Dati e Algoritmi (Pietracaprina)

Esercizi sugli Alberi

Problema 1 *Si supponga che la visita in preorder di un albero ordinato di 6 nodi incontri i nodi nell'ordine ABCDEF. Dire quali delle seguenti sequenze può rappresentare la visita in postorder dello stesso albero* (*motivando la risposta e disegnando l'albero*)*: BAFECD, CDBFEA, CDAEFB.*

Soluzione. L'unica sequenza che può rappresentare la visita in postorder dello stesso albero è CDBFEA, in quanto il primo nodo visitato in preorder (la radice), ovvero il nodo A, deve essere l'ultimo visitato in postorder. L'unico albero compatibile con entrambe le visite è il seguente: A radice; B ed E figli della radice, C e D figli di B; ed F figlio di E. □

Problema 2 (Esercizio C-8.50 del testo [GTG14]) *Sia T un albero. Dati due nodi $v, w \in T$ si definisce il* Lowest Common Ancestor *di v e w* ($\text{LCA}(v, w)$) *come l'antenato comune più profondo. Progettare un algoritmo efficiente per calcolare* $\text{LCA}(v, w)$*, analizzandone la complessità.*

Soluzione. L'idea dell'algoritmo è di risalire dal più profondo dei due nodi passati in input sino all'antenato che sta alla stessa profondità dell'altro, e da qui risalire da entrambi sino a trovare il primo antenato comune. (Si osservi che un antenato in comune esiste sicuramente ed è la radice.) Per determinare la profondità di un nodo si utilizza l'algoritmo `depth` visto a lezione. Lo pseudocodice dell'algoritmo richiesto è il seguente:

Algoritmo `LCA(v,w)`
input `nodi v,w ∈ T`
output `LCA(v,w)`

```
d_v ← T.depth(v); d_w ← T.depth(w);
if (d_v > d_w) then for i ← 1 to d_v − d_w do v ← T.parent(v)
else for i ← 1 to d_w − d_v do w ← T.parent(w);
while (v ≠ w) do {
  v ← T.parent(v)
  w ← T.parent(w)
}
return v
```

Le invocazioni di `depth` hanno complessità proporzionale alle profondità di v e w. I cicli **for** e il ciclo **while** eseguono, ciascuno, un numero di iterazioni limitato superiormente dalla massima profondità dei due nodi, e in ciascuna iterazione eseguono un numero costante di operazioni. Dato che la profondità di un nodo di un albero è limitata superiormente dall'altezza dell'albero, concludiamo che la complessità dell'algoritmo è $O(h)$, con h altezza di T. In effetti la complessità è $\Theta(h)$, dato che esiste un'istanza che richiede tempo proporzionale ad h (quella in cui v è la foglia a profondità massima in T, ovvero profondità h). Si osservi che la correttezza dell'algoritmo e l'analisi non richiedono assunzioni sull'arietà di T. □

Problema 3 *Si supponga di calcolare l'altezza di un albero T di n nodi invocando l'algoritmo* `depth`(v) *da ciascuna foglia v di T e restituendo come altezza la massima profondità ottenuta. Dimostrare che tale strategia ha una complessità al caso pessimo $\Omega\left(n^2\right)$. (È l'algoritmo* `heightBad` *decritto in [GTG14]. Si veda anche l'esercizio C-8.27 del testo.*)

Soluzione. L'algoritmo `heightBad` è il seguente

Algoritmo `heightBad(T)`
input `Albero T`
output `Altezza di T calcolata come max depth di una foglia`

```
h ← 0;
forall v ∈ T.positions() do {
  if T.isExternal(v) then h ← max{h, T.depth(v)}
}
return h
```

Per ottenere un lower bound alla complessità è sufficiente limitarsi al costo delle invocazioni di `depth`. Si ricordi che `depth`(v) richiede $\Theta\left(d_v\right)$ operazioni, dove d_v è la profondità del nodo v. Si ha quindi che la complessità della strategia indicata è

$$\Omega\left(\sum_{\text{foglie } v\in T}(d_v)\right)$$

Gli esempi nella Fig. 1 mostrano che per ogni n esistono alberi di n nodi in cui $\sum_{\text{foglie } v\in T}(d_v) \in \Omega\left(n^2\right)$. (Gli esempi considerano il caso n pari, ma possono essere facilmente estesi al caso n dispari.) Nell'albero a sinistra ci sono $n/2$ foglie a profondità $n/2$ ciascuna, mentre in quello a destra ci sono $\lceil n/4 \rceil$ foglie a profondità $\lceil n/4 \rceil$ ciascuna. □

Problema 4 *Si consideri un albero T in cui ogni nodo v contiene un valore binario, restitutito da* `v.getElement()`*, e si dice* alive *se tale valore è* 1*, e* dead *se esso è* 0*. Progettare un algoritmo* ricorsivo `allLiveAncestors` *che per ogni nodo $v \in T$ memorizzi in un campo v.*LA *il numero di antenati* alive *di v, e analizzare la complessità.*

