Dati e Algoritmi I (Pietracaprina)

Esercizi sugli Alberi

Problema 1 *Progettare un algoritmo che dato un albero T, per ciascun nodo $v \in T$ memorizzi la sua profondità in un campo* `v.depth`*, e analizzarne la complessità in funzione del numero di nodi n dell'albero.*

Soluzione. La soluzione si trova nei lucidi sugli Alberi Generali (Lucidi 32 e 33). □

Problema 2 *Si supponga di calcolare l'altezza di un albero T di n nodi invocando l'algoritmo* `depth`(T, 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 $\Omega(n^2)$. (È l'algoritmo* `heightBad` *decritto in [GTG14]. Si veda anche l'esercizio C-8.27 del testo.)*

Soluzione. La soluzione si trova nei lucidi sugli Alberi Generali (Lucidi 38 e 39). □

Problema 3 *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 una semplice modifica dell'algoritmo `height(T,v)` 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 `albero T, nodo v` $\in$ `T`
output `altezza di` v `e impostazione del campo height di tutti i nodi 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`$(T,T.\text{root}())$. La complessità è la stessa dell'algoritmo `height`, ovvero $\Theta(n)$ □

Problema 4 (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* (LCA(v, w)) *come l'antenato comune più profondo. Progettare un algoritmo efficiente per calcolare* 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 Albero T, nodi v,w ∈ T
output LCA(v,w)

d_v ← depth(T,v); d_w ← depth(T,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, ad esempio, 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 5 *Si supponga che la visita in preorder di un albero ordinato di 6 nodi incontri i nodi nell'ordine ABCDEF.*

a. *Dire quali delle seguenti sequenze può rappresentare la visita in postorder dello stesso albero (motivando la risposta): BAFECD, CDBFEA, CDAEFB.*

b. *Disegnare l'albero compatibile con le due sequenze di preorder e postorder.*

Soluzione.

a. 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.

b. 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 6 *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 $d \geq 2$ il numero di figli della radice. Per $1 \leq i \leq d$ 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

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

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

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

in quanto $d \geq 2$. □

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 *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?*

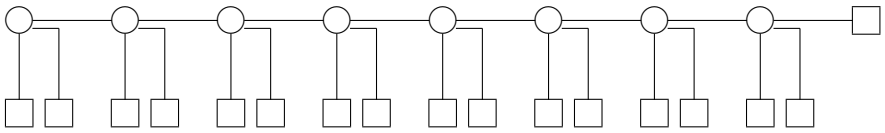
b. *Sia T un albero d-ario proprio di altezza h con n nodi interni ed m foglie. Usando gli esempi precedenti e il buon senso trovare l'unica tra le seguenti relazioni che può valere per T arbitrario e $d \geq 2$:*

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

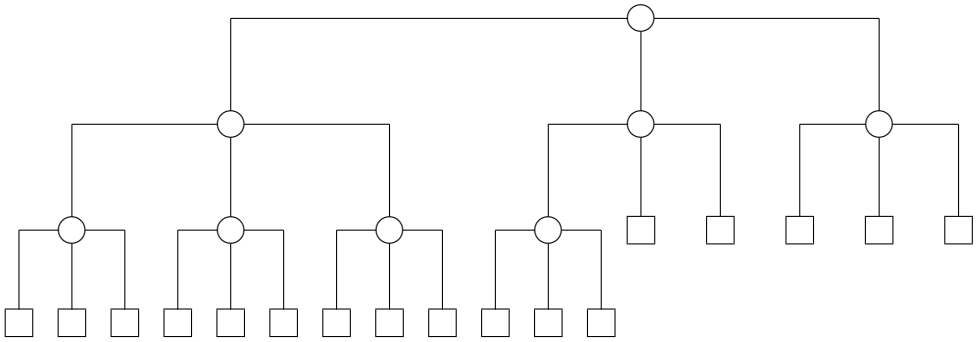
c. Dimostrare la relazione trovata al punto precedente per induzione su h.

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 9 (Esercizio C-8.41 del testo [GTG14]) *Progettare tre algoritmi* `preorderNext(T,v)`, `inorderNext(T,v)` *e* `postorderNext(T,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 Albero T, 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 Albero T, 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 10 *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'informazione più ricca (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 $O(n)$, dove n è il numero di nodi di T.

Osservazione. È un esercizio importante che mostra come a volte per ottenere una strategia algoritmica efficiente sia opportuno *calcolare un'informazione più ricca di quella richiesta* (ad es., l'altezza oltre alla somma delle altezze). □

Problema 11 *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. (Si ricordi che T_v è il sottoalbero con radice v e include v.) Progettare un algoritmo ricorsivo per contare il numero di nodi strong in T, analizzandone 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 sommadei 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 12 (Adattamento dall'esercizio C-8.52 del testo [GTG14]) *Sia T un albero binario, non necessariamente proprio, con n nodi. Il* **diametro** *di T è definito come la massima distanza tra due nodi (la distanza tra due nodi u e v è la somma delle loro profondità nel sottoalbero con radice $LCA(u, v)$). È facile vedere che il diametro di T si ottiene come massimo tra tre quantità: il diametro del sottoalbero sinistro, il diametro del sottoalbero destro, e la somma $d_x + d_y$, dove d_x è la massima profondità di una foglia appartenente al sottoalbero sinistro, e d_y è la massima profondità di una foglia appartenente al sottoalbero destro. (Se uno dei due sottoalberi non esiste, il diametro di tale sottoalbero e la massima profondità di una sua foglia si assumono pari 0.) Basandosi su quest'ultima osservazione, progettare un algoritmo efficiente per calcolare il diametro di un albero, e analizzarne la complessità. Per comodità, si assuma che se un nodo interno ha solo un figlio, tale figlio è quello sinistro.*

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 if (T.right(v) = null) then {
  (diam,dmax) ← diametro(T,T.left(v))
  diam ← max{diam,dmax+1}; dmax ← dmax+1;
  return (diam,dmax)
}
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. □

Problema 13 *Si vuole progettare un algoritmo* ricorsivo `deepLeaf` *per trovare la foglia più profonda in un albero binario proprio* T. *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.

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

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

Soluzione. La soluzione verrà presentata durante la lezione del 14 novembre 2019. □

Problema 14 *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.liveAncestors *il numero di antenati* alive *di* v. *Detta* `allLiveAncestors(T,v)` *la generica invocazione su un nodo* $v \in T$, *per risolvere il problema si eseguirà* `allLiveAncestors(T,T.root())`.

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

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

Soluzione. La soluzione verrà presentata durante la lezione del 14 novembre 2019. □