INTRODUZIONE AGLI ALGORITMI 25 Giugno 2018
Prof.ssa Fachini/Prof.ssa Petreschi

I PARTE

1. Definire il concetto di stabilità di un algoritmo di ordinamento e dire se, e sotto quali ipotesi, i seguenti algoritmi risultano stabili:

a) Insertion sort
b) Bubble sort
c) Heapsort

Vedere testo o lucidi.

2. La seguente funzione `Part()` dovrebbe essere usata nel Quick-Sort per effettuare la partizione ma è errata (si ricorda che `Part()` dovrebbe ritornare un intero k con $a \leq k \leq b$ tale che ogni valore in $A[a...k]$ è minore od uguale a tutti i valori in $A[k+1...b]$). Esibire un esempio su cui questa versione di `Part()` sbaglia e poi proporre una piccola modifica allo pseudocodice che lo renda corretto.

```
Part(A,a, b)
input: A è un array di interi, a e b due indici che individuano inizio e
fine della porzione dell'array su cui Part lavora
output: intero k con a ≤ k ≤ b tale che ogni valore in A[a...k] è minore
od uguale a tutti i valori in A[k+1...b].
k = a, j = b, v = A[a];
while (k < j) {
        if (A[k] > v) {
           SWAP(A[k], A[j])
           j--; }
        else k++; }
}
return k;
```

Se eseguiamo l'algoritmo su un piccolo array vediamo che il problema nasce dal fatto che quando j=k si esce dal while, ma l'elemento A[k] non è stato confrontato.

Prendiamo per esempio A= [2,5,8,1], all'inizio k=0 e j=3, poi k=1 e poiché 5 è maggiore di 2 si scambia 5 con 1 e si pone j = 2 ottenendo A= [2,1,8,5], poi 1 è minore di 2 e quindi k=2. A questo punto si esce dal while, ma 8 non è stato confrontato con 2 ed è falso che gli elementi di indice da 0 a 2 siano minori di 2.
In un altro caso, per esempio se A= [2,8,1,5] succede che k va a 1 dopo il primo confronto, poi j = 2 perché si scambia 8 e 5, ottenendo A= [2,5,1,8]. Poi 5 viene scambiato con 1 e j decrementato, quindi ora j=k=1 e A= [2,1,5,8], ora è 1 che non è stato confrontato con 2.

Notiamo che in questo caso la risposta giusta è k (=1), mentre nell'esempio precedente era k-1 (=1).
Ci si può convincere che la funzione lavora bene in un caso intermedio notando che se supponiamo che sia vero che ogni valore in $A[a...k-1]$ è minore od uguale a tutti i valori in $A[j+1...b]$). Quando si considera l'elemento di indice k questo viene aggiunto alla porzione dei minori o uguali se risulta tale, semplicemente incrementando k, mentre se è maggiore viene scambiato con quello di indice j, così ampliando la zona dei maggiori e infatti j viene correttamente decrementato. Si ottiene quindi di nuovo la situazione che ogni valore in $A[a...k-1]$ è minore od uguale a tutti i valori in $A[j+1...b]$). Poiché però si esce dal ciclo con j=k quell'elemento non viene confrontato. Non serve modificare la condizione di permanenza nel ciclo, mettendo "while k≤j" perché nel caso A[k] > v si farebbe lo scambio A[k] con A[j], ma essendo k=j questo non modifica l'array.
Se A[k] > v semplicemente gli elementi di indice da 0 a k-1 sono tutti minori o uguali a v e i successivi maggiori, mentre se A[k] ≤ v allora sono gli elementi di indice da 0 a k ad essere minori o uguali a v e i successivi maggiori. Quindi basta aggiungere fuori dal ciclo:
if (A[k] > v) return k-1 else return k

3. Dato uno heap massimo H contenente n chiavi intere, un intero positivo $k \leq n$ e un intero x, si scriva un algoritmo che determina se il k-simo elemento più grande in A è maggiore di x. Spiegare dettagliatamente l'algoritmo ideato e calcolarne il tempo computazionale.

Sol. Possiamo determinare il k-simo estraendo il massimo k-1 volte, infatti dopo queste estrazioni il max-heap avrà alla radice il k-simo più grande. Poi basterà confrontare questo valore con x e dare in output l'esito del confronto. Questo comporta un tempo di esecuzione O(k lg n). Nell'heapsort l'estrazione del massimo si ripete fino a ordinare tutto l'array, ma per determinare il k-simo non è necessario.

Pseudocodice:
K-simo(H,k,x)
input: un array di interi H, due interi k e x
prec: H è un max-Heap, n ≥ k
output: vero se il *k*-simo elemento più grande in *H* è maggiore di *x*, falso altrimenti

```
n = H.length
i = n
while k > 1 do
        scambia H[1] e H[i]
        H.heap-size   = H.heap-size  - 1
        Max-Heapify (H,1)
        i = i-1
        k = k-1
```

return (x > H[1])

INTRODUZIONE AGLI ALGORITMI 25 Giugno 2018
Prof.ssa Fachini/Prof.ssa Petreschi

II PARTE

1) Dato un albero AVL di altezza *h*, sia *m(h)* la massima differenza possibile (in valore assoluto) tra il numero di nodi dei sottoalberi destro e sinistro della radice. Ad esempio, *m(2)=2* ed *m(3)= 5*.
   a. Si dia una costruzione generale per l'AVL di massima differenza.
   b. Si dia il valore di *m(h)* nei casi $h = 5$ e $h = 7$, motivando la risposta.

