Algoritmi e Strutture Dati
10 Settembre 2020

Note

1. La leggibilità è un prerequisito: parti difficili da leggere potranno essere ignorate.
2. Quando si presenta un algoritmo è fondamentale spiegare l'idea e motivarne la correttezza.
3. L'efficienza e l'aderenza alla traccia sono criteri di valutazione delle soluzioni proposte.
4. Si consegna la scansione dei fogli di bella copia, e come ultima pagina un documento di identità.

Domande

Domanda A (8 punti) Si consideri la seguente funzione ricorsiva con argomento un intero $n \geq 0$

```
exp(n)
   if n = 0
      return 0
   else
      return exp(n-1) + exp(n-1) + 1
```

Dimostrare induttivamente che la funzione calcola il valore $2^n - 1$. Inoltre, determinare la ricorrenza che esprime la complessità della funzione e mostrare che la soluzione è $\Omega(2^n)$. È anche $O(2^n)$? Motivare le risposte.

Soluzione: Per quanto riguarda la funzione calcolata, procediamo per induzione su n. Se $n = 0$, la funzione ritorna 0 che è in effetti $2^0 - 1 = 1 - 1 = 0$. Se $n > 0$, allora per ipotesi induttiva, $exp(n-1) = 2^{n-1} - 1$. Quindi $exp(n) = exp(n-1) + exp(n-1) + 1 = 2^{n-1} - 1 + 2^{n-1} - 1 + 1 = 2 \cdot 2^{n-1} - 1 = 2^n - 1$, come desiderato.

Per quanto riguarda la ricorrenza, nel caso ricorsivo, si effettuano due chiamate ricorsive con argomento $n-1$ e tre somme (operazioni di tempo costante) si ottiene quindi

$$T(n) = 2\,T(n-1) + c$$

È facile dimostrare con il metodo di sostituzione che $T(n) = \Omega(2^n)$, ovvero che esistono $d > 0$ e n_0 tali che $T(n) \geq d2^n$, per $n \geq n_0$. Si procede per induzione su n:

$$\begin{aligned}
T(n) &= 2\,T(n-1) + c && \text{[dalla definizione della ricorrenza]}\\
&\geq 2d2^{n-1} + c && \text{[ipotesi induttiva]}\\
&= d2^n + c\\
&\geq d2^n
\end{aligned}$$

qualunque sia $d > 0$ e n.

Vale anche $T(n) = O(2^n)$. Tuttavia, il tentativo di provare direttamente per induzione che $d > 0$ e n_0 tali che $T(n) \leq d2^n$ per un'opportuna costante $d > 0$ e $n \geq n_0$ fallisce. Si riesce invece a dimostrare che $T(n) \leq d(2^n - 1)$. Infatti:

$$\begin{aligned}
T(n) &= 2\,T(n-1) + c && \text{[dalla definizione della ricorrenza]}\\
&\leq 2d(2^{n-1} - 1) + c && \text{[ipotesi induttiva]}\\
&= d2^n - 2d + c\\
&= d(2^n - 1) - d + c\\
&\leq d2^n
\end{aligned}$$

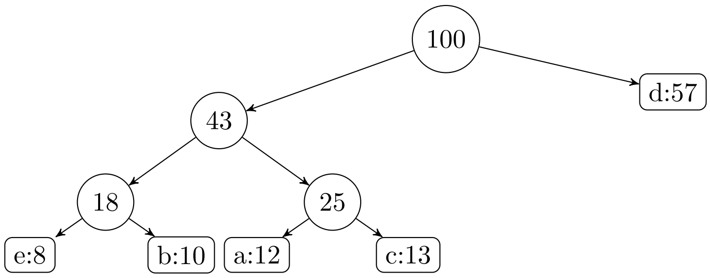
dove l'ultima disuguaglianza vale se $d \geq c$. Quindi $T(n) = O(2^n - 1) = O(2^n)$.

