato un albero binario i cui nodi contengono interi, si vuole aggiungere ad ogni foglia un figlio contenente la somma dei valori che appaiono nel cammino dalla radice a tale foglia. Si scriva una procedura ricorsiva di complessità ottima.

**Soluzione:** Sezione 2.4

### 1.5 Albero binario completo (Esercizio 5.12 del libro)

Un albero binario completo di altezza  $k$  è un albero binario in cui tutti i nodi, tranne le foglie, hanno esattamente due figli, e tutte le foglie si trovano al livello  $k$ . Si dimostri per induzione che, in un albero binario completo di altezza  $k$ , il numero dei nodi è  $2^{k+1} - 1$  ed il numero delle foglie è  $2^k$ .

**Soluzione:** Sezione 2.5

### 1.6 Visite iterative (Esercizio 5.13 del libro)

Utilizzando le pile, si scrivano tre procedure iterative di complessità ottima per effettuare, rispettivamente, le visite anticipata, differita e simmetrica di un albero binario.

**Soluzione:** Sezione 2.6

### 1.7 Altezza minimale

Dato un albero binario  $T$ , definiamo *altezza minimale* di un nodo  $v$  la minima distanza di  $v$  da una delle foglie del suo sottoalbero.

- Descrivere un algoritmo che riceve in input un nodo  $v$  e restituisce la sua altezza minimale.
- Calcolare la complessità in tempo dell'algoritmo proposto.

**Soluzione:** Sezione 2.7

### 1.8 Alberi binari strutturalmente diversi

Due alberi binari si dicono “strutturalmente” diversi se disegnando correttamente i figli destri e sinistri, si ottengono figure diverse. Ad esempio, un nodo radice con un figlio destro è diverso da un nodo radice con un figlio sinistro.

1. Si dica quanti sono i possibili alberi binari strutturalmente diversi composti da 1, 2, 3 e 4 nodi.
2. Dare una formula di ricorrenza per il caso generale di  $n$  nodi.
3. Data la ricorrenza al punto 2) trovare il più stretto limite asintotico inferiore che riuscite a trovare.  
Suggerimento: la formula risultante non vi ricorda nulla? Guardate negli appunti.

**Soluzione:** Sezione 2.8

### 1.9 Alberi pieni

Un albero binario “pieno” è un albero binario in cui tutti i nodi hanno esattamente 0 o 2 figli, e nessun nodo ha 1 figlio. Scrivere un programma ricorsivo che valuti  $P_n$ , ovvero il numero di alberi binari strutturalmente diversi che si possono ottenere con  $n$  nodi. Si valuti la complessità dell'algoritmo risultante.

**Soluzione:** Sezione 2.9

### 1.10 Altezza specificata

Dato un albero binario con radice  $T$  e un intero  $k$ , scrivere un algoritmo in pseudocodice che restituisca il numero di nodi di  $T$  che hanno altezza  $k$ . Calcolare la complessità in tempo e in spazio dell'algoritmo proposto.

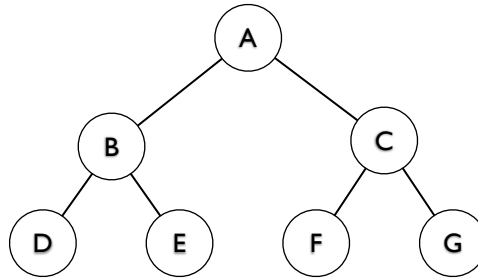


Figura 1: Un albero con lunghezza di cammino 10

**Soluzione:** Sezione 2.10

### 1.11 Lunghezza di cammino

In un albero binario, definiamo lunghezza di cammino come la somma delle distanze dei nodi dalla radice. Ad esempio, la lunghezza del cammino dell'albero in Figura 1 è pari a 10.

Scrivere un algoritmo in pseudo-codice per calcolare la lunghezza di cammino in un albero binario  $t$ . Discutere la complessità in tempo dell'algoritmo proposto in funzione del numero di nodi dell'albero.

**Soluzione:** Sezione 2.11

### 1.12 Larghezza di livello

Dato un albero binario  $T$  e un intero  $k$ , scrivere un algoritmo che restituisca il numero di nodi presenti nel livello  $k$ .

**Soluzione:** Sezione 2.12

## 2 Soluzioni

### 2.1 Larghezza (Esercizio 5.4 del libro)

È possibile modificare semplicemente un algoritmo di visita in ampiezza per assegnare un livello ad ognuno dei nodi, e contare quanti nodi appartengono allo stesso livello. Il costo dell'algoritmo risultante è ovviamente  $O(n)$ , oltre ad utilizzare una quantità di memoria aggiuntiva  $O(n)$  per associare un livello ad ogni nodo.