Soluzione. Detta `allLiveAncestors(T,v)` la generica invocazione su un nodo $v \in T$, per risolvere il problema si eseguirà `allLiveAncestors(T,T.root())`. L'idea è simile a quella usata nell'algoritmo `allDepths` visto a lezione. Si usa lo schema della visita in preorder di T_v impostando il campo `LA` di v basandosi sul valore del campo corrispondente nel padre (che sarà già stato impostato) e del bit contenuto nel nodo. Dopodiché, se v è interno, si invoca ricorsivamente l'algoritmo sui figli. Lo pseudocodice è il seguente:

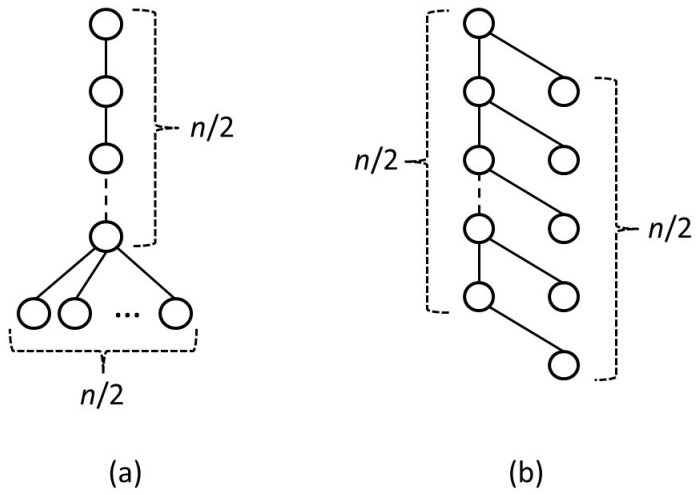


Figure 1: Esempi di alberi in cui la somma delle profondità delle foglie è $\Omega\left(n^2\right)$.

Algoritmo `allLiveAncestors(T,v)`
input `nodo v ∈ T e ∀u∈T, u antenato di v, u.LA correttamente impostato`
output `T con u.LA correttamente impostato ∀u∈T`$_v$

if `T.isRoot(v)` **then** `v.LA ← v.getElement()`
else `v.LA ← v.getElement()+T.parent(v).LA`
foreach `(w ∈ T.children(v))` **do** `allLiveAncestors(T,w)`

Analizziamo la complessità di `allLiveAncestors(T,T.root())`. Si osserva che l'algoritmo esegue una visita in preorder di T, dove la visita di un nodo interno corrisponde alla impostazione del campo v`.LA`, che richiede $\Theta(1)$ operazioni. Dall'analisi della visita in preorder fatta a lezione si ricava immediatamente che la complessità è $\Theta(n)$, dove n è il numero di nodi di T. □

Problema 5 *Progettare un algoritmo che dato un albero T, per ciascun nodo $v \in T$ memorizzi la sua altezza in un campo v*`.height`*, e analizzarne la complessità in funzione del numero di nodi n dell'albero.*

Soluzione. L'algoritmo `allHeights(T,v)` riportato sotto è un semplice adattamento dell'algoritmo `height` visto a lezione. Si osservi che oltre che impostare il campo `height` dei nodi del sottoalbero T_v restituisce anche l'altezza di v. In questo modo, dall'invocazione ricorsiva sui figli di v si ottengono le loro altezze e si è quindi in grado di determinare l'altezza di v.

Algoritmo `allHeights(T,v)`
input `nodo v` $\in$ `T`
output `altezza di v e T con u.height correttamente impostato` $\forall$`u`$\in$`T`$_v$

```
h ← 0
foreach (w ∈ T.children(v)) do h ← max {h,1+allHeights(T,w)}
v.height ← h
return h
```

Per memorizzare le altezze in tutti i nodi dell'albero sarà sufficiente invocare `allHeights`(T,T.`root()`). La complessità è la stessa dell'algoritmo `height`, ovvero $\Theta(n)$.

Osservazione. Questo esercizio (come diversi altri nella dispensa) mostra che, a volte, *calcolare una informazione più ricca permette di migliorare le prestazioni.* □

Problema 6 *Si consideri un albero binario proprio T dove ciascun nodo v contiene un intero v.*`val` ≥ 0. *Un nodo $v \in T$ si dice* massimale *se vale la condizione v.*`val` $\geq u$.`val` *per ogni u antenato di v. Si noti che la radice è sempre massimale.*

a. *Progettare in pseudocodice un algoritmo ricorsivo* `MaximalSet` *che, per ogni nodo v, imposta un flag binario v.*`maximal` *a* 1*, se v è massimale, a* 0 *altrimenti. Alla fine dell'algoritmo il campo* `maximal` *di tutti i nodi deve risultare impostato.*

b. *Analizzare la complessità dell'algoritmo progettato per il punto precedente.*

Soluzione.