Domanda B (5 punti) Indicare, in forma di albero binario, il codice prefisso ottenuto tramite l'algoritmo di Huffmann per l'alfabeto $\{a, b, c, d, e\}$, supponendo che ogni simbolo appaia con le seguenti frequenze:

a	b	c	d	e
12	10	13	57	8

Spiegare il processo di costruzione del codice.

Soluzione: Per la descrizione del processo di costruzione, si rimanda al libro. Il risultato è il seguente:



Esercizi

Esercizio 1 (9 punti) Realizzare una funzione $append(T, x, z)$ che dato un albero binario di ricerca T, un suo nodo *foglia* x ed un nuovo nodo z, verifica se è possibile inserire z come figlio (destro o sinistro) di x in T. In caso affermativo ritorna l'albero esteso, altrimenti ritorna `nil`. (Assumere che i nodi dell'albero abbiano gli usuali campi $x.left$, $x.right$, $x.p$, $x.key$ e la radice sia $T.root$.) Valutarne la complessità.

Soluzione: Affinchè z possa essere inserito come figlio destro o sinistro di z, occorre in primo luogo che la chiave di z sia $\leq$ delle chiavi degli antenati di z che hanno x e quindi z nel sottoalbero sinistro e $\geq$ delle chiavi degli antenati di x che hanno x nel sottoalbero destro.

Quindi risalendo tali antenati, si verifica che la condizione sia soddisfatta. Se questo accade e $z.key$ è $\leq x.key$ potrà essere inserito come figlio sinistro di x, altrimenti come figlio destro. Altrimenti z non può essere figlio di x e si ritorna nil.

Lo pseudocodice può essere il seguente:

```
append(T,x,z)

    # verifica che z.key sia <= delle chiavi degli antenati di x
    # per i quali x e' nel sottoalbero dx e
    y = x

    while (y.p <> nil) and
          ( (y == y.p.left   and  z.key <= y.p.key) or
            (y == y.p.right  and  z.key >= y.p.key) )
        y = y.p

    # se siamo arrivati alla radice, significa che la condizione e' soddisfatta
    if (y.p == nil)
        # e z viene inserito come figlio dx o sx
        z.p = x
        if (z.key <= x.key)
            x. left = z
        else
            x.right = z

        return T
    else                                 # altrimenti si ritorna nil
        return nil
```

La complessità è chiaramente $O(h)$, dove h è l'altezza dell'albero. Infatti, il ciclo while effettua un numero di iterazioni pari al numero di antenati di x, e quindi limitato dall'altezza dell'albero. E ciascuna iterazione ha costo costante.

Esercizio 2 (9 punti) Sia $n > 0$ un intero. Si consideri la seguente ricorrenza $M(i,j)$ definita su tutte le coppie (i,j) con $1 \leq i \leq j \leq n$:

$$M(i,j) = \begin{cases} 1 & \text{se } i = j, \\ 2 & \text{se } j = i+1, \\ M(i+1,j-1) \cdot M(i+1,j) \cdot M(i,j-1) & \text{se } j > i+1. \end{cases}$$

1. Scrivere una coppia di algoritmi INIT_M(n) e REC_M(i,j) per il calcolo memoizzato di $M(1,n)$.
2. Calcolare il numero esatto $T(n)$ di moltiplicazioni tra interi eseguite per il calcolo di $M(1,n)$.

Soluzione:

1. Pseudocodice:

```
INIT_M(n)
if n=1 then return 1
if n=2 then return 2
for i=1 to n-1 do
   M[i,i] = 1
   M[i,i+1] = 2
M[n,n] = 1
for i=1 to n-2 do
   for j=i+2 to n do
      M[i,j] = 0
return REC_M(1,n)


REC_M(i,j)
if M[i,j] = 0 then
   M[i,j] = REC_M(i+1,j-1) * REC_M(i+1,j) * REC_M(i,j-1)
return M[i,j]
```

2.

$$T(n) = \sum_{i=1}^{n-2} \sum_{j=i+2}^{n} 2 = 2 \sum_{i=1}^{n-2} n - i - 1 = 2 \sum_{k=1}^{n-2} k = (n-2)(n-1)$$