---

```

int larghezza(TREE t)
{
    if t == nil then
        return 0
    int larghezza = 1
    int level = 1
    int count = 1
    QUEUE Q = Queue()
    Q.enqueue(t)
    t.level = 0
    while not Q.isEmpty() do
        TREE u = Q.dequeue()
        if u.level ≠ level then
            level = u.level
            count = 0
        count = count + 1
        if count > larghezza then
            larghezza = count
        TREE v = u.leftmostChild()
        while v ≠ nil do
            v.level = u.level + 1
            Q.enqueue(v)
            v = v.rightSibling()
    return larghezza

```

---

Alcuni studenti hanno proposto invece una versione basata su DFS, che si appoggia su un vettore dinamico, indicizzato sui livelli, per contare il numero di nodi per livello, e poi selezionare il valore massimo. La complessità in tempo è sempre  $O(n)$ , la complessità in spazio è sempre  $O(n)$ , il codice è leggermente più semplice. Per variare, ne vediamo una versione basata su alberi binari.

---

```

int larghezza(TREE t)
{
    VECTOR count = new VECTOR
    larghezza(t, count, 0)
    return max(count)
}

larghezza(TREE t, VECTOR count, int level)
{
    if t ≠ nil then
        count[level] = count[level] + 1
        larghezza(t.left, count, level + 1)
        larghezza(t.right, count, level + 1)
}

```

---

Uno studente, Mattia Morandi, ha proposto la versione seguente, che sfrutta ancora meglio la visita per livelli, notando che quando tutti i nodi di un livello vengono estratti dalla coda, la coda contiene solo e unicamente i nodi del livello successivo. Basta quindi utilizzare la dimensione della coda come misuratore della larghezza del livello, e confrontarla con la larghezza massima trovata fino ad ora. Non solo, il valore della larghezza può essere copiato nella variabile *count*, e utilizzato per scoprire quando sarà terminato il prossimo livello. Il costo di questa funzione, più elegante della precedente, è sempre  $O(n)$ , ma in questo caso non è richiesta memoria aggiuntiva pari ad  $O(n)$ , ma solo pari a  $O(\ell)$ , dove  $\ell$  è la massima larghezza.

---

```

int larghezza(TREE t)
    if t == nil then
        return 0
    int count = 1           % Numero di nodi da visitare del livello corrente; inizialmente la radice
    int larghezza = 1       % Massimo larghezza trovata finora; inizialmente la radice
    QUEUE Q = Queue()
    Q.enqueue(t)
    while not Q.isEmpty() do
        TREE u = Q.dequeue()
        TREE v = u.leftmostChild()
        while v ≠ nil do
            Q.enqueue(v)
            v = v.rightSibling()
        count = count + 1
        if count == 0 then                                     % Nuovo livello
            count = Q.size()
            larghezza = max(larghezza, count)
    return larghezza

```

---

## 2.2 Visite (Esercizio 5.5 del libro)

L'albero risultante è mostrato in Figura 2. È possibile ragionare nel modo seguente:

- A è la radice, perchè visitata per prima nell'ordine anticipato;
- A ha figlio sinistro e destro, perchè B,E,G,F,C appartengono al suo sottoalbero sinistro e D,H,I al suo sottoalbero destro (per la visita in ordine simmetrico);
- E è il figlio sinistro di A, perchè il primo ad essere visitato dopo A nella visita anticipata;
- B ha figlio sinistro e destro, perchè B appartiene al sottoalbero sinistro di E e G,F,C al suo sottoalbero destro (per la visita in ordine simmetrico);
- B è figlio sinistro di E (per la visita in ordine simmetrico);
- F è figlio destro di E (per la visita in ordine anticipato);
- G e C sono figli sinistri e destro di F (per la visita in ordine simmetrico);
- D è figlio destro di A (in quanto primo ad essere esaminato dopo il sottoalbero sinistro di A nell'ordine anticipato);
- Restano H e I; poichè I viene visitato prima di H nell'ordine anticipato, I deve essere padre di H e figlio di D;
- poichè I non viene visitato prima di D nell'ordine simmetrico, non può essere suo figlio sinistro; è quindi figlio destro di D;

- poichè H è visitato prima di I nell'ordine simmetrico, H deve essere figlio sinistro di I;

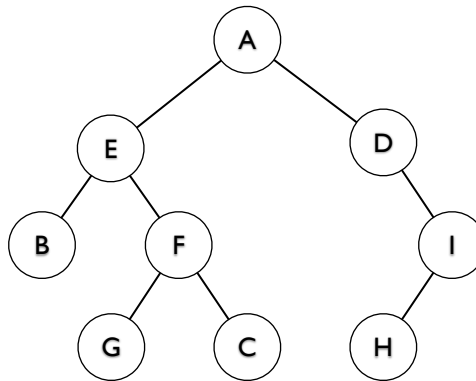


Figura 2: Albero risultante

### 2.3 Albero inverso (Esercizio 5.9 del libro)

L'algoritmo, molto semplicemente, scambia il sottoalbero destro e sinistro, e lavora ricorsivamente su di essi allo stesso modo. Il tempo di calcolo è  $O(n)$ .