a. L'algoritmo è basato su una visita in preorder di T e nella generica invocazione sul nodo v si passa come parametro anche un intero m che rappresenta il massimo valore in un antenato proprio di v. Lo pseudocodice è il seguente:

```
Algoritmo MaximalSet (T,v,m)
input nodo v ∈ T e m = massimo u.val con u antenato di v, u ≠ v
output T con u.maximal impostato correttamente ∀ u ∈ T_v.

if T.isRoot(v) then v.maximal ← 1
else {
  if v.val ≥ m then v.maximal ← 1;
  else v.maximal ← 0
}
if T.isInternal(v) then {
  m' ← max {m,v.val};
  MaximalSet(T,T.left(v),m');
  MaximalSet(T,T.right(v),m');
```

La prima invocazione è `MaximalSet`$(T, T.$`root`$(), -1)$. Si noti che invece di passare il parametro m come input, si poteva usare un campo aggiuntivo in ogni nodo v in cui veniva salvato il massimo valore di un suo antenato.

b. Analizziamo la complessità di `MaximalSet`. L'algoritmo esegue una visita in preorder di T, dove la visita di un nodo v corrisponde a tutte le operazioni richieste per impostare il campo $v.$`maximal` ed eventualmente il nuovo massimo m'. Quindi la visita di un nodo richiede $\Theta(1)$ operazioni. Dall'analisi della visita in preorder fatta a lezione si ricava immediatamente che la complessità è $\Theta(n)$, dove n è il numero di nodi di T.

□

Problema 7 *Dimostrare che in un albero binario proprio con m foglie e altezza h si ha che $m \geq h + 1$.*

Soluzione. Dimostriamo la relazione per induzione sull'altezza h dell'albero. La base $h = 0$ è vera in quanto un albero di altezza 0 ha 1 foglia, e quindi $m = h + 1$. Fissiamo $h \geq 0$, supponiamo che la relazione sia vera per alberi di altezza sino ad h, e consideriamo un albero T di altezza $h + 1 > 0$. Siano T_1 e T_2 i due sottoalberi figli della radice. Si indichi con m_i il numero di foglie in T_i e con h_i l'altezza di T_i $(i = 1, 2)$. Si ricordi che, per definizione, l'altezza di T, ovvero $h + 1$, è uguale a $1 + \max\{h_1, h_2\}$, e vale quindi che $h_1, h_2 \leq h$. Se T ha m foglie si ha che

$$\begin{aligned} m &= m_1 + m_2 \\ &\geq h_1 + 1 + h_2 + 1 \qquad \text{(per hp. induttiva)} \\ &\geq \max\{h_1, h_2\} + 2 \\ &= (h + 1) + 1. \end{aligned}$$

□

Problema 8 *Dimostrare che per ogni $n > 0$ dispari esiste un albero binario proprio T con n nodi e altezza $\Theta(\sqrt{n})$.*

Soluzione. Se $n = 1$ la dimostrazione è banale. Supponiamo $n \geq 3$. Sia $h = \lfloor \sqrt{n} \rfloor$. L'albero cercato si ottiene combinando un albero molto sbilanciato T' di altezza h e con $2h+1$ nodi, al quale viene attaccato, come figlio della radice, un albero molto bilanciato T'' con $n-2h$ nodi e altezza $O(\log n)$. Si noti che l'ipotesi $n \geq 3$, assicura che $n - 2h \geq 0$. Più specificatamente, si consideri come T' un albero costituito da una dorsale di $h+1$ nodi $u_0, u_1, \ldots, u_h$, con u_0 radice e u_i figlio sinistro di u_{i-1}, e dai figli destri di $u_0, u_1, \ldots, u_{h-1}$. Invece, si consideri come T'', la cui radice viene fatta coincidere con il figlio destro di u_0, un albero tale che ogni suo livello i, tranne l'ultimo, ha il massimo numero (2^i) di nodi. Esempi di T' e T'' sono stati disegnati nei lucidi sugli Alberi Binari presentati a lezione, come casi di massimo e minimo sbilanciamento. □

Problema 9 *Dimostrare che in un albero non vuoto T con n nodi di cui m foglie, dove ogni nodo interno ha almeno 2 figli, si ha che $m \geq n - m + 1$.*

Soluzione. La dimostrazione procede per induzione sull'altezza h dell'albero. Come base osserviamo che la proprietà è vera per un albero di altezza $h = 0$, che quindi ha un solo nodo (foglia). Supponiamo che la proprietà sia vera per alberi di altezza al più h, con $h \geq 0$ fissato, e consideriamo un albero T di altezza $h + 1 \geq 1$. Chiaramente la radice di T deve essere un nodo interno. Sia $k \geq 2$ il numero di figli della radice. Per $1 \leq i \leq k$ sia n_i il numero di nodi ed m_i il numero di foglie nel sottoalbero radicato nelll'i-esimo figlio della radice. Sia inoltre m il numero di foglie ed n il numero di nodi di T. Si ha allora che

$$\begin{aligned} n &= 1 + \sum_{i=1}^{k} n_i \\ m &= \sum_{i=1}^{k} m_i. \end{aligned}$$

