Introduzione agli algoritmi
Prof.sse T. Calamoneri - E. Fachini - R. Petreschi
5 luglio 2019
Prova scritta completa

1. Dato un albero binario **qualunque** *T*, il suo *attraversamento pre-ordine* è una stringa che rappresenta l'elenco delle chiavi di *T* che scaturisce ordinatamente da una visita in pre-ordine.

    a. Mostrare, fornendo due diversi alberi binari con lo stesso attraversamento, che non si può ricostruire in modo univoco un albero binario conoscendo solo il suo attraversamento in pre-ordine.

    b. Descrivere poi un algoritmo ricorsivo che ricostruisca in modo univoco un albero binario del quale, oltre al suo attraversamento in pre-ordine, si conosce per ogni nodo se esso abbia figlio sinistro e/o figlio destro.

Sol.

a. Ricordiamo che la visita in pre ordine è definita ricorsivamente come la visita in cui la visita della radice precede (da qui "pre") quella del sotto albero sinistro e del sotto albero destro. Basta prendere due alberi con due nodi, stesse chiavi, ma il primo costituito dalla radice e un figlio sinistro e l'altro dalla radice e un figlio destro. Per esempio:

10 e 10

5 5

L'attraversamento in entrambi i casi è 10 5.

b. L'algoritmo riceve in input due array, A e B, A con le chiavi rappresentanti la visita in pre-ordine e B con i valori 2, se il nodo ha due figli, 1 se ha il figlio destro, -1 se ha il figlio sinistro e 0 se è una foglia e dà in output il puntatore alla radice dell'albero creato. L'indice i del valore dei due array da esaminare viene incrementato ogni volta che si genera un nuovo nodo. Inizialmente questo valore sarà 0.

L'algoritmo prende in considerazione A[i] e crea un nuovo nodo T di chiave A[i], poi controlla B[i] e se B[i] è 0 dà in output il puntatore al nodo creato; se B[i] = -1 dovrà fare una chiamata per agganciare il figlio sinistro al nodo T creato, infatti il figlio sinistro è visitato subito dopo la radice essendo a sua volta la radice del sotto albero sinistro; se B[i] = 1 vuol dire che il nodo ha solo il figlio destro e quindi si dovrà fare una chiamata per agganciare il figlio destro al nodo T creato, infatti il figlio destro è visitato subito dopo la radice visto che il sotto albero sinistro è vuoto e il figlio destro è la radice del sotto albero desto;

se B[i] = 2 si dovranno fare due chiamate per agganciare il figlio sinistro e il figlio destro al nodo creato. Il figlio sinistro è visitato subito dopo la radice, visitato il sotto albero sinistro si eseguirà la chiamata per agganciare il figlio destro; l'indice di scorrimento degli array è incrementato ad ogni passo in modo che quando si rientra nella chiamata del nodo T l'indice sia quello ottenuto dopo l'ultimo incremento relativo all'ultimo nodo visitato nel sotto albero sinistro (può essere pensato come una variabile globale).

2. Progettare un algoritmo che, dati in input due heap H1 e H2, dia in output un unico heap sugli elementi di H1 e di H2. Dell'algoritmo presentato:

    a. si dia la spiegazione a parole
    b. si valuti la complessità computazionale
    c. si fornisca lo psudocodice.

Per fondere due heap in un unico basta copiare i due array H1 e H2 in un nuovo array H che abbia come numero di elementi la somma dei numeri degli elementi dei due heap di partenza. All'array H si applica funzione BuildMax(Min)heap che prende in input un array qualunque e lo trasforma in un heap (sia che si tratti di un Max-heap che di un Min-heap). Il nuovo array ha come lunghezza la somma delle dimensioni dei due heap.
Il tempo di esecuzione asintotico è lineare nella dimensione del nuovo heap. Nello pseudocodice scegliamo di avere due Max-Heap.

Pseudocodice:

```
FondiHeap(H1,H2)
Input: due array di interi
Prec: i due array sono Max-heap
Output: un unico Max-heap sugli elementi di entrambi.
n = H1.heapsize + H2.heapsize
Crea un array H di n elementi (quindi H.length =n)
for i =1 to H1.heapsize do
    H[i] = H1[i]
    j =1
for i = H1.heapsize +1 to n do
    H[i] = H2[j]
    j++
BuildMaxHeap(H)
return H
```

```
Build-Max-Heap(A)
input: A è un array
output: A è un Max-heap
A.heapsize=A.length
for i = ⌊heapsize[A]/2⌋ downto 1 do
        Max-Heapify (A,i)
```