---

```

inverti(TREE t)
  if t == nil then
    return
  t.left ↔ t.right
  inverti(t.left)
  inverti(t.right)

```

---

### 2.4 Aggiungi un figlio (Esercizio 5.10 del libro)

La procedura seguente risolve il problema in tempo  $O(n)$ . La chiamata iniziale è `addChild(t, 0)`.

---

```

addChild(TREE t, int v)
  if t == nil then
    return
  v = v + t.value
  if t.left == t.right == nil then
    t.insertLeft(v)
  else
    addChild(t.left, v)
    addChild(t.right, v)

```

---

### 2.5 Albero binario completo (Esercizio 5.12 del libro)

Si procede per induzione su  $k$ . Sia  $T_k$  un albero completo di altezza  $k$ ; sia  $n_k$  il numero di nodi contenuti in  $T_k$ . Per  $k = 0$ ,  $T_0$  è formato da un solo nodo, quindi  $n_0 = 2^{k+1} - 1 = 1$  è banalmente vero; inoltre, questo nodo è foglia, quindi il numero di foglie corrisponde a  $2^0 = 1$ .

È possibile notare che ogni foglia in  $T_{k-1}$  ha due figli foglia in  $T_k$ . Ragionando per induzione, sappiamo che il numero di nodi in  $T_k$  è pari a  $2^k - 1$  nodi, di cui  $2^{k-1}$  sono al livello  $k - 1$ . Il numero di foglie in  $T_k$  è quindi pari a  $2^{k-1} \cdot 2 = 2^k$ ; sommato ai nodi di  $T_{k-1}$  abbiamo  $2^k - 1 + 2^k = 2^{k+1} - 1$ , come volevasi dimostrare.

## 2.6 Visite iterative (Esercizio 5.13 del libro)

Questo è un esempio di pre-visita iterativa basata su stack.

---

previsita-iterativa(TREE  $t$ )

---

```
precondition:  $t \neq \text{nil}$ 
STACK  $S = \text{Stack}()$ 
 $S.\text{push}(t)$ 
while not  $S.\text{isEmpty}()$  do
    TREE  $u = S.\text{pop}()$ 
    visita del nodo  $u$ 
    if  $u.\text{right} \neq \text{nil}$  then
         $S.\text{push}(u.\text{right})$ 
    if  $u.\text{left} \neq \text{nil}$  then
         $S.\text{push}(u.\text{left})$ 
```

---

Nel caso della postvisita e della invisita, è necessario aggiungere un flag ai nodi; quando vengono inseriti per la prima volta, il flag viene settato a **false**; quando vengono estratti con flag a **false**, vengono re-inseriti con flag a **true**, avendo premura di intervallare i nodi figli e il nodo padre nel modo corretto. Quando un

nodo viene estratto con flag a **true**, la visita viene eseguita effettivamente.

---

postvisita-iterativa(TREE *t*)

---

```

precondition: t ≠ nil
STACK S = Stack()
S.push(t, false)
while not S.isEmpty() do
    u, f = S.pop()
    if f then
        | visita del nodo u
    else
        | S.push(u, true)
        | if u.right ≠ nil then
        | | S.push(u.right, false)
        | if u.left ≠ nil then
        | | S.push(u.left, false)

```

---



---

invisita-iterativa(TREE *t*)

---

```

precondition: t ≠ nil
STACK S = Stack()
S.push(t, false)
while not S.isEmpty() do
    u, f = S.pop()
    if f then
        | visita del nodo u
    else
        | if u.right ≠ nil then
        | | S.push(u.right, false)
        | S.push(u, true)
        | if u.left ≠ nil then
        | | S.push(u.left, false)

```

---

## 2.7 Altezza minimale

L'altezza minimale del nodo radice di un albero è pari al minimo delle altezze minimali dei due sottoalberi, aumentata di 1. Quindi un semplice algoritmo ricorsivo per calcolare l'altezza minimale è il seguente:

---

int altezza-minimale(TREE *T*)

---

```

if T == nil then
    | return +∞
if T.left() == nil and T.right() == nil then
    | return 0
hl = altezza-minimale(T.left())
hr = altezza-minimale(T.right())
return min(hl, hr) + 1

```

---

L'algoritmo risultante viene eseguito in tempo  $O(n)$ , in quanto deve visitare tutti i nodi.



