Introduzione agli algoritmi

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

7 Luglio 2020

Es1. Si imposti la relazione di ricorrenza che definisce il tempo di esecuzione della seguente funzione e la si risolva usando il metodo della sostituzione. Si commentino opportunamente i passaggi del calcolo, si disegni l'albero della ricorsione e come si giunge alla previsione sull'andamento del tempo di calcolo, si imposti l'induzione con chiarezza, sia nello scrivere quanto si vuole dimostrare sia nel formulare l'ipotesi induttiva.

```
fun test(array A, int i, int f) {
    n = f–i+1;
    t = n³;
    m = n/2;
    if (n ≤ 1) then { return 1; }
    while (t ≥ 1) do { t = t-2; }
    return test(A, i, i+m) + test(A, i+m+1, f);
}
```

Sol. Le istruzioni al di fuori della chiamata sono eseguite in tempo costante, tranne il ciclo while che viene eseguito t/2 volte, cioè $n^3/2$ volte.

Ci sono inoltre due chiamate della funzione su circa la metà degli elementi quindi la relazione di ricorrenza da risolvere è

$T(n) = 2T(n/2) + \Theta(n^3)$.

Esplicitando le costanti:

$T(n) = 2T(n/2) + cn^3$ se $n > 1$

$T(n) = d$ altrimenti

L'albero della ricorsione, che ci consente di semplificare i calcoli per giungere a una previsione dell'andamento di T è binario, visto che ci sono due chiamate, e sotto l'ipotesi semplificatrice che $n = 2^h$, ha altezza h e ogni nodo sul livello i, $0 \le i < h$, è etichettato con $c(n/2^i)^3$. Sommando sul livello i si ottiene dunque $2^i\, c(n/2^i)^3 = cn^3/2^{2i}$. Al livello delle foglie invece ogni nodo corrisponde al costo per n= 1 quindi a d.

In conclusione $T(n) = 2^h d + \sum_{0\ldots h-1} cn^3/2^{2i} = 2^h d + cn^3 \sum_{0\ldots h-1} 1/2^{2i} \le 2^h d + cn^3 \sum_{0\ldots\infty} 1/2^{2i} = O(n^3)$, perchè la serie converge a una costante.

Ora verifichiamo la previsione per induzione.
Faremo vedere che esistono due costanti positive k e n' tali che $T(n) \leq kn^3$, per ogni $n \geq n'$.
Sia vero per ipotesi induttiva per ogni $m < n$, consideriamo T(n):
$T(n) = 2T(n/2) + c\ n^3 \leq$
$2k\ (n/2)^3 + c\ n^3 =$
$k\ n^3/4 + c\ n^3$.
Vediamo se troviamo due valori per k e n' tali che $k\ n^3/4 + c\ n^3 \leq k\ n^3$
Sviluppando:
$k\ n^3/4 + c\ n^3 \leq k\ n^3 \Leftrightarrow k\ n^3 + 4c\ n^3 \leq 4k\ n^3 \Leftrightarrow 4c\ n^3 \leq 3k\ n^3$ quest'ultima disuguaglianza è vera per $k = 2c$ e per ogni $n \geq 1$

Considerando il caso base abbiamo che $T(1) = d$ e $T(2) = 2d+8c$ e $T(2) \leq 8k$ è vero prendendo $k = 2d+8c$.
Questo valore va bene in tutti i casi.

Es 2. Siano dati un array *A[]* di interi di dimensione *n* e due pivot cioè due numeri interi *p1* e *p2*, con *p1* < *p2*. Si scriva una funzione che partizioni *A[]* in tre zone contigue: nella prima zona si trovano gli elementi minori o uguali a *p1*, nella seconda quelli maggiori di *p1* e minori o uguali a *p2* e nella terza quelli maggiori di *p2*.

Si calcoli il costo computazionale dell'algoritmo proposto.

Sol. Utilizzando le idee della partizione utilizzata nel quicksort possiamo costruire due algoritmi per risolvere il problema il primo dei quali è una semplice applicazione della partizione semplice e il secondo dei quai è ispirato all'algoritmo tripartition.

Nel primo caso eseguendo la partizione utilizzando p1 come pivot, spostiamo gli elementi minori o uguali a p1 tutti a sinistra e a seguire quelli maggiori di p1. In output la funzione dà l'indice j che individua l'ultima posizione degli elementi minori di p1, mentre quelli maggiori o uguali a p1 sono nelle entrate da j+1 a n-1. A questo punto la stessa funzione si può utilizzare per la porzione di A da j+1 a n-1, usando p2 come perno. Questa seconda chiamata sposterà tutti gli elementi minori o uguali a p2 sulla sinistra e a seguire i maggiori di p2 e fornirà in output l' indice k, che individua l'ultimo elemento minore o uguale a p2. In definitiva quindi si avrà che gli elementi di A di indice da 0 a j sono minori di p1, quelli di indice j+1 fino a k sono maggiori di p1 ma minori o uguali a p2 e da k a n-1 sono tutti maggiori di p2.

Riportiamo il codice della funzione partizione, non richiesto all'esame, ma per comodità del lettore.