Segue la specifica di Max-Heapify:

```
Max-Heapify (A,i)
Input: un  l'array A e un intero i
Prec: A[left(i)] e A[right(i)] sono radici di Max-heap mentre  A[i] può
essere più piccolo dei suoi figli
Output: l'array A è tale  che anche A[i] è radice di un Max-Heap
```

3. Siano *A* e *B* due alberi binari di ricerca. Gli alberi sono memorizzati con strutture a puntatore e le radici dei due alberi hanno un campo aggiuntivo NumNodi con il numero dei nodi dell'albero. Progettare un algoritmo che, presi in input *A* e *B*, dia in output un array ordinato contenente l'unione delle chiavi di *A* e *B* (senza ripetizioni). L'algoritmo deve essere lineare nella somma del numero dei nodi dei due alberi. Dell'algoritmo presentato:

    a. si dia la spiegazione a parole
    b. si dimostri che ha l'andamento lineare atteso

Sol. Ci sono due soluzioni altrettanto semplici, entrambe basate sul fatto che la visita inorder di un ABR fornisce l'elenco delle sue chiavi in ordine crescente. La prima consiste nell'eseguire due visite inorder, una per ogni albero dato memorizzando la chiave di ogni nodo in un array, diciamo Ar per l'albero A e Br per l'albero B, poi eseguire una fondi dei due array in un terzo array. La fondi va un po' modificata in modo da non copiare due volte uno stesso elemento, ma per questo basta modificare i confronti facendo in modo che in caso di uguaglianza si memorizzi l'elemento nell'array di destinazione una sola volta, incrementando poi entrambi gli indici. Ricordiamo infatti che ogni ABR non ha ripetizioni. Questa soluzione ha il tempo di esecuzione asintotico richiesto, infatti detto n_A il numero dei nodi dell'albero A e n_B quello dell'albero B, abbiamo $\Theta(n_A) + \Theta(n_B)$ per le visite, poi $\Theta(n_A+n_B)$ per la fusione.

Pseudocodice:

```
ArrayChiaviOrd(A,B)
Input: A e B sono puntatori alle radici di alberi binari
prec: A e B sono due ABR, senza ripetizioni
Output un array con l'unione delle chiavi di A e B, in ordine crescente
   n_A = A.NumNodi
   n_B = B.NumNodi
   Crea un array Ar di n_A elementi
   Crea un array Br di n_B elementi
   InorderArray(A,Ar)
   InorderArray(B,Br)
   Crea un array C di n_A +n_B elementi
   FondiNoRip(Ar,Br,C)
   return C
```

Segue la specifica della funzione fondi:

```
FondiNoRip(L,R,A)
Input: L,R ed A  sono array di m, n e n+m elementi, rispettivamente
prec: L e R sono ordinati, cioè
L[0] ≤ ...≤ L[m-1] e R[0]≤…≤R[n-1]
output: A contiene l'unione degli elementi di L e di R e
A[0] ≤ ...≤ A[m+n-1]
```

Segue lo pdseudocodice della funzione di visita inorder con risultati in un array

```
InorderArray(T,A)
X = Min(T) // dà in output il puntatore al nodo di chiave minima
i=0
while  X ≠ nil then
        A[i] = X.key
        X = SUCC(X)
         i++
```

Seconda soluzione.

Questa soluzione non prevede array d'appoggio temporanei ma esegue la fusione direttamente nell'array definitivo C. Si parte confrontando i due minimi nei due ABR poi si copia in C il più piccolo tra i due. In ogni passo si confrontano le chiavi dei nodi correntemente in esame nei due alberi: il più piccolo viene copiato in C e si individua il nodo di chiave successiva a quella copiata in C per il prossimo confronto. Se i due elementi confrontati risultano uguali si copia da uno

dei due e si individuano i nodi con le chiavi successive nei due alberi. Si continua in questo modo fino a quando uno dei due alberi si svuota. Eventuali nodi rimanenti nell'altro albero hanno chiavi tutte più grandi dell'ultima inserita in C che quindi possono essere inserite di seguito, proseguendo la visita inorder per averli ordinati.
Il tempo di esecuzione asintotico è lineare nella somma dei numeri dei nodi nei due alberi, visto che si tratta di visitarli tutti, operando in tempo costante su ogni nodo.

```
ArrayChiaviOrd(A,B)
nA = A.NumNodi
nB = B.NumNodi
Crea un array C di nA + nB elementi
X = Min(A)
Y = Min(B)
i=0
while  X ≠ nil and Y ≠ nil then
            if X.key < Y.key then
                        C[i] = X.key
                        X = SUCC(A,X)
            else
                    if X.key > Y.key then
                            C[i] = Y.key
                            Y = SUCC(B,Y)
                    else
                            C[i] = X.key
                            X = SUCC(A,X)
                            Y = SUCC(B,Y)
            i++
while  X ≠ nil then
            C[i] = X.key
             X = SUCC(A,X)
            i++
while  Y ≠ nil then
            C[i] = Y.key
             Y = SUCC(A,Y)
            i++

return C
```