Sol. Ricordiamo che l'altezza è la massima profondità di una foglia, la profondità di un nodo è il numero dei suoi antenati, escluso il nodo stesso. L'albero AVL di altezza h con il massimo numero di nodi è l'albero completo, che ne ha 2^{h+1}-1, mentre quello con il minor numero di nodi è l'albero di Fibonacci. Quindi se consideriamo un albero di altezza h il cui sotto albero sinistro è un albero completo di altezza h-1 e il sotto albero destro è un albero di Fibonacci di altezza h-2 avremo un albero AVL in cui la differenza (in valore assoluto) tra il numero di nodi dei sotto alberi destro e sinistro della radice è massima. Naturalmente invertendo i sotto alberi si ottiene un altro albero che soddisfa la richiesta.
Per l'AVL T così costruito si ha, detto $n_c(h)$ il numero dei nodi di un albero completo di altezza h e F_{h-2} il numero dei nodi di un albero di Fibonacci di altezza h-2:
$m(h) = n_c(h-1) - F_{h-2} = 2^h - 1 - F_{h-2}$
Per calcolare F_h dobbiamo ricordare che $F_0 = 1$, $F_1 = 2$ e $F_h = F_{h-1} + F_{h-2} + 1$.
Siamo quindi in grado di calcolare
$m(5) = 2^5 - 1 - F_3 = 31 - 7 = 24$ e $m(7) = 2^7 - 1 - F_5 = 127 - 20 = 107$

2) Si considerino le seguenti funzioni in pseudocodice:

```
f(int x)
i=x;
if (x == 1) return 1;
while i ≥ 1 do
      j=1;
      while (j < i) do j++;
      i = i-3;
return j


int g(int y)
if (y == 0) return 2;
return f(y)+3*g(⌊y/3⌋);
```

Si ricavi l'equazione di ricorrenza che esprime tempo di esecuzione asintotico di `g(y)`.

Sol. La funzione f contiene due cicli annidati, il più interno viene eseguito i volte e i valori di i sono n, poi (n - 3), poi (n - 6) …. L'esecuzione dei due cicli è quindi pari alla somma dei numeri da n a 1 a distanza 3, cioè n + (n - 3) + (n - 6) + … + (n-3*k), dove k è tale che n-3*(k+1) ≤ 1. Dunque k è circa n/3 e poiché quella somma cresce con n*k = n*n/3, il tempo di esecuzione asintotico di f è $O(n^2/3)$.
La funzione g chiama f e se stessa su un terzo degli elementi quindi la relazione di ricorrenza che esprime il tempo di esecuzione di g è
$T(n) = T(n/3) + O(n^2/3)$.
Per risolverla esplicitiamo le costanti e aggiungiamo il caso base:
$T(n) = d$ se $n \leq 1$
$T(n) = T(n/3) + cn^2$. (la costante c assorbe il fattore 1/3, così si semplificano i calcoli)

Ponendo $n = 3^h$ e sviluppando la relazione otteniamo la sommatoria $cn^2 + c(n/3)^2 + c(n/3^2)^2 + \dots + c(n/3^h)^2 = cn^2(1+1/3^2 + 1/(3^2)^2 + \dots + 1/(3^2)^h) = O(n^2)$ perché la somma di potenze con base minore di 1 può essere maggiorata con la sua estensione all'infinito che converge a una costante.
A questo punto si tratta di dimostrare che questa previsione di andamento è corretta, procedendo per induzione.
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/3 < n possiamo applicare l'ipotesi induttiva a T(n/3) e quindi abbiamo che $T(n/3) \leq k(n/3)^2$. Ricordando la definizione di T:
$T(n) = T(n/3) + cn^2 \leq k(n/3)^2 + cn^2$ imponiamo che questo valore sia $\leq kn^2$ così otteniamo
$3k(n/3)^2 + cn^2 \leq kn^2 \Leftrightarrow k/3n^2 + cn^2 \leq kn^2 \Leftrightarrow 2kn^2 - 3cn^2 \geq 0$ questo è vero per k = 2c e per ogni n≥1. Possiamo quindi concludere che $T(n) = O(n^2)$. Poiché il termine additivo della relazione di ricorrenza è anch'esso quadratico possiamo concludere che $T(n) = \Theta(n^2)$, visto che il termine additivo fornisce sempre un limite inferiore per T(n).

3) Dato un albero binario di ricerca *T*, progettare un algoritmo che trasforma *T* in un albero binario di ricerca *T'* in cui la radice di *T* è una foglia di *T'*. Spiegare dettagliatamente l'algoritmo ideato e calcolarne il tempo computazionale.