Poichè i sottoalberi figli della radice hanno altezza al più h, possiamo applicare l'ipotesi induttiva e dedurre che

$$m = \sum_{i=1}^{k} m_i \geq \sum_{i=1}^{k} (n_i - m_i + 1) = \sum_{i=1}^{k} n_i - \sum_{i=1}^{k} m_i + k = n - m + (k-1) \geq n - m + 1,$$

in quanto $k \geq 2$. □

Problema 10 *Si definisca* albero d-ario proprio *un albero ordinato in cui ogni nodo interno ha esattamente d figli.*

a. *Disegnare due alberi ternari* ($d = 3$) *propri* T_1 *e* T_2 *con* 8 *nodi interni ciascuno, tali che* T_1 *ha altezza massima e* T_2 *ha altezza minima. Quante foglie hanno* T_1 *e* T_2*?*

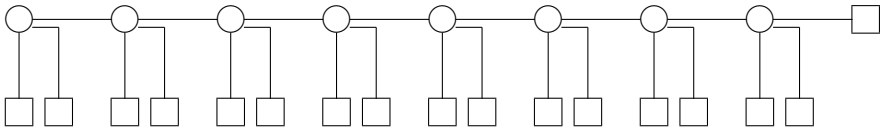
b. *Sia* T *un albero* d*-ario proprio di altezza* h *con* n *nodi interni ed* m *foglie. Usando gli esempi precedenti e il buon senso trovare l'unica tra le seguenti relazioni che può valere per* T *arbitrario e* $d \geq 2$*:*

$$\text{(i) } m = (d-1)^h + 1; \text{ (ii) } m = (d-1)n + 1; \text{ (iii) } m = 5d + 2$$

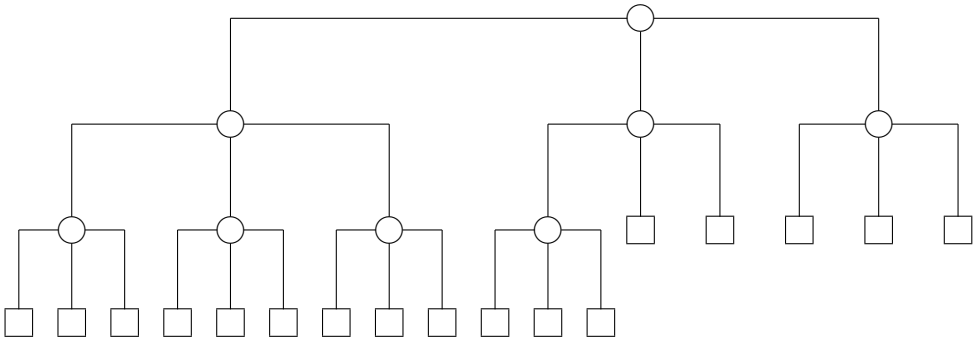
c. *Dimostrare la relazione trovata al punto precedente per induzione su* h*.*

Soluzione.

a. L'albero T_1 è il seguente (le foglie sono rappresentate da quadrati e i nodi interni da cerchi).



L'albero T_2 è il seguente:



Entrambi gli alberi hanno 17 foglie.

b. La (i) non è soddisfatta dall'albero T_1. La (iii), pur essendo soddisfatta sia da T_1 che da T_2, non ha senso in quanto non dipende da n. L'unica che può valere è la (ii).

c. Dimostriamo che $m = (d-1)n+1$ per induzione sull'altezza h dell'albero. La base $h = 0$ è vera in quanto un albero di altezza 0 ha 1 foglia e 0 nodi interni. Fissiamo $h \geq 0$, supponiamo che la relazione sia vera per alberi di altezza sino ad h, e consideriamo un albero T di altezza $h + 1 > 0$. Sia T_i l'i-esimo sottoalbero figlio della radice di T,

$1 \leq i \leq d$, e si indichi con n_i il numero di nodi interni di T_i e con m_i il numero di foglie di T_i. Poichè ogni T_i ha altezza al più h, per ipotesi induttiva vale che $m_i = (d-1)n_i+1$. Inoltre

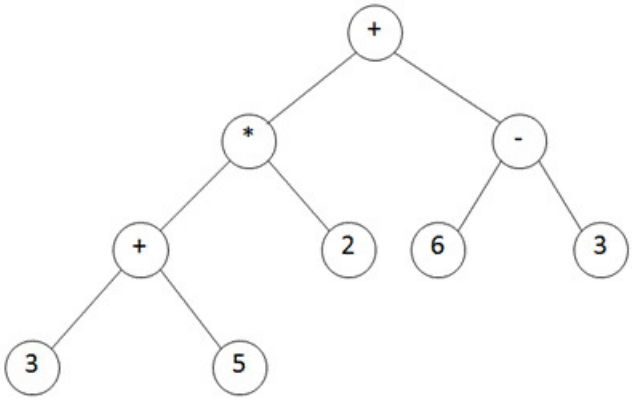
$$\begin{aligned} m &= \sum_{i=1}^{d} m_i \\ n &= 1 + \sum_{i=1}^{d} n_i, \end{aligned}$$

