Introduzione agli Algoritmi
Esame Scritto a canali unificati
Testo con idee per la soluzione

docenti: T. Calamoneri, A. Monti
Sapienza Università di Roma
15 Settembre 2022

Esercizio 1 (10 punti):

Si supponga di avere un algoritmo speciale in grado di eseguire la fusione di due sottoarray ordinati di $n/2$ elementi ciascuno in $O(\sqrt{n})$ operazioni. Quanto sarebbe, in questo caso, il costo computazionale dell'algoritmo di Merge Sort?

a) Si imposti la relazione di ricorrenza che definisce il tempo di esecuzione giustificando dettagliatamente l'equazione ottenuta.

b) Si risolva la ricorrenza usando due metodi a scelta, dettagliando i passaggi del calcolo e giustificando ogni affermazione.

La relazione di ricorrenza è $T(n) = 2T\left(\frac{n}{2}\right) + O(\sqrt{n})$ con $T(1) = \Theta(1)$.
Applicando il teorema principale si vede che siamo nel caso 1. e quindi la soluzione è $\Theta(n^{\log_b a}) = \Theta(n)$.
Lo stesso risultato si ottiene ad esempio con il metodo iterativo, in cui l'equazione k-esima risulta:

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} 2^i O\left(\sqrt{\frac{n}{2^i}}\right) = 2^k T\left(\frac{n}{2^k}\right) + O\left(\sqrt{n}\sum_{i=0}^{k-1}\sqrt{2}^i\right).$$

Prendendo $k = \log_2 n$ si ha

$$T(n) = n \cdot \Theta(1) + \sqrt{n} \cdot O(\sqrt{n}) = \Theta(n).$$

Esercizio 2 (10 punti): Sia dato un array A contenente n interi distinti e ordinati in modo crescente. Progettare un algoritmo che, in tempo $O(\log n)$, individui la posizione più a sinistra nell'array per cui si ha $A[i] \neq i$, l'algoritmo restituisce -1 se una tale posizione non esiste.

Ad esempio, per $A = [0, 1, 2, 3, 4]$ l'algoritmo deve restituire -1, per $A = [0, 5, 6, 20, 30]$ la risposta deve essere 1 e per $A = [-3, 1, 2, 3, 6]$ la risposta deve essere 0.

Dell'algoritmo proposto:

a) si dia la descrizione a parole,

b) si scriva lo pseudocodice,

c) si giustifichi il costo computazionale.

a) **Per ottenere un tempo $O(\log n)$ nella dimensione dell'array, possiamo ricorrere ad una versione modificata della ricerca binaria: per prima cosa, verifichiamo se già il primo elemento dell'intervallo coincide con la sua posizione, in caso contrario restituiamo quella posizione. Si noti che, se non si esce qui, vuol dire che $A[0] = 0$ e che, quindi, gli elementi dell'array sono tutti non negativi, cosa che quindi vale da questo punto in poi. Altrimenti, confrontiamo l'elemento centrale dell'intervallo con il suo indice m; se si ha $A[i] = m$ allora l'elemento mancante, se c' è, deve essere necessariamente nell'intervallo di destra (perché le posizioni alla sinistra sono occupate dai numeri che vanno da 0 e m); se, al contrario, $A[m] \neq m$, si cerca nell'intervallo che va da i ad m.**
Verrà restituito -1 solo se si arriva all'intervallo "vuoto" (vale a dire l'intervallo (i, j) con $i > j$).

b) **Ecco di seguito una possibile implementazione in Python dell'algoritmo che utilizza una versione ricorsiva della ricerca binaria modificata La prima volta la sotto-funzione deve essere invocata sull'intero array, vale a dire $es2(A,\, 0,\, len(A) - 1)$.**

