Test Person

Email: testperson@example.com
Class: E1a
Teacher: Professor Teacher

2020-06-22

2020-06-11 am Esame Introduzione agli Algoritmi [Caminiti]

Word count: 1140

Introduzione agli algoritmi

Proff. T. Calamoneri – S. Caminiti - E. Fachini

11 Giugno 2020 - Test A (mattina)

Soluzioni (prof. S. Caminiti)

Testo scritto direttamente su Exam.net (con i suoi limiti e errori di formattazione)

Es 1. *Si dimostri, utilizzando la definizione di Θ, che*

$$f(n) = n^3 + lg^3 (n^3) = \Theta(n^3)$$

mettendo in evidenza e commentando con chiarezza i passi seguiti.

Dimostriamo separatamente O e Omega

Th. f(n) = O(n) ovvero Esistono c>0, n0>=0 t.c. f(n)<=cn per ogni n>=n0

dim.

$n^3 + lg^3 (n^3) <= cn^3$

$n^3 + 27 lg^3(n) <= cn^3$

impongo c = 28

$n^3 + 27 lg^3(n) <= 28n^3$

$27 lg^3(n) <= 27n^3$

$lg^3(n) <= n^3$

lg(n) <= n

vero per ogni n >= n0=0

qed.

Nota: $lg^3 (n^3) = lg(n^3) * lg(n^3) * lg(n^3) = 3lg(n) * 3lg(n) * 3lg(n) = \mathbf{27}lg^3(n)$

Th. f(n) = Omega(n) ovvero Esistono c>0, n0>=0 t.c. f(n)>=cn per ogni n>=n0

dim.

scelgo c=1

$n^3 + lg^3 (n^3) >= n^3$

$27lg^3 (n) >= 0$

$lg^3 (n) >= 0$

lg(n) >= 0

vero quando n >= n0=1

qed.

Avendo dimostrato O e Omega vale Theta.

***Es 2**. Si imposti la relazione di ricorrenza che definisce il tempo di esecuzione della seguente funzione e la si risolva usando il metodo della sostituzione e si commentino opportunamente i passaggi del calcolo; si imposti l'induzione con chiarezza, sia nello scrivere quanto si vuole dimostrare sia nel formulare l'ipotesi induttiva.*
Per ottenere l'ipotesi da verificare si utilizzi l'albero della ricorsione.

```
fun test(A, i, j) {
   n = j-i+1                          c1
   m = n/4                            c2
   if n < 1 then return 0             c3
   if n = 1 then return (A[1]);       c4
   x = test(A, i, i+m) + test(A, i+m+1, i+2m)      c5+ 2T(n/4)
   for h = i..j do {                  n-volte
      x = x + A[h];                      c6
   }
   return (x);                        c7
}
```

L'eq di ricorrenza che descrive il costo della funzione test è:
T(n) = 2T(n/4) + Theta(n)
T(1) = O(1)

Risolvo la verisone semplificata senza notazione asintotica:
T(n) = 2T(n/4) + cn
T(1) = c'

Cerco un'ipotesi analizzando l'albero della ricorsione:

							#nodi	costo nodo	costo livello
			(n)				1	cn	cn
		/		\					
	(n/4)				(n/4)		2	cn/4	cn/2
/		\		/		\			
(n/16)		(n/16)		(n/16)		(n/16)	4	cn/16	cn/4
:									
(n/4^i)							2^i	cn/4^i	cn/2^i
:									
(1)							2^h	c'	2^h c'

h = $\log_4$ n 2^h = n^(1/2) = √n

Sommando il contributo di tutti i livelli ottengo:
T(n) = $\sum_{i=0..h-1}$ [cn/2^i] + 2^h c' = cn 2 + √n = Theta(n)

Dimostro per sostituzione prima T(n) = O(n) e poi T(n) = Omega(n).

Th. Esite d>0 t.c. T(n) <= dn = O(n)
dim.
passo base n=1
T(1) = c' <= d*1
devo prendere d >= c'

passo generico:
hp ind. la th vale per ogni m<n
T(n) = 2T(n/4) + cn = 2 [d n/4] + cn =
dn/2 + cn <= dn
d/2+c <= d
c <= d/2
devo prendere d >= 2c
Quindi prendendo d = max{c', 2c} la th è vera per ogni n >= n0=1
qed.

Th. Esite d>0 t.c. T(n) >= dn = Omega(n)
passo base n=1
T(1) = c' >= d*1
devo prendere d <= c'
passo generico:
hp ind. la th vale per ogni m<n
T(n) = 2T(n/4) + cn = 2 [d n/4] + cn =
dn/2 + cn >= dn
d/2+c >= d
c >= d/2
devo prendere d <= 2c

Quindi prendendo d = min{c', 2c} la th è vera per ogni n >= n0=1
qed.

***Es 3**. Data un array A di interi che rappresenta la visita in preorder di un Albero Binario di Ricerca T in **cui le chiavi sono tutte distinte**, si definisca una algoritmo iterativo che dia in output l'ABR T che ha A come visita in preorder.*
Si descriva a parole l'idea algoritmica, si produca lo pseudocodice e si analizzi il tempo di esecuzione asintotico.

La visita in order ha la struttura
A = [r]::S::D
quindi creo un albero vuoto, inserisco r, poi tutti i nodi nella parte sinistra S e poi tutti quelli nella parte destra D. In pratica inserisco tutti i valori di A dal primo all'ultimo.

```
fun rebuild(A, n) {
  if (n == 0) return null;
```

```
  T = new Node(A[0]);
  for i = 1..n-1
     abr_insert(T, A[i]);
  return T
}
```

Questo algoritmo ricostruisce correttamente T la cui visita in preorder è A in quanto:

- per ogni nodo v in T, v è inserito prima dei suoi figli perchè compare in A prima di loro.
- se v ha figlio sinistro x questo è il successivo in A, quindi abr_insert lo iserirà come figlio sx di v.
- se v ha figlio destro y, tutti i valori tra v e y in A sono < v e quindi saranno stati inseriti nel sottoalbero sx di v, che avrà ancora figlio destro null. Quindi quando vado ad inserire y verrà messo come figlio destro di v.

Il costo è pari a n chiamate ad abr_insert:
T(n) = n*O(h) = O(n^2).

Bonus:
E' possibile ottimizzare il costo tenendo traccia dell'ultimo nodo inserito v e cercando di inserire i nuovi nodi in prossimità di v, senza chiamare abr_insert che riparte ogni volta dalla radice:

```
fun rebuild(A, n) {
  if (n == 0) return null;
  T = new Node(A[0]);
  v = T;
  i = 1;
  while (i < n) {
     k = A[i];
     if (k < v.key) {  // k è figlio sx di v
        v.sx = new Node(k);
        i++;
        v = v.sx;
     } else if (k < v.parent.key) {  // k è figlio destro di v
        v.dx = new Node(k);
        i++;
        v = v.dx;
     } else {  // k non può essere figlio di v
        v = v.parent;
     }
  }
  return T;
}
```

Costo
T(n) = Theta(w)
dove w = numero esecuzioni del while

Analizziamo w in base ai 3 casi che possono verificarsi nel corpo del while:
$w = w_s + w_d + w_p$

$w_s + w_d = n-1$ // questo è il numero di volte in cui creo un figlio sx o un figlio dx

$w_p < n-1$ // questo è il numero di volte in cui risalgo verso il padre

$w <= 2n-2 = Theta(n)$

In definitiva l'algoritmo è lineare

$T(n) = Theta(n)$