e quindi

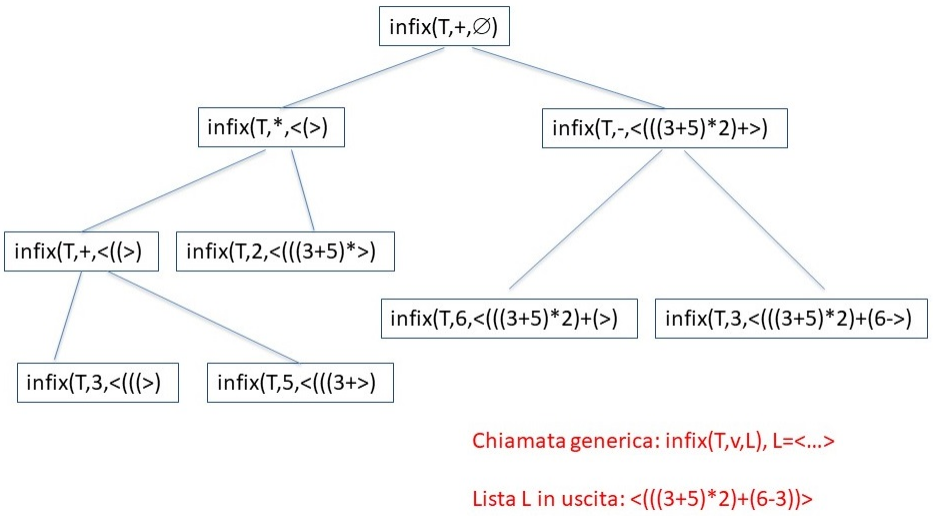
$$m = \sum_{i=1}^{d}((d-1)n_i + 1) = (d-1)n + 1.$$

□

Problema 11 *Disegnare l'albero della ricorsione relativo all'esecuzione di* `infix` *sul seguente Parse Tree, determinando la lista L di uscita.*



Soluzione. L'albero e la lista di uscita sono rappresentati nella seguente figura.



□

Problema 12 *Progettare e analizzare un algoritmo che dato un Parse Tree T restituisce il valore della espressione E associata a T, assumendo che i valori di costanti e variabili siano noti.*

Soluzione. L'algoritmo si basa sullo schema di visita in postorder. Lo pseudocodice è il seguente:

Algoritmo `evaluateExpression(T,v)`
input `nodo v ∈ T, dove T e' il Parse Tree per E`
output `Valore della espressione associata al sottoalbero` T_v

```
if (T.isExternal(v)) then return v.getElement()
else {
  x ← evaluateExpression(T,T.left(v));
  y ← evaluateExpression(T,T.right(v));
  op ← v.getElement();
  return x op y
}
```

Come anticipato, la struttura dell'algoritmo è quella della visita in postorder, e la visita di un nodo v consiste nella lettura di `op` e nel calcolo del valore `x op y`, che assumiamo richieda $\Theta(1)$ operazioni. Quindi la complessità di `evaluateExpression(T,T.root())` è $\Theta(n)$, dove n è il numero di nodi di T, cioè il numero di costanti, variabili e operatori in E. □

Problema 13 *Progettare e analizzare un algoritmo che data una lista che rappresenta una espressione E in notazione postfissa, restituisce il valore di E, assumendo che i valori di costanti e variabili siano noti.*

Soluzione. Considerata la definizione di notazione postfissa, per i calcolo del valore dell'espressione E è sufficiente una scansione della lista che rappresenta l'espressione, appunto in notazione postfissa, e l'uso di una pila di appoggio. Lo pseudocodice è il seguente.

```
Algoritmo evaluatePostfix(E)
input Positional List E che rappresenta un'espressione in notazione postfissa
output Valore della espressione rappresentata da E

S ← Stack vuota; /* contenitore per operatori/costanti/variabili */
p ← E.first();
while (p <> null) do {
  if (p.getElement() = costante/variabile) then S.push(p.getElement());
  else {
    op = p.getElement(); /* operatore rappresentato da p */
    x ← S.pop();
    y ← S.pop();
    S.push(x op y);
  }
  p ← E.after(p);
}
return S.top()
```

Assumendo che `op` sia una operazione elementare, ogni iterazione del **while** richiede $\Theta(1)$ operazioni, e la complessità di `evaluatePostfix`(E) è $\Theta(n)$, dove $n = |E|$, cioè il numero di costanti, variabili e operatoriche compaiono nell'espressione. □

Problema 14 (Esercizio C-8.41 del testo [GTG14]) *Progettare tre algoritmi* iterativi `preorderNext(v)`, `inorderNext(v)` *e* `postorderNext(v)` *che dato un albero binario proprio T e un nodo $v \in T$ restituiscano il nodo visitato dopo v nella visita di T rispettivamente in preorder, inorder e postorder, o* `null`*, se v è l'ultimo nodo visitato, analizzandone la complessità.*