Parte I

1. Dimostrare la verità o falsità delle seguenti affermazioni:

   a. $f(n) = 12n^3 + 13n + 14$ è $O(n^3)$
   b. $f(n) = n^7 - n^3 + 4n^2$ è $\Omega(n^5)$
   c. $f(n) = n(n+1)/2$ è $\Theta(n)$

Sol.

a. *$12n^3 + 13n + 14$ è $O(n^3)$,* cioè sappiamo individuare due costanti positive n_0 e c tali che $12n^3 + 13n + 14 \leq cn^3$. Basta infatti prendere c = 12+13+14, allora si ha che $12n^3 + 13n + 14 \leq 12n^3 + 13n^3 + 14n^3$, disuguaglianza vera per ogni $n \geq 1$.
b. *$n^7 - n^3 + 4n^2$ è $\Omega(n^5)$* cioè sappiamo individuare due costanti positive n_0 e c tali che $n^7 - n^3 + 4n^2 \geq cn^5$. Per semplicità di calcolo dividiamo ogni termine per n^2, ottenendo $n^5 - n + 4 \geq cn^3$. Prendendo c = 1 abbiamo $n^5 - n + 4 \geq n^3$, che è vera per $n \geq 2$.
c. *$n(n+1)/2$ è $\Theta(n)$,* questa è falsa infatti una funzione quadratica non può essere limitata superiormente da una lineare. Infatti non troviamo una coppia c ed n_0 tali che $n(n+1)/2 \leq cn$ per ogni $n \geq n_0$. Se imponiamo $n(n+1)/2 \leq cn$, svolgendo i calcoli otteniamo $n^2 + n \leq 2cn$ e dividendo per n si ha $n + 1 \leq 2c$. E' evidente che per ogni scelta di c basta prendere n = 2c per avere una contraddizione.

2. Progettare un algoritmo che, dati in input due heap, dia in output un unico heap su tutti gli elementi. Dell'algoritmo presentato:

   i. si dia la spiegazione a parole
   ii. si valuti la complessità computazionale
   iii. si fornisca lo psudocodice.

Sol. Si veda la soluzione del medesimo esercizio come esame completo.

3. Descrivere a parole una procedura che preso in input un vettore di *n* coppie di interi *(k, i)*, in cui *k* può assumere solamente i valori 0, 1, e 2, lo ordina rispetto alla prima componente in modo stabile (cioè in modo tale che se *(k, i)* precede *(k, j)* nel vettore originale, allora *(k, i)* deve precedere *(k, j)* anche nel vettore ordinato). La procedura deve avere complessità *O(n).* Si motivi la linearità della soluzione.

Sol.

Per rispettare il vincolo sulla linearità della soluzione dobbiamo ricorrere al counting sort, visto che i dati da ordinare soddisfano la condizione di applicazione di questo algoritmo essendo nell'intervallo [0,2]. Non pos-

siamo utilizzare la versione semplice dell'algoritmo, poiché si tratta di coppie. La versione più generale consente anche di rispettare il vincolo di stabilità. Come è noto l'algoritmo utilizza un array ausiliario in cui si calcola il rango di ogni elemento in A. Il rango di un elemento A[i].1, la prima componente di A[i], cioè il numero degli elementi minori o uguali a A[i].1, fornisce la posizione che ogni elemento di A deve assumere nell'array ordinato. Se C[i] contiene il numero delle occorrenze di i tra le prime componenti delle coppie in A, per calcolare il rango di i basta sommare C[i-1] a C[i], partendo con i = 1. In C[i] si avrà la posizione più a destra della coppia con i come prima componente in A. Nel secondo passo quando si scorre da destra A, e per ogni A[i] si cerca in C la posizione che A[i] deve assumere in B nell'array C, decrementandone il rango in C per poter inserire nella corretta posizione eventuali duplicati.