## 2.8 Alberi strutturalmente diversi

Otteniamo una risposta per la domanda (1) risolvendo la domanda (2) e applicando poi la formula.

Per definire una ricorrenza, ci basiamo sulla seguente osservazione. Un albero di  $n$  nodi ha sicuramente una radice; i restanti  $n - 1$  nodi possono essere distribuiti nei sottoalberi destro e sinistro della radice. Ad esempio, un albero di 4 nodi può avere 3 nodi nel sottoalbero destro e 0 nel sinistro; oppure 2 nodi nel destro e 1 nel sinistro; oppure 1 nel destro e 2 nel sinistro; oppure 0 nel destro e 3 nel sinistro.

Detto quindi  $k$  il numero di nodi nell'albero sinistro, una formula ricorsiva per calcolare il numero di nodi  $P(n)$  è la seguente:

$$P(n) = \sum_{k=0}^{n-1} P(k)P(n-1-k)$$

I casi base sono rappresentati da  $P(0) = 1$ .

I valori per  $n = 1, 2, 3, 4, 5$  corrispondono a 1, 2, 5, 14, 42.

È possibile dimostrare che  $P(n)$  è  $\Omega(2^n)$ , ovvero che  $\exists c > 0, m \geq 0 : P(n) \geq c2^n, \forall n \geq m$ :

- Passo induttivo: supponiamo la proprietà dimostrata per i valori inferiori ad  $n$  ( $P(n') \geq c2^{n'}, \forall n' < n$ ), e dimostriamo che vale per il valore  $n$ .

$$\begin{aligned} P(n) &= \sum_{k=0}^{n-1} P(k)P(n-1-k) \\ &\geq \sum_{k=0}^{n-1} (c \cdot 2^k) \cdot (c \cdot 2^{n-1-k}) \\ &= c^2 \cdot n \cdot 2^{n-1} \\ &\geq c2^n \end{aligned}$$

L'ultima condizione è vera per  $c \geq \frac{2}{n}$ .

- Casi base:

$$P(0) = 1 \geq c \cdot 2^0 \Leftrightarrow c \leq 1$$

Qui abbiamo un problema: le due disequazioni (per il caso base e per il passo induttivo) non sono compatibili; da un lato si chiede di avere un valore "maggiore o uguale" di  $2/n$ , dall'altro "minore o uguale" di 1.

E' possibile considerare estendendo la formula:

$$\begin{aligned} P(1) = 1 &\geq c \cdot 2^1 \Leftrightarrow c \leq \frac{1}{2} & P(2) = 2 &\geq c \cdot 2^2 \Leftrightarrow c \leq \frac{1}{2} \\ P(3) = 5 &\geq c \cdot 2^3 \Leftrightarrow c \leq \frac{5}{8} \\ P(4) = 14 &\geq c \cdot 2^4 \Leftrightarrow c \leq \frac{14}{16} \\ P(5) = 42 &\geq c \cdot 2^5 \Leftrightarrow c \leq \frac{42}{32} \end{aligned}$$

Fino a  $n = 3$ , le due condizioni non sono compatibili. Da  $n = 4$  in poi, è possibile trovare un valore  $c$  che rispetti entrambe le condizioni (ad esempio,  $c = 1/2$ ). In altre parole, il valore  $c = \frac{1}{2}$  è un valore adatto a soddisfare la nostra condizione per tutti i valori di  $n \geq 4$ .

Questa sequenza corrisponde all' $n$ -esimo numero catalano (vedi programmazione dinamica), per il quale un limite asintotico più stretto è:

$$\Omega\left(\frac{4^n}{n^{3/2}}\right)$$

## 2.9 Alberi pieni

Il numero di alberi pieni strutturalmente diversi è calcolato nel modo seguente. Innanzitutto, è impossibile costruire un albero pieno se  $n$  è pari. Questo perché gli alberi pieni si ottengono partendo dalla radice e aggiungendo via via coppie di figli. Per i valori dispari, il numero di nodi pieni è ottenuto ricorsivamente partendo dal caso base  $n = 1$  (che corrisponde ad un albero solo). Per  $n > 1$ , si crea una radice e poi si dividono i rimanenti  $n - 1$  figli fra il sottoalbero destro e il sottoalbero sinistro. Se  $i$  nodi vanno a destra,  $1 \leq i \leq n - 2$ ,  $(n - 1) - i$  nodi vanno a sinistra.

$$T[n] = \begin{cases} 0 & \text{se } n \text{ è pari} \\ 1 & \text{se } n = 1 \\ \sum_{i=1}^{n-2} T(i) \cdot T((n-1) - i) & \text{se } n > 1 \end{cases}$$

È possibile calcolare il valore con un semplice programma ricorsivo, ma questo avrebbe complessità superpolinomiale. Per velocizzare l'algoritmo, è opportuno utilizzare la programmazione dinamica.