Soluzione. Progettiamo prima l'algoritmo `preorderNext(v)`. Assumiamo che se v è l'ultimo nodo visitato nella vista in preorder di T l'algoritmo restituisca `null`. Si osservi che se v è un nodo interno, allora il suo successore nella visita in preorder è il suo figlio sinisitro. Altrimenti, dobbiamo risalire da v sino a trovare il primo antenato u (cioè l'antenato più profondo) tale che v sta nel sottoalbero sinistro di u. In questo caso, il successore di v nella visita in preorder è il figlio destro di u. Se tale antenato u non esiste, significa che v è la foglia che si incontra scendendo sempre a destra a partire dalla radice di T, ed è quindi l'ultimo nodo visitato dalla vista in preorder di T. In questo caso l'algoritmo restituisce `null`. Lo pseudocodice dell'algoritmo è il seguente:

```
Algoritmo preorderNext(v)
input nodo v ∈ T
output successore di v nella visita in preoder, se esiste, altrimenti null

if (T.isInternal(v)) then return T.left(v)
while (!T.isRoot(v)) do {
  if (v=T.left(T.parent(v))) then return T.sibling(v)
  else v ← T.parent(v)
}
return null
```

Si noti che il numero di iterazioni del **while** è limitato superiormente dalla profondità di v, e che in ciascuna iterazione si esegue un numero costante di operazioni. Per il resto l'algoritmo esegue un numero costante di operazioni. Dato che la profondità di un nodo è minore o uguale all'altezza dell'albero, concludiamo che la complessità è $O(h)$ con h altezza di T.

Consideriamo adesso `inorderNext(v)`. Si osservi che se v è un nodo interno, il suo successore nella visita inorder sarà la foglia più a sinistra nel sottoalbero destro di v. Se invece v è una foglia, il suo successore nella visita inorder sarà il primo antenato incontrato risalendo verso la radice, tale che v è nel suo sottoalbero sinistro. Se tale antenato non esiste, allora v è la foglia più a destra dell'albero ed è quindi l'ultimo nodo visitato. In questo caso, l'algoritmo restituisce `null`. Lo pseudocodice dell'algoritmo è il seguente:

```
Algoritmo inorderNext(v)
input nodo v ∈ T
output successore di v nella visita inorder, se esiste, altrimenti null

if (T.isInternal(v)) then {
  v ← T.right(v)
  while (!T.isExternal(v)) do v ← T.left(v)
  return v
}
while (!T.isRoot(v)) do {
  if (v=T.left(T.parent(v))) then return T.parent(v)
  else v ← T.parent(v)
}
return null
```

Si noti che il numero di iterazioni di ciascuno dei due cicli **while** è limitato dall'altezza h di T, e che in ciascuna iterazione si esegue un numero costante di operazioni. Per il resto l'algoritmo esegue un numero costante di operazioni. Concludiamo quindi che la complessità è $O(h)$. Il progetto e l'analisi di `postorderNext(v)` sono lasciati come esercizio. □

Problema 15 *Si vuole progettare un algoritmo* ricorsivo `heightSum` *per calcolare la somma delle altezze di tutti i nodi di un albero binario proprio T. Detta* `heightSum(T,v)` *la generica invocazione su un nodo $v \in T$, per risolvere il problema si eseguirà* `heightSum(T,T.root())`.

a. *Descrivere tramite pseudocodice* `heightSum(T,v)`*, specificandone con attenzione l'input e l'output.*

b. *Analizzare la complessità di* `heightSum(T,T.root())` *in funzione del numero n di nodi in T.*

Soluzione.

a. L'algoritmo `heightSum(T,v)` invocato su un nodo $v \in T$ restituisce la somma delle altezze dei nodi di T_v e l'altezza di v. Anche in questo caso, calcolando un output più ricco (somma delle altezze e altezza) si ottiene una strategia algoritmica più efficiente. Lo pseudocodice è il seguente:

```
Algoritmo heightSum(T,v)
input Albero T, nodo v ∈ T
output somma delle altezze dei nodi di T_v, altezza di v

if (T.isExternal(v)) then return (0,0)
(sL,hL) ← heightSum(T,T.left(v))
(sR,hR) ← heightSum(T,T.right(v))
h ← max{hL,hR}+1
s ← sL+sR+h
return (s,h)
```

b. Se invocato con $v =$ `T.root()`, l'algoritmo esegue una visita in postorder di T, dove la visita di un nodo interno corrisponde al calcolo delle quantità s e h, basandosi su quelle ottenute dai figli, e quindi richiede tempo $O(1)$. La complessità risulta essere $\Theta(n)$.

