Esame scritto di
INTRODUZIONE AGLI ALGORITMI
Prof.ssa T. Calamoneri/Prof.ssa E. Fachini/Prof.ssa R. Petreschi
21 Settembre 2018

ESERCIZIO 1
Si consideri il seguente algoritmo di ordinamento:

Input: array H di n interi
Output: array H ordinato in ordine crescente

Passo 1: trasforma H in un Max-heap;
Siano 0, …, k i livelli di H
Siano $\mathbf{a_i}$ e $\mathbf{b_i}$ gli indici del primo ed ultimo elemento a livello i in H
Passo 2: for i=1 to k do
mergeSort(A,$\mathbf{a_i}$,$\mathbf{b_i}$)
(ordina in modo crescente il livello i, cioè gli elementi A[$\mathbf{a_i}$]...A[$\mathbf{b_i}$])
merge(A,1,$\mathbf{b_{i-1}}$,$\mathbf{b_i}$)
(fonde A[1]...A[$\mathbf{b_{i-1}}$] con A[$\mathbf{a_i}$]...A[$\mathbf{b_i}$])

Rispondere alle seguenti domande, motivando le risposte:

1) Quanto tempo richiede il Passo 1?
Risposta: Richede O(n) passi, utilizzando la funzione che costruisce un Max-Heap da un array qualunque visitando l'albero a partire dal primo nodo non foglia nel penultimo livello fino alla radice, proseguendo da destra verso sinistra su ogni livello Durante la visita si utilizza la funzione che ripristina la proprietà di essere Max-Heap violata alla radice del sotto albero in considerazione in ogni passo.
2) Quanto vale k in funzione di n?
Risposta: k = lg n (parte intera inferiore) visto che k è l'altezza di H. Un albero quasi completo ha altezza logaritmica nel numero dei nodi, visto che il più piccolo albero di questo tipo di altezza k ha esattamente 2^k nodi, mentre il completo di altezza k ne ha 2^{k+1} -1.
3) Quanti sono i nodi nel livello i del Max-heap, per $0 \leq i < k$? Quindi, quali sono i valori di $\mathbf{a_i}$ e $\mathbf{b_i}$ nel Passo 2?
Risposta: i nodi del livello i sono 2^i, con $0 \leq i < k$, visto che un Max-Heap è un albero completo fino al penultimo ultimo livello.

4) Qual è il numero totale di nodi nei livelli da 0 a i-1?
Risposta: E' la somma dei nodi nei primi i-1 livelli quindi 2^i -1.
5) **(Facoltativo: sarà corretto solo se si è risposto a tutte le 4 le domande precedenti)** Si stimi nel modo più accurato possibile il tempo di esecuzione del

Passo 2; a tal scopo può essere utile ricordare che $\sum_{i=0}^{k} 2^i$ è in $\Theta(2^k)$ mentre $\sum_{i=0}^{k} i\,2^i$ è in $\Theta(k2^k)$.

Risposta: Nel passo 2 si esegue k volte la chiamata di mergeSort(A,$\mathbf{a_i}$,$\mathbf{b_i}$), poiché il mergeSort ha un tempo di esecuzione in Θ(m lg m) se lavora su m elementi, nel nostro caso ha un tempo di esecuzione in $\Theta(2^i \lg 2^i)$ allora ogni chiamata ha un tempo di esecuzione $\Theta(2^i i)$, e la somma di tutti questo tempi per i =0 fino a k è esattamente $\Theta(k2^k) = \Theta(n \lg n)$. La funzione merge invece, in ogni passo, fonde gli elementi dei livelli da 0 a i-1 già ordinati nei passi precedenti, con il livello i appena ordinato.
Questo ha un tempo di esecuzione pari alla somma degli elementi da fondere, che nel nostro caso è 2^{i+1} -1. La somma quindi $\sum_{i=0}^{k} 2^{i+1} - \sum_{i=0}^{k} 1 = 2\sum_{i=0}^{k} 2^i - k = 2(2^{k+1} - 1) - k = \Theta(2^k) = \Theta(n)$. In definitiva i due passi hanno quindi un tempo di esecuzione $\Theta(n \lg n) + \Theta(n) = \Theta(n \lg n)$.

ESERCIZIO 2
Si consideri la seguente funzione:

```
fun (intero n) {
i=0;
for a=1 to n do
        for b=1 to a do
                i++;
if n<2 return i
else return fun(n/4);
}
```

Si scriva l'equazione di ricorrenza che esprime il tempo di esecuzione di `fun` in funzione di `n` e la si risolva senza fare uso del teorema principale o dell'esperto.