Sol. La soluzione più semplice consiste in tre passi:
1 salvare la chiave della radice
2 cancellare il nodo radice
3 inserire la chiave salvata.
Sappiamo infatti che ogni nuovo nodo viene inserito come foglia.

Nello pseudocodice si utilizzeranno le funzioni DELETE e INSERT qui specificate.
DELETE(T,X)
input: il puntatore e alla radice di un albero binario a chiavi intere e a un nodo
prec: T è un ABR e X è un suo nodo

output: l'albero T privato del nodo X

INSERT(T,k)
input: il puntatore e alla radice di un albero binario e un intero k
prec: T è un ABR
output: l'albero T con l'aggiunta di un nodo di chiave k

RadFo(T)
if T== NULL or T è una foglia then return
x = T.key
DELETE(T,T)
INSERT(T,x)

Volendo possiamo dettagliare meglio la funzione di cancellazione, riproducendo i casi previsti dalla cancellazione in generale, ma tenendo conto che operiamo sulla radice e che cancelliamo fisicamente il nodo di chiave successiva se la chiave di quest'ultimo rimpiazza quella della radice. Presentiamo un possibile pseudocodice che si riferisce a una rappresentazione in memoria dei nodi con quattro campi: figlio sinistro, figlio destro, padre e chiave. Si utilizza la funzione MINIMUM qui specificata:

MINIMUM(X)
input: il puntatore a un nodo di un albero binario
prec: X è un ABR e
output: il puntatore al nodo che contiene la chiave minima nel sotto albero destro di X

```
RadFo2(T)
if T == NULL or T è una foglia  then return
x = T.key
if  T.right == NULL then
                                T = T.left
                                T.left.p = NULL   // T.left non è nullo
if  T.left  ==  NULL then
                                T = T.right
                                 T.right.p = NULL // T.right non è nullo
else  // T ha due figli
       y = MINIMUN(T.right)
        T.key = y.key
         z = y.p
        if  z == T  then T.right =  y.right else  z.left = y.right
        if y.right ≠ NULL then  y.right.p = z
```

//se y è il figlio destro di T basta rendere figlio destro di T il figlio destro di y, altrimenti si rende il figlio destro di y figlio sinistro di suo padre, infine se il figlio destro di y non è nullo occorre modificarne anche il puntatore al padre, che in ogni caso è z.

INSERT(T,x)

Un'altra soluzione possibile usa le rotazioni per portare il nodo radice in basso fino a farlo diventare una foglia. Useremo le funzioni RotD(T) e RotS(T) qui specificate:

RotR(T)
input: il puntatore a un nodo di un albero binario a chiavi intere
prec: T è un ABR
output: il puntatore T del nodo dopo la rotazione a destra su T

RotS(T)
input: il puntatore a un nodo di un albero binario a chiavi intere
prec: T è un ABR
output: il puntatore T del nodo dopo la rotazione a sinistra su T

```
RadFo3(T)
if T == NULL or T è una foglia  then return
if (T.left ≠ NULL) then X = RotD(X)   else X = RotS(X)
// non possono essere entrambi nulli
T = X.p
//dopo la prima rotazione ho la nuova radice
while (X.left ≠ NULL or  X.right ≠ NULL) do
        if (T.left ≠ NULL) then X = RotD(X)
        else X = RotS(X) // non possono essere entrambi nulli
return T
```

INTRODUZIONE AGLI ALGORITMI 25 Giugno 2018

Prof.ssa Calamoneri / Prof.ssa Fachini /Prof.ssa Petreschi Intero esame

1. La seguente funzione `Part()` dovrebbe essere usata nel Quick-Sort per effettuare la partizione ma è errata (si ricorda che `Part()` dovrebbe ritornare un intero k con $a \leq k \leq b$ tale che ogni valore in $A[a...k]$ è minore od uguale a tutti i valori in $A[k+1...b]$). Esibire un esempio su cui questa versione di `Part()` sbaglia e poi proporre una piccola modifica allo pseudocodice che lo renda corretto.

```
Part(A,a, b)
input: A è un array di interi, a e b due indici che individuano inizio e
fine della porzione dell'array su cui Part lavora
output: intero k con a ≤ k ≤ b tale che ogni valore in A[a...k] è minore
od uguale a tutti i valori in A[k+1...b]).
k = a, j = b, v = A[a];
while (k < j) {
        if (A[k] > v) {
           SWAP(A[k], A[j])
           j--; }
        else k++; }
}
return k;
```

2. Si considerino le seguenti funzioni in pseudocodice:

```
f(int x)
i=x;
if (x == 1) return 1;
while i ≥ 1 do
     j=1;
     while (j < i) do j++;
     i = i-3;
return j

int g(int y)
if (y == 0) return 2;
return f(y)+3*g(⌊y/3⌋);
```

Si ricavi l'equazione di ricorrenza che esprime il costo computazionale di `g(y)`.

3. Dato un albero binario di ricerca T, progettare un algoritmo che trasforma T in un albero binario di ricerca T' in cui la radice di T è una foglia di T'. Spiegare dettagliatamente l'algoritmo ideato e calcolarne il tempo computazionale.

Si vedano le soluzioni degli esercizi scelti tra la prima e la seconda parte.