Alberi – Albero livello–valore

Scrivere un algoritmo che preso in input un albero binario T i cui nodi sono associati ad un valore intero $T.value$, restituisca il numero di nodi dell'albero il cui valore è uguale al livello del nodo.

Vi ricordo che il livello del nodo è pari al numero di archi che devono essere attraversati per raggiungere il nodo dalla radice. Per cui la radice ha livello 0, i suoi figli hanno livello 1, etc.

Alberi – Cammino radice–discendente crescente

Dato un albero binario contenente interi, scrivere un algoritmo che restituisca la lunghezza del più lungo cammino monotono crescente radice-discendente, dove:

- il discendente non è necessariamente foglia;
- con lunghezza si intende il numero totale di archi attraversati;
- con monotona crescente si intende che i valori contenuti nei nodi della sequenza devono essere ordinati in senso crescente da radice a discendente.

Discuterne correttezza e complessità.

Alberi – Grado di sbilanciamento

Si consideri un albero binario T:

- Il grado di sbilanciamento di un nodo v è pari alla differenza, in valore assoluto, fra il numero di foglie presenti nel sottoalbero sinistro di v e quelle presenti nel sottoalbero destro di v.
- Il grado di sbilanciamento dell'albero T è pari al massimo grado di sbilanciamento dei nodi di T.

Scrivere un algoritmo che dato un albero T, restituisca il grado di sbilanciamento dell'albero. Discuterne correttezza e complessità.

Nota: In pseudocodice, è possibile restituire una coppia di valori:

(int, int) fun1(TREE T)

```
[...]
return (a, b)
```

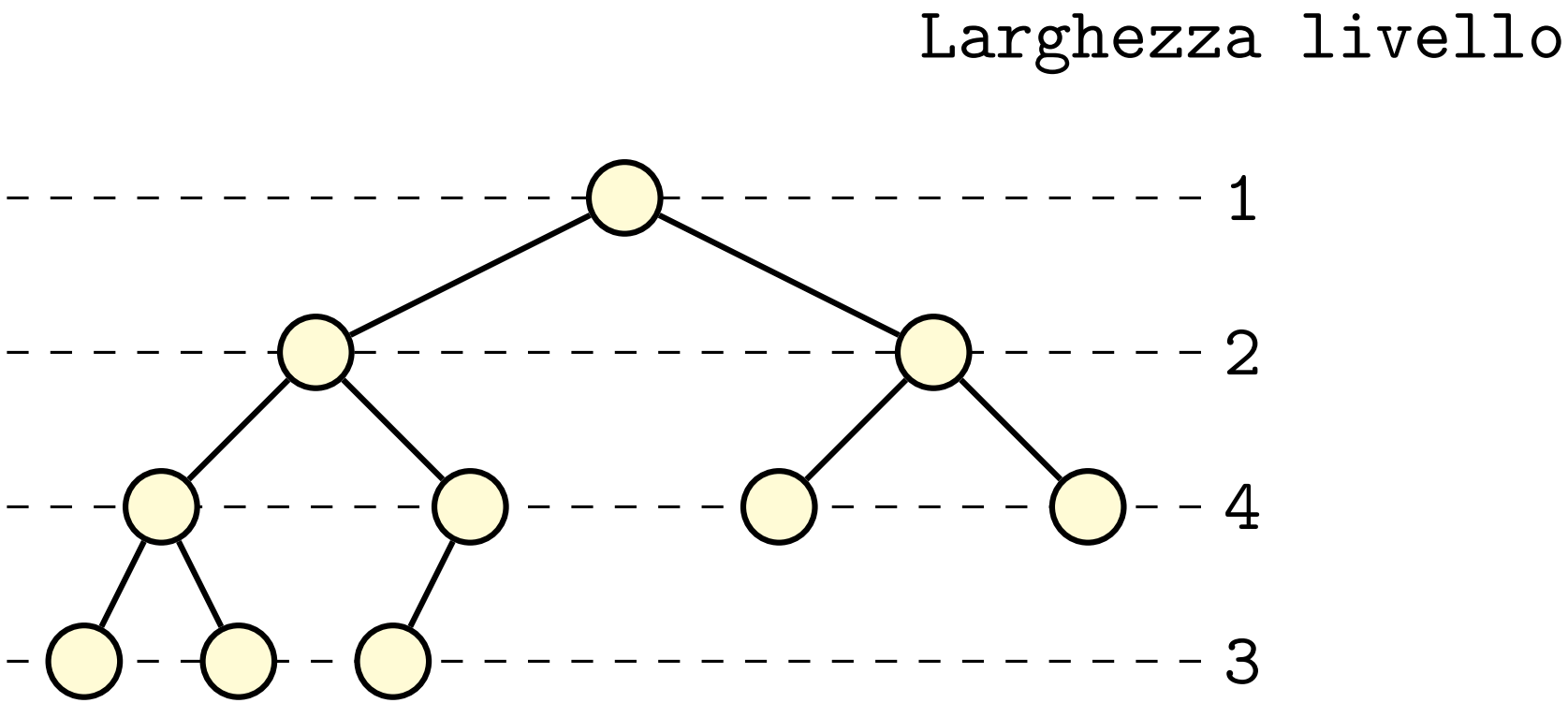
int fun2(TREE T)

```
int, int x, y = fun1(TREE T)
return y
```

Alberi – Larghezza albero

La larghezza di un albero T è il numero massimo di nodi di T che stanno tutti al medesimo livello.

