Domanda 28 Sia T un albero binario i cui nodi x hanno i campi *x.left*, *x.right*, *x.key*. L'albero si dice un *sum-heap* se per ogni nodo x, la chiave di x è maggiore o uguale sia alla somma delle chiavi nel sottoalbero sinistro che alla somma delle chiavi nel sottoalbero destro.

Scrivere una funzione `IsSumHeap(T)` che dato in input un albero T verifica se `T` è un sum-heap e ritorna un corrispondente valore booleano. Valutarne la complessità.

Soluzione: La soluzione può essere

```
IsSumHeap(T)
  return IsSumHeap-rec(T.root)

IsSumHeap-rec(x) // verifica se l'albero radicato in x e' un sum-heap
                    e ritorna true/false e la somma delle chiavi nel
                    sottoalbero radicato in x
  if x = nil
     return true, 0
  else
     isSumHeapL, sumL = IsSumHeap-rec(x.left)
     isSumHeapR, sumR = IsSumHeap-rec(x.right)
     return (x.key >= sumL) and (x.key >= sumR),
            x.key + sumL + sumR
```

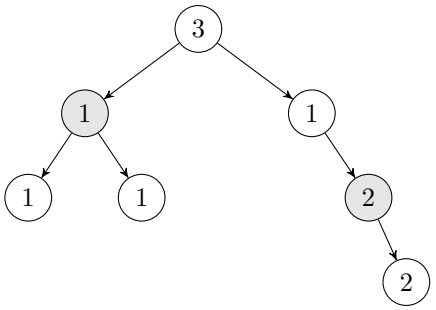
Per quanto riguarda la complessità, se l'albero è bilanciato, $T(n) = c + 2T(n/2)$ per un'opportuna costante c e quindi, utilizzando il master theorem, si deduce che la complessità è $\Theta(n)$. Se l'albero non è bilanciato, si può osservare che si tratta di una visita (costo lineare) o più precisamente scrivere la ricorrenza

$$T(n) = T(k) + T(n-k-1) + c$$

e provare, per sostituzione, che $T(n) = an + b$ è soluzione per opportune costanti a, b.

Esercizio 10 Un nodo x di un albero binario T si dice *fair* se la somma delle chiavi nel cammino che conduce dalla radice dell'albero al nodo x (escluso) coincide con la somma delle chiavi nel sottoalbero di radice x (con x incluso). Realizzare un algoritmo ricorsivo *printFair(T)* che dato un albero T stampa tutti i suoi nodi fair. Supporre che ogni nodo abbia i campi *x.left*, *x.right*, *x.p*, *x.key*. Valutare la complessità dell'algoritmo.

Un esempio: i nodi grigi sono fair



Soluzione: L'algoritmo può essere il seguente:

```
printFair(x,path)       // x = node of the tree
                        // path=sum of the keys in the path from the root to x
```

```
                        // Action: print the fair nodes and
                       //         returns the sum of the keys in the subtree
   if (x == nil)
      return 0

   left   = printFair(x.l, path + x.key)
   right  = printFair(x.r, path + x.key)
   sumTree = left + right + x.key
   if (path == sumTree)
      print x
   return sumTree
```

e viene chiamato come `printFair(T.root, 0)`.

Si tratta di una visita, quindi con costo $O(n)$ (più precisamente ottenibile con il master theorem come soluzione della ricorrenza $T(n) = 2T(n/2) + c$.

Esercizio 11 Sia dato un albero i cui nodi contengono una chiave intera $x.key$, oltre ai campi $x.l$, $x.r$ e $x.p$ che rappresentano rispettivamente il figlio sinistro, il figlio destro e il padre. Si definisce *grado di squilibrio* di un nodo il valore assoluto della differenza tra la somma delle chiavi nei nodi foglia del sottoalbero sinistro e la somma delle chiavi dei nodi foglia del sottoalbero destro. Il grado di squilibrio di un albero è il massimo grado di squilibrio dei suoi nodi.

Fornire lo pseudocodice di una funzione `sdegree(T)` che calcola il grado di squilibrio dell'albero T (si possono utilizzare funzioni ricorsive di supporto). Valutare la complessità della funzione.

Soluzione:

```
// computes the sum of the leaf nodes of the subtree and the sdegree
// for the node x (returns two values)

sdegree(x)

  if (x == nil)
     sum = 0
     degree = 0
  elif (x.left == nil) and (x.right = nil)      # leaf
     sum = x.key
     degree = 0
  else
     suml, degreel = sdegree(x.l)
     sumr, degreer = sdegree(x.r)
     sum = suml + sumr
     degree = max { degreel, degreel, abs(suml - sumr) }

  return sum, degree
```

Esercizio 12 Si consideri un albero binario T, i cui nodi x hanno i campi $x.l$, $x.r$, $x.p$ che rappresentano il figlio sinistro, il figlio destro e il padre, rispettivamente. Un *cammino* è una