Partizione(A,r,s,p1)

input: A è un array di interi, r,s,p1 sono interi

prec: 0≤r≤s≤A.length

Output: l'indice del primo elemento in A maggiore di p1, inoltre

nell'array A gli elementi ≤ p1 sono tutti spostati nelle posizioni da r a i.

```
 i = r-1  All'inizio non ci sono elementi ≤ p1
 j = r     e nemmeno > p1
 while j ≤ n-1 do
in ogni passo gi elementi tra r e i sono  ≤ p1  e gli elementi di indice tra i+1 e
j-1 sono > p1
              if  A[j] ≤ p1 then
                i =  i+1
                scambia A[i] con A[j]
         j= j+1
 return i+1
```

```
Algoritmo 1(A,p1,p2)
n = A.length
j = Partizione(A,0,n-1,p1)
k = Partizione(A,j+1,n,p2)
return (j,k)
```

La seconda soluzione qui sotto è più efficiente perchè otteniamo il risultato voluto con un'unica scansione dell'array, applicando la stessa idea della tripartition, vista a lezione.

```
        TriPartizione2pivot(A,p1,p2)
input: un array A e due interi
output: due indici i e j, gli elementi di indice da 0 a i sono minori di p1, quelli
di indice tra i+1 e j sono maggiori di p1 ma minori o uguali a p2 e quelli di
indice tra j e n-1 sono maggiori di p2.

n = A.length
   i = j = 0  All'inizio non ci sono
                elementi ≤ p1
    k = n-1     e nemmeno > p2
    while j ≤ k   do
                if  A[j] > p2 then
                      scambia A[j] con A[k]
                      k = k-1 endif
                if  A[j] < p1 then
                      scambia A[j] con A[i]
                       i = i+1
                       j = j+1 endif
                if  A[j] ≥ p1 then j = j+1
       endwhile
       scambia il pivot con A[j]
       return i,j
```

Es 3. Dato l'albero AVL mostrato qui sotto, si inserisca una nuova chiava scelta in modo da rendere necessaria una doppia rotazione per ripristinare il bilanciamento.

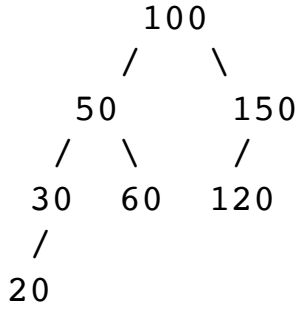
a) Si indichi il valore della chiave da inserire.

b) Si scriva come si aggiornano i fattori di bilanciamento.

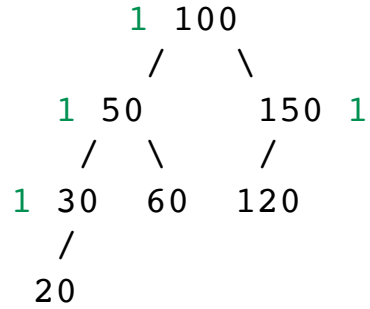
c) Si dica quali rotazioni devono essere eseguite, in che ordine e intorno a quali nodi.

d) Si aggiornino di conseguenza i fattori di bilanciamento.

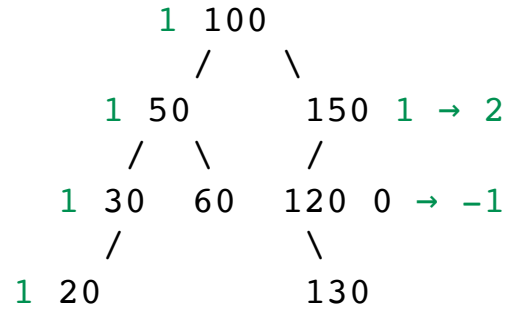
e) Sono necessari ulteriori controlli lungo il cammino verso la radice (se sì indicare quali se no indicare il motivo)?



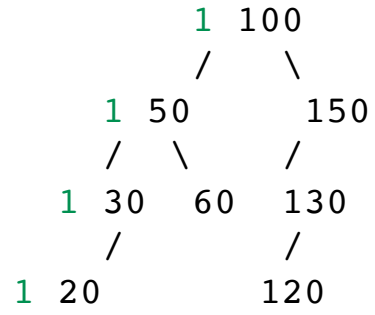
Sol. Per individuare i nodi nei quali l'inserimento può comportare uno sbilanciamento, conviene aggiungere i fattori di bilanciamento:



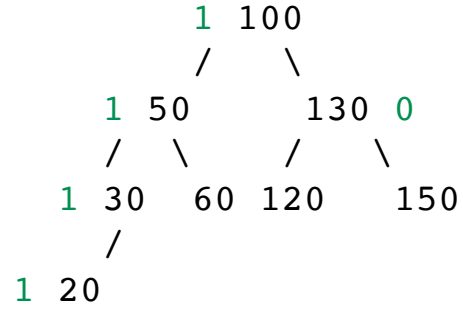
Per avere una doppia rotazione bisogna avere uno sbilanciamento con cambiamento di segno, quindi se aggiungiamo per esempio 130 il fattore di bilanciamento di 120 diventa -1 e il fattore di 150 diventa invece 2:



E’ un caso right-left, serve una rotazione a sinistra su 120, e una a destra su 150.
Dopo la rotazione a sinistra su 120:



Dopo la rotazione a destra su 150:



Non sono necessari ulteriori controlli risalendo verso la radice perchè il sottoalbero radicato in 130, che rimpiazza il sottoalbero radicato in 150 originario ha la stessa altezza, nello specifico ha altezza 1.