□

Problema 16 *Si vuole progettare un algoritmo* ricorsivo `deepLeaf` *per trovare la foglia più profonda in un albero binario proprio T.*

a. *Descrivere tramite pseudocodice la generica invocazione di* `deepLeaf`*, specificandone con attenzione l'input e l'output.*

b. *Analizzare la complessità di* `deepLeaf`*.*

Soluzione.

a. Detta `deepLeaf(T,v)` la generica invocazione su un nodo $v \in T$, per risolvere il problema si eseguirà `deepLeaf(T,T.root())`. Si tenga presente che per ogni sottoalbero T_v la profondità (in T_v) della sua foglia più profonda è pari all'altezza di T_v, e la foglia più profonda in T_v sarà la foglia più profonda nel sottoalbero sx o dx di altezza massima. È quindi opportuno seguire lo schema della visita in postorder chiedendo che `deepLeaf(T,v)` restituisca oltre alla foglia w più profonda in T_v anche l'altezza di T_v (ovvero la profondità di w in T_v). Ancora una volta, calcolando un'informazione più ricca (foglia più profonda e altezza) si ottiene una strategia algoritmica più efficiente. Lo pseudocodice è il seguente:

Algoritmo `deepLeaf(T,v)`
input `Albero T, nodo v` $\in$ `T`
output `Foglia` $w \in T_v$ `piu' profonda e altezza` h `di` T_v

```
if (T.isExternal(v)) then return (v,0)
(wL,hL) ← deepLeaf(T,T.left(v))
(wR,hR) ← deepLeaf(T,T.right(v))
if (hL > hR) then {w ← wL; h ← hL+1}
else {w ← wR; h ← hR+1}
return (w,h)
```

b. Se invocato con `v=T.root()`, l'algoritmo esegue una visita in postorder di T, dove la visita di un nodo interno corrisponde alla determinazione di w e h, basandosi sulle coppie `(wL,hL)` e `(wR,hR)` ottenute dai figli, e quindi richiede tempo $\Theta(1)$. La complessità risulta essere $\Theta(n)$, dove n è il numero di nodi di T.

□

Problema 17 *Sia T un albero binario proprio dove ogni nodo $v \in T$ contiene un valore binario. Un nodo $v \in T$ si dice* 3-balanced*, se $|n_0(v) - n_1(v)| \leq 3$, dove $n_0(v)$ e $n_1(v)$ denotano il numero di 0 ($n_0(v)$) e 1 ($n_1(v)$) nel sottoalbero T_v. Si progetti in pseudocodice un algoritmo ricorsivo* `All3Balanced` *per determinare se* tutti i nodi di T sono 3-balanced*, con complessità lineare nel numero di nodi di T.*

Soluzione. Sia `All3Balanced`(T, v) l'invocazione generica dell'algoritmo su un nodo $v \in T$. Per determinare se tutti i nodi di T sono 3-balanced, sin invocherà `All3Balanced`(T, v) con $v =$ `T.root()`. Lo pseudocodice è il seguente

Algoritmo `All3Balanced(T,v)`
input `Albero T, nodo v` $\in$ `T`
output $n_0(v)$, $n_1(v)$, `booleano` b_v `che vale ''true'' se tutti i nodi di` T_v
`sono 3-balanced, e ''false'' altrimenti`
if $(v$`.getElement()` $= 0)$ **then** $\{n_0(v) \leftarrow 1;\ n_1(v) \leftarrow 0\}$
else $\{n_0(v) \leftarrow 0;\ n_1(v) \leftarrow 1\}$;
if $(T$`.isExternal`$(v))$ **then return** $(n_0(v), n_1(v),$ `true`$)$;
$(n_{0,L}, n_{1,L}, b_L) \leftarrow$ `All3Balanced`$(T, T$`.left`$())$;
$(n_{0,R}, n_{1,R}, b_R) \leftarrow$ `All3Balanced`$(T, T$`.right`$())$;
$n_0(v) \leftarrow n_0(v) + n_{0,L} + n_{0,R}$;
$n_1(v) \leftarrow n_1(v) + n_{1,L} + n_{1,R}$;
if $(|n_0(v) - n_1(v)| \leq 3)$ **then return** $(n_0(v), n_1(v), (b_L$ `AND` $b_R))$
else return $(n_0(v), n_1(v),$ `false`$)$

Se invocato con $v =$ `T.root()`, l'algoritmo esegue una visita in postorder di T, dove la visita di un nodo interno corrisponde al calcolo delle quantità $n_0(v), n_1(v)$ e del valore del boolea no da restituire, e quindi richiede tempo $O(1)$. La complessità risulta essere $\Theta(n)$. □

Problema 18 *Un* albero di sensori *è un albero binario proprio T i cui nodi rappresentano* sensori. *Ciascun sensore $v \in T$ ha un flag v*`.ON` *che vale 1 se il sensore è acceso, e 0 se è spento. Si vuole progettare un algoritmo ricorsivo* `maxOnHeight` *che, dato un albero di sensori T, restituisca la* massima altezza di un sensore acceso*. Se tutti i sensori sono spenti, l'algoritmo restituisce* `-1`*.*

a. *Descrivere tramite pseudocodice la generica invocazione di* `maxOnHeight`*, specificandone con attenzione l'input e l'output.*

b. *Analizzare la complessità di* `maxOnHeight`*.*

Soluzione.