---

```

int alberipieni(int n)
{
    if n è pari then
        | return 0
    else
        | int[] D = new int[1...n]
        | D[1] = 1
        | for int i = 3 to n step 2 do
        |   | D[i] = 0
        |   | for int j = 1 to i - 2 step 2 do
        |   |   | D[i] = D[i] + D[j] · D[(n - 1) - j]
        | return D[n]
}

```

---

La complessità di questo algoritmo è  $O(n^2)$ .

## 2.10 Altezza specificata

Ricordiamo che in un albero  $T$ , l'altezza di un nodo  $v$  è la massima distanza di  $v$  da una sua foglia. Vogliamo descrivere un algoritmo che calcoli l'altezza di  $v$ , per ogni nodo  $v$  dell'albero  $T$ . È chiaro che l'altezza di una foglia è 0; invece, l'altezza di un nodo interno  $v$  si calcola facilmente una volta note le altezze dei suoi figli: è sufficiente considerare la massima tra queste e incrementarla di 1.

Definiamo una procedura ricorsiva  $\text{visita}(v)$  che, su input  $v$ , restituisce l'altezza di  $v$ . Tale procedura restituisce il valore 0 se  $v$  è una foglia; altrimenti richiama se stessa sui figli di  $v$  determinando il valore massimo ottenuto, incrementa di 1 tale valore e lo assegna alla variabile  $h$ . Se inoltre  $h$  coincide con  $k$  si incrementa un opportuno contatore rappresentato da una variabile globale  $s$  (inizialmente posta a 0). Il valore finale di  $s$  sarà l'output dell'algoritmo.

Il costo della visita postordine è  $O(n)$ .

---

```
int countK(TREE v)
```

---

```
    s = 0
    visita(v)
    return s
```

---



---

```
int visita(TREE v)
```

---

```
    int L = iff(v.left = nil, -1, visita(v.left))
    int R = iff(v.right = nil, -1, visita(v.right))
    int h = max(L, R) + 1
    if h == k then
        | s = s + 1
    return h
```

---

## 2.11 Lunghezza di cammino

La soluzione è una semplice procedura ricorsiva di costo  $O(n)$ . La chiamata iniziale è `lunghezzaCammino(t, 0)`.

---

```
int lunghezzaCammino(TREE t, int ℓ)
```

---

```
    if t == nil then
        | return 0
    return ℓ + lunghezzaCammino(t.left, ℓ + 1) + lunghezzaCammino(t.right, ℓ + 1)
```

---

## 2.12 Larghezza di livello

Dato un nodo  $T$ , il numero di nodi a livello  $k$  è pari alla somma del numero di nodi a livello  $k - 1$  nel sottoalbero radicato nei figli sinistro e destro di  $T$ . Al solito, l'algoritmo risultante è una visita (post-visita) il cui costo computazionale è  $O(n)$ .

---

```
int depth(TREE v, int k)
```

---

```
    if t == nil then
        | return 0
    if k == 0 then
        | return 1
    return depth(T.left(), k - 1) + depth(T.right(), k - 1)
```

---

### 3 Problemi aperti

#### 3.1 Raggruppa le foglie con 0 e 1 (Esercizio 5.8 del libro)

Dato un albero binario le cui foglie contengono 0 od 1 e i cui nodi interni contengono solo 0, si vuole cambiare il contenuto delle foglie in modo che, visitandole da sinistra verso destra, si incontrino prima tutti gli 0 e poi tutti gli 1. Si scriva una procedura ricorsiva di complessità ottima.

#### 3.2 Discendenti specificati

Dato un albero con radice  $t$  e un intero  $k$ , scrivere un algoritmo che restituisca il numero di nodi in  $t$  il cui numero di *discendenti* è pari a  $k$ .

#### 3.3 Potatura

Dato un albero binario completo, con radice  $T$ , rappresentato con puntatori primo figlio/fratello, scrivere un algoritmo che “poti” l’albero, ovvero elimini tutti i nodi foglia.