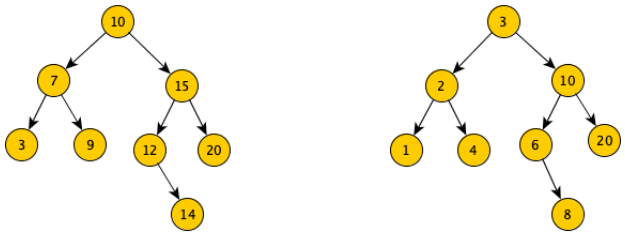
```
def es2(A):
    if A[0]!= 0:  return 0
    return es2R(A, 0, len(A) - 1).
```

```
def es2R(A, i,  j):
    if i > j:
        return −1
    if A[i]! = i:
        return i
    m = (i + j)//2
    if A[m] == m:
        return es2R(A, m + 1,  j)
    else:
        return es2R(A, i,  m)
```

c) **La ricorrenza dell'algoritmo è quella tipica della ricerca binaria, vale a dire $T(n) \leq T\left(\frac{n}{2}\right) + \Theta(1)$ per $n \geq 1$, $T(n) = \Theta(1)$ altrimenti. Questa ricorrenza risolta, ad esempio con il teorema principale ha come soluzione $O(\log n)$. Si noti che non basta dire questo, ma bisogna determinare e risolvere esplicitamente l'equazione.**

Esercizio 3 (10 punti): Progettare un algoritmo che, dato il puntatore alla radice di un albero binario T avente per chiavi degli interi, verifica se l'albero è un albero binario di ricerca.

Ad esempio, l'algoritmo per l'albero sulla sinistra deve restituire $True$ mentre per l'albero sulla destra deve restituire $False$ (infatti nel sottoalbero di sinistra del nodo con chiave 3 è presente un nodo con chiave 4)



Il costo computazionale dell'algoritmo proposto deve essere $\Theta(n)$ dove n è il numero di nodi dell'albero.
Dell'algoritmo proposto

a) si dia la descrizione a parole,

b) si scriva lo pseudocodice,

c) si giustifichi il costo computazionale.

a) **Bisogna verificare che la chiave di ciascun nodo x sia strettamente maggiore di TUTTE le chiavi presenti nei nodi del suo sottoalbero di sinistra e sia strettamente minore di TUTTE le chiavi presenti nei nodi del suo sottoalbero di destra. Un possibile algoritmo è il seguente:**

   - **Effettua una visita inorder dell'albero e metti le chiavi che via via incontri in un array A.**
   - **Scorri l'array A; se la sequenza di chiavi che incontri è strettamente crescente restituisci $True$ altrimenti restituisci $False$.**

   Per fare a meno dell'array ausiliario, si può effettuare una visita postorder e restituire al padre di ogni nodo v due valori che rappresentano l'intervallo dei valori contenuti nel sottoalbero radicato in v; l'intero albero risulta un albero binario di ricerca se la chiave memorizzata in ogni nodo è strettamente maggiore dell'intervallo restituito dal figlio sinistro e strettamente maggiore dell'intervallo restituito dal figlio destro.

b) **per semplicità riportiamo di seguito una possibile codifica in Python del primo algoritmo, dove abbiamo un programma principale che invoca una funzione ricorsiva per la visita inorder dell'albero:**

```
def es3(r):
    A = []
    visita_INORDER(r, A)
    for i in range(1,len(A)):
        if A[i]<=A[i-1]:  return False
    return True


def visita_INORDER(r, A):
    if not r:
        return
    visita_INORDER(r.left, A)
    A.append(r.key)
    visita_INORDER(r.right, A)
```

c) **Il costo computazionale è quello della visita di un albero con n nodi più il costo $O(n)$ dovuto alla successiva scansione dell'array A di n elementi. L'equazione di ricorrenza relativa alla visita è:**

- $T(n) = T(k) + T(n-1-k) + \Theta(1)$
- $T(0) = \Theta(1)$

che si può risolvere con il metodo di sostituzione dando come soluzione $\Theta(n)$.
Di conseguenza lil costo dell'algoritmo è, come richiesto, $\Theta(n)$.