Pseudocodice:

```
OrdinaCoppie(A)
Input: un vettore di n coppie di interi (k, i), in cui k può assumere solamente i valori 0, 1, e 2,
Output: il vettore A ordinato in modo stabile sulla prima componente
k = 2
Crea un vettore C di tre elementi, inizializzati a 0
for j=1 to n do
        i=A[j].1  la prima componente di A[j]
       C[i]=C[i] + 1
for i=0 to k do
       C[i]=C[i] + C[i-1]
for j = n downto 1 do
        m = A[j].1
        i = C[m]
        B[i].1 = A[j].1
        B[i].2 = A[j].2
        C[m] = C[m] - 1
```

Parte II

1. Dato un albero binario qualunque *T*, il suo *attraversamento pre-ordine* è una stringa che rappresenta l'elenco delle chiavi di *T* che scaturisce ordinatamente da una visita in pre-ordine.

   a. Mostrare, fornendo due diversi alberi binari con lo stesso attraversamento, che non si può ricostruire in modo univoco un albero binario conoscendo solo il suo attraversamento in pre-ordine.

   b. Descrivere poi un algoritmo che ricostruisca in modo univoco un albero binario del quale, oltre al suo attraversamento in pre-ordine, si conosce per ogni nodo se esso abbia figlio sinistro e/o figlio destro.

Sol. Si veda la soluzione del medesimo esercizio come esame completo.

2. Scrivere e risolvere con il metodo della sostituzione l'equazione di ricorrenza che esprime il tempo di esecuzione della funzione **Analizzami**, dove *A* è un array di *n* interi:

```
Analizzami(A, n)
if n ≤ 3 then return (A[1]);
j = 1;
while j ≤ n do
        A[j] = A[j] − A[n − j];
        j= j*3
for i = 1 to  3 do
        A[i] = A[i] + A[i + 1];
        Analizzami(A, n/3);
endfor
```

Sol. Ci sono tre chiamate ricorsivi su un terzo degli elementi e il ciclo esterno alle chiamate è eseguito $\Theta(\log_3 n)$, visto che l'indice j parte da 1 e ad ogni esecuzione del ciclo viene moltiplicato per 3, inoltre non appena $3^h > n$ si esce dal ciclo.

La relazione è quindi $T(n) = 3T(n/3) + \Theta(\log_3 n)$. Esplicitando le costanti ed evidenziando il caso base si ha:

$T(n) = d$ se $n \leq 3$
$T(n) = 3T(n/3) + c\log_3 n$

L'albero della ricorsione è un albero ternario completo se $n = 3^h$, e il costo di un nodo al livello i è $c \log_3(n/3^i)$, poichè i nodi sono 3^i il costo computazionale relativo al livello i è $3^i \log_3(n/3^i) = c3^i(\log_3 n - \log_3 3^i) = c3^i(\log_3 n - i)$ In conclusione

$T(n) = 3^hT(1) + \Sigma_{i=0\ldots h}\, c3^i(\log_3 n - i) \leq 3^hT(1) + \Sigma_{i=0\ldots h}\, c3^i \log_3 n =$
$nT(1) + c\log_3 n\, \Sigma_{i=0\ldots h}\, 3^i = nT(1) + c\log_3 n\, O(3^h) = O(n\log_3 n)$

Prova induttiva:
Si vuole dimostrare che esistono k e $n_0 \geq 0$ tali che $T(n) \leq kn\log_3 n$, per ogni $n \geq n_0$. Supponiamo vera la tesi per ogni $m < n$ e consideriamo T(n). Poichè $n/3 < n$ possiamo applicare l'ipotesi induttiva a T(n/3) e quindi:
$T(n) = 3T(n/3) + c\log_3 n \leq 3kn/3\log_3 n/3, + c\log_3 n = kn(\log_3 n - 1) + c\log_3 n = kn\log_3 n - kn + c\log_3 n$, imponiamo che questo valore sia $\leq kn\log_3 n$ così otteniamo
$kn\log_3 n - kn + c\log_3 n \leq kn\log_3 n \Leftrightarrow -kn + c\log_3 n \leq 0 \Leftrightarrow c\log_3 n \leq kn$ questo è vero per k = 1 e per ogni $n \geq 1$. Per il caso base notiamo che $T(3) = 3T(3/3) + c\log_3 3 = 3d + c$ e cerchiamo un valore di k per cui $3d + c \leq k3\log_3 3 = 3k$. Basta prendere k = d+c per soddisfare la disuguaglianza, possiamo concludere che $T(n) \leq (d+c)n\log_3 n$, per ogni $n \geq 3$.

3. Siano *A* e *B* due alberi binari di ricerca. Gli alberi sono memorizzati con strutture a puntatore e le radici dei due alberi hanno un campo aggiuntivo NumNodi con il numero dei nodi dell'albero. Progettare un algoritmo che, presi in input *A* e *B*, dia in output un array ordinato contenente l'unione delle chiavi di *A* e *B* (senza ripetizioni). L'algoritmo deve essere lineare nella somma del numero dei nodi dei due alberi. Dell'algoritmo presentato:
    a. si dia la spiegazione a parole
    b. si dimostri che ha l'andamento lineare atteso

Sol. Si veda la soluzione del medesimo esercizio come esame completo.