a. Detta `maxOnHeight(T,v)` la generica invocazione su un nodo $v \in T$, per risolvere il problema si eseguirà `maxOnHeight(T,T.root())`. Basiamo l'algoritmo sullo schema di visita in postorder.

```
Algoritmo maxOnHeight(T,v)
input Albero binario proprio T, nodo v ∈ T
output (mv,hv):  mv = max altezza di un sensore acceso in T_v
        (-1 se T_v non ha sensori accesi); hv = altezza di T_v

if (T.isExternal(v)) then
  if (v.ON=1) then return (0,0) else return (-1,0);
(mL,hL) ← maxOnHeight(T,T.left(v));
(mR,hR) ← maxOnHeight(T,T.right(v));
h ← max{hL,hR}+1;
m ← max{mL,mR};
if (v.ON=1) then return (h,h) else return (m,h)
```

b. Se invocato con $v =$ `T.root()`, l'algoritmo esegue una visita in postorder di T, dove la visita di un nodo richiede tempo $O(1)$. La complessità risulta essere $O(n)$, dove n è il numero di nodi di T.

□

Problema 19 *Sia T un albero binario proprio dove ogni nodo $v \in T$ memorizza un valore v.*`val` *che può essere 1 o -1. Un nodo $v \in T$ si dice* strong *se la somma dei valori memorizzati nei nodi del sottoalbero T_v è > 0. Progettare un algoritmo* `countStrong` *per contare il numero di nodi strong in un tale albero T, e analizzarne la complessità.*

Soluzione. Scriviamo un algoritmo ricorsivo `countStrong` che dato T e un nodo $v \in T$ restituisce il numero di nodi strong in T_v, la somma dei valori T_v. Il numero di nodi strong di T si otterrà invocando l'algoritmo con $v =$ T.root(). Ancora una volta, calcolando un'informazione più ricca (numero di nodi strong e somma dei valori) si ottiene una strategia algoritmica più efficiente. Lo pseudocodice dell'algoritmo è il seguente:

```
Algoritmo countStrong(T,v)
input Albero T, nodo v ∈ T
output count = numero di nodi strong in T_v, sum = somma dei valori T_v

if (T.isExternal(v)) then
  if (v.val=1) then return (1,1) else return (0,-1)
(cL,sL) ← countStrong(T,T.left(v))
(cR,sR) ← countStrong(T,T.right(v))
sum ← sL+sR+v.val
if (sum>0) then (count ← cL+cR+1) else (count ← cL+cR)
return (count,sum)
```

Se invocato con $v =$ `T.root()`, l'algoritmo esegue una visita in postorder di T, dove la visita di un nodo interno corrisponde al calcolo delle quantità `sum` e `count`, basandosi su quelle ottenute dai figli e sul valore del nodo, e quindi richiede tempo $O(1)$. La complessità risulta quindi la stessa della visita in postorder, ovvero $\Theta(n)$, dove n è il numero di nodi di T. □

Problema 20 (Adattamento dall'esercizio C-8.52 del testo [GTG14]) *Sia T un albero binario proprio con n nodi. Il* **diametro** *di T è definito come la massima distanza tra due nodi, dove la distanza tra due nodi u e v è la somma delle loro profondità nel sottoalbero con radice $LCA(u, v)$. Progettare e analizzare un algoritmo* ricorsivo *efficiente per calcolare il diametro di un albero.* (**Suggerimento:** *è facile vedere che, se $n > 1$, il diametro di T si ottiene come massimo tra tre quantità: il diametro del sottoalbero sinistro, il diametro del sottoalbero destro, e la somma $d_1 + d_2 + 2$, dove d_1 è la massima profondità di una foglia nel sottoalbero sinistro, e d_2 è la massima profondità di una foglia nel sottoalbero destro.*)

Soluzione. Scriviamo un algoritmo ricorsivo `diametro` che dato T e un nodo $v \in T$ restituisce il diametro di T_v e la massima profondità di una foglia di T_v. Il diametro di T si otterrà invocando l'algoritmo con $v =$ T.root(). Ancora una volta, calcolando un'informazione più ricca (diametro e max profondità di una foglia) si ottiene una strategia algoritmica più efficiente. Lo pseudocodice dell'algoritmo è il seguente:

Algoritmo `diametro(T,v)`
input `Albero T, nodo v` $\in$ `T`
output `diametro di` T_v `e max profondita' di una foglia di` T_v

```
if (T.isExternal(v)) then return (0,0)
else {
  (diam1,dmax1) ← diametro(T,T.left(v))
  (diam2,dmax2) ← diametro(T,T.right(v))
  diam ← max{diam1,diam2,dmax1+dmax2+2}
  dmax ← max{dmax1,dmax2}+1
  return (diam,dmax)
}
```

Se invocato con $v =$ `T.root()`, l'algoritmo esegue una visita in postorder di T, dove la visita di un nodo interno corrisponde al delle quantità `diam` e `dimax`, basandosi su quelle ottenute dai figli e sul valore del nodo, e quindi richiede tempo $O(1)$. La complessità risulta quindi la stessa della visita in postorder, ovvero $\Theta(n)$, dove n è il numero di nodi di T. □