Scrivere un algoritmo che restituisca la larghezza di un albero ordinato T contenente n nodi.



SortinoSort

Il professor Sortino ha inventato un nuovo algoritmo di ordinamento.

- Il vettore di input viene diviso in tre parti, di dimensioni circa $n/3$.
- Vengono ordinati ricorsivamente i primi due terzi, i secondi due terzi, e infine di nuovo i primi due terzi.

SortinoSort(**int**[ ] A, **int** i, **int** j)

---

if $j - i + 1 \leq 6$ **then**
    InsertionSort(A, i, j)
else
    int $s = \lceil (j - i + 1)/3 \rceil$
    SortinoSort$(A, i, i + 2s - 1)$
    SortinoSort$(A, i + s, j)$
    SortinoSort$(A, i, i + 2s - 1)$

SortinoSort

1. Qual è la complessità di questo algoritmo? Il Prof. Sortino finirà nella prossima edizione del mio libro?
2. (Difficile, Opzionale) Dimostrare per induzione che questo algoritmo è corretto. Per comodità, assumete pure che tutti i valori siano distinti.

Spoiler alert!

Albero livello-valore

Una semplice visita posticipata risolve il problema. La complessità è ovviamente $O(n)$.

---

int sameLevel(TREE T)

---

return sameLevelRec(T, 0)

---

int sameLevelRec(TREE T, **int** ℓ)

```
int tot = 0
if T ≠ nil then
    tot = sameLevelRec(T.right(), ℓ + 1) + sameLevelRec(T.left(), ℓ + 1)
    if T.value == ℓ then
        tot = tot + 1
return tot
```

Alberi – Percorso cammino-discendente

```
int monotone(Tree T)
int maxl = 0
int maxr = 0
if T ≠ nil then
    if T.left() ≠ nil and T.left().value > T.value then
        maxl = 1 + monotone(T.left())
    if T.right() ≠ nil and T.right().value > T.value then
        maxr = 1 + monotone(T.right())
return max(maxl, maxr)
```

Alberi - Grado di sbilanciamento

int unbalance(TREE T)

int, **int** $leafs, max =$ unbalanceRec(TREE T)
return max

(**int**, **int**) unbalanceRec(TREE T)

if T == **nil then**
  return $(0, 0)$
if $T.left$ == **nil and** $T.right$ == **nil then**
  return $(1, 0)$
int, **int** $L_{leafs}, L_{max} =$ unbalance$(T.left)$
int, **int** $R_{leafs}, R_{max} =$ unbalance$(T.right)$
return $(L_{leafs} + R_{leafs}, \max(L_{max}, R_{max}, |L_{leafs} - R_{leafs}|))$

Larghezza (1)

int breadth(TREE t)

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

Larghezza (2)

```
int breadth(Tree t)
int count = 1          % # nodi nel livello corrente da visitare; radice
int breadth = 1        % Massima larghezza trovata finora; 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()
        breadth = max(breadth, count)
return breadth
```

SortinoSort – 12/09/10

La complessità è rappresentata dalla seguente equazione di ricorrenza:

$$T(n) = \begin{cases} 3T(\frac{2}{3}n) + 1 & n > 6 \\ 1 & n \leq 6 \end{cases}$$

Utilizzando il teorema delle ricorrenze lineari con partizione bilanciata, si ottiene che $a = 3$, $b = 3/2$, da cui $\alpha = \log_{3/2} 3$; inoltre, $\beta = 0$. Siamo quindi nel caso $T(n) = n^{\alpha}$.
Non avete una calcolatrice e non sapete quanto sia $\log_{3/2} 3$?

SortinoSort – 12/09/10

La complessità è rappresentata dalla seguente equazione di ricorrenza:

$$T(n) = \begin{cases} 3T(\frac{2}{3}n) + 1 & n > 6 \\ 1 & n \leq 6 \end{cases}$$

Utilizzando il teorema delle ricorrenze lineari con partizione bilanciata, si ottiene che $a = 3$, $b = 3/2$, da cui $\alpha = \log_{3/2} 3$; inoltre, $\beta = 0$. Siamo quindi nel caso $T(n) = n^{\alpha}$.
Non avete una calcolatrice e non sapete quanto sia $\log_{3/2} 3$?

E' semplice: $\left(\frac{3}{2}\right)^2 = \frac{9}{4} < 3$, mentre $\left(\frac{3}{2}\right)^3 = \frac{27}{8} > 3$. Quindi α è compreso fra 2 e 3, e quindi questo algoritmo è addirittura peggiore di Insertion Sort. Il prof. Sortino non finirà nel mio libro.