Sol. C'è una sola chiamata su un quarto degli elementi e il tempo di esecuzione delle istruzioni al di fuori delle chiamate è $\Theta(n^2/2)$ perchè il ciclo interno viene eseguito, tenendo conto di quello esterno, tante volte quant'è la somma dei primi n interi. Quindi la relazione è

$T(n) = T(n/4) + \Theta(n^2/2)$

Per risolverla esplicitiamo le costanti e semplifichiamo il termine additivbo:
$T(n) = d$ se $n \leq 1$
$T(n) = T(n/4) + cn^2$

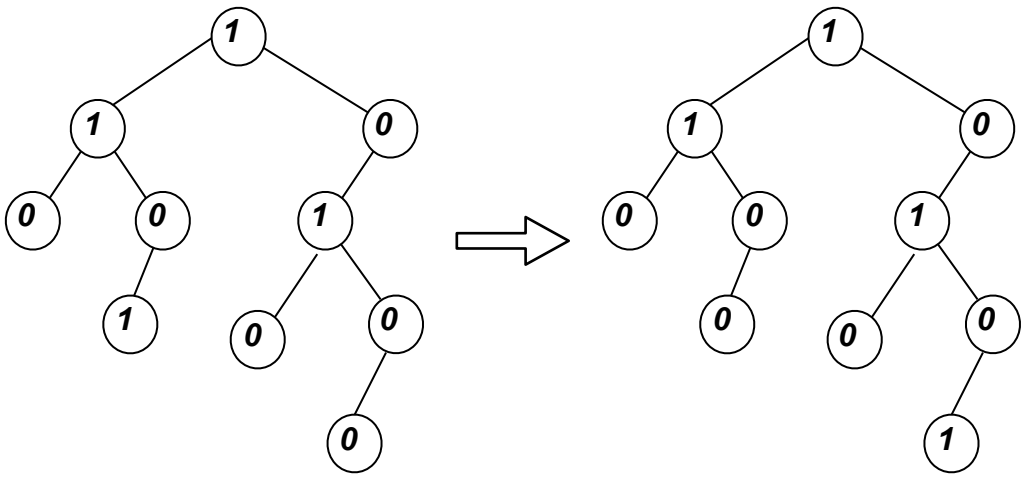
L'albero della ricorsione è un albero degenere di h livelli, se $n = 4^h$, e in cui il costo di un nodo al livello i è $c(n/4^i)^2$. In conclusione
$T(n) = T(1) + \Sigma_{i=0\ldots h} cn^2/4^{2i} =$
$T(1) + \Sigma_{i=0\ldots h} cn^2(1/4^2)^i \leq T(1) + \Sigma_{i=0\ldots\infty} cn^2(1/4^2)^i = O(n^2)$, perché la

somma di potenze con base minore di 1 si può maggiorare con la sua estensione all'infinito che converge a una costante.

Prova induttiva:
Si vuole dimostrare che esistono k e $n_0 \geq 0$ tali che $T(n) \leq kn^2$, per ogni $n \geq n_0$.
Supponiamo vera la tesi per ogni $m < n$ e consideriamo T(n). Poichè $n/4 < n$ possiamo applicare l'ipotesi induttiva a T(n/4) e quindi:
$T(n) \leq k(n/4)^2 + cn^2$ imponiamo che questo valore sia $\leq kn^2$ così otteniamo
$k(n/4)^2 + cn^2 \leq kn^2 \Leftrightarrow n^2k/4^2 + cn^2 \leq kn^2 \Leftrightarrow kn^2 + 16cn^2 \leq 16kn^2$
$\Leftrightarrow 16cn^2 \leq 15kn^2$ questo è vero per $k = 2c$ e per ogni $n \geq 1$.
Trascuriamo il caso base.
Abbiamo così dimostrato che $T(n) = O(n^2)$. Poiché il termine additivo della relazione di ricorrenza è quadratico possiamo concludere che $T(n) = \Theta(n^2)$.

ESERCIZIO 3
Dato un albero binario memorizzato tramite strutture a puntatori e contenenti solo chiavi 0 o 1 (senza dati satellite), scrivere un algoritmo che modifica le chiavi delle sole foglie in modo che, leggendole da sinistra a destra, tutti gli zeri precedano gli 1.

Si analizzino correttezza e tempo di esecuzione dell'algoritmo proposto.
Esempio:



Sol. Il problema si risolve con due visite inorder. La prima per contare il numero degli 0 nelle foglie e la seconda per scrivere tanti zeri quant'è il valore

restituito dalla prima visita come etichette delle prime foglie e poi tutti 1.
La visita per cambiare le chiavi nelle foglie in modo da rispettare la richiesta di aver una frontiera ordinata è organizzata in modo che la visita del sotto albero sinistro restituisca il numero di foglie la cui chiave viene posta a 0, in modo che nella chiamata al sotto albero destro si tenga conto delle foglie già modificate a sinistra.

```
Front(T)
input: un albero binario  T
prec: le chiavi sono 0 o 1
output: T con tutte le chiavi a 0 nelle foglie che precedono quelle con chiave 1
k = Conta0(T)
CambiaF(T,k)
```

```
Conta0 (T)
input: un albero binario
prec: le chiavi sono 0 o 1
output: il numero di foglie con chiave 0
if T== NULL return 0
n = Conta0(T.left)
m = Conta0(T.right)
if T.left == NULL and T.right == NULL then
      if T.key = 0 then return n+m+1 else return n+m
```

```
CambiaF(T,k)
input: un albero binario T e un numero intero k
prec: le chiavi sono 0 o 1 e k è il numero di foglie a chiave 0
output: T con tutte le chiavi a 0 nelle foglie che precedono quelle con chiave 1
e il numero delle foglie che da 0 diventano 1
if T== NULL return (T,0)
(T,h1) = CambiaF(T.left,k)
(T,h2) = CambiaF(T.right,k - h1)
if (T.left == NULL and T.right == NULL  and  k > 0 ) then
         T.key=0 return  (T,(h1+h2+1)) else
if (T.left == NULL and T.right == NULL  and k ≤ 0)
         T.key=1
return (T,h1+h2)
```