Introduzione agli algoritmi
Prova di esame del 19/9/2016
Prof.sse E. Fachini - R. Petreschi

Parte prima

1) Si dimostri il teorema sulla limitazione inferiore per il tempo asintotico di esecuzione nel caso peggiore di un algoritmo di ordinamento nel modello basato sui confronti.

2) a) Dimostrare la verità o la falsità delle seguenti asserzioni:

$$2^{n+1} = O(2^n)$$

Sol. E' facile vedere che esistono due costanti c ed $n_0 \geq 0$ tali che $2^{n+1} \leq c\, 2^n$ visto che $2^{n+1}=22^n \leq c\, 2^n$ per una qualsiasi scelta di $c \geq 2$ e per ogni $n \geq n_0=0$.

$$2^{2n} = O(2^n)$$

Sol. E' facile vedere che non possono esiste due costanti c ed $n_0 \geq 0$ tali che $2^{2n} \leq c\, 2^n$ per ogni $n \geq n_0$. Infatti $2^{2n} = 2^n 2^n$ e $2^n 2^n \leq c\, 2^n$ sse $2^n \leq c$, ma nessuna costante può maggiorare una funzione crescente.

$$f(n) = O(g(n)) \text{ implica } 2^{f(n)} = O(2^{g(n)})$$

Sol. A prima vista sembra molto ragionevole questa implicazione, ma non è vera. Infatti se si prende $f(n) = 2n$ e $g(n) = n$ si ha che $f(n) = O(g(n))$, ma $2^{f(n)} = 2^{2n}$ e $2^{g(n)} = 2^n$ e abbiamo appena dimostrato che $2^{2n} \neq O(2^n)$

b) Se si dimostra che un algoritmo ha tempo di esecuzione $\Theta(n^2)$ nel caso migliore, posso dedurne che nel caso peggiore terminerà in $O(n^2)$ passi?

Sol.La risposta è no perché un limite superiore per il caso migliore non necessariamente vale per il peggiore, che potrebbe per esempio avere tempo di esecuzione in $O(n^3)$.

3) Un albero d-ario è definito come un albero binario, ma il massimo numero di figli che può avere è $d \geq 2$ e non 2. Un albero d-ario quasi completo è un albero d-ario in cui tutti i livelli sono pieni, tranne l'ultimo nel quale le foglie sono a sinistra. Supponiamo che le chiavi dei nodi siano intere. Un albero d-ario quasi completo può essere rappresentato in memoria in un array A come segue:

La chiave della radice è in A[1], quelle dei figli sono in A[2] ,…, A[d+1] e così via come nel caso binario.

Il padre di un nodo si individua usando la procedura:

D-ARY-PARENT(i) return $\lfloor (i-2)/d + 1 \rfloor$.

Il j-simo figlio di un nodo i si individua usando la procedura:

D-ARY-CHILD(i, j) return $d(i-1) + j + 1$

L'altezza dell'albero d-ario quasi completo è $\Theta(\log_d n)$.

Supponiamo ora che le chiavi soddisfino la proprietà del max-heap, cioè che ogni nodo abbia chiave maggiore o uguale a quella dei suoi figli.Si implementi le si analizzi dal punto di vista de tempo di esecuzione asintotico l'operazione di estrazione, con eliminazione, del massimo d-ARY-HEAP-EXTRACT-MAX.

Sol. Si estende facilmente al caso d-ario l'idea utilizzata nel caso binario. Per prima cosa si copia il massimo A[1], in una variabile di appoggio per poterla esportare, poi si prende l'ultimo elemento dell'array A nel max-heap, cioè l'ultima foglia a destra sull'ultimo livello, e se ne copia il valore nella prima posizione. A questo punto si deve ripristinare la proprietà del max-heap violata nella radice, e solo nella radice. Per questo possiamo avvalerci di una estensione agli alberi d-ari della procedura usata per i binari. Si calcola il massimo tra i figli della radice e lo si scambia con questa se risulta minore, si ripete su ogni figlio su cui si fa lo scambio fino a che la proprietà è ristabilita o si è giunti in una foglia. La complessità è $O(d\log_d n)$, visto che calcolare il massimo sui d figli, con costo O(d), è un'operazione che può essere compiuta nel caso peggiore $\log_d n$ volte e tutte le altre operazioni sono di costo costante.

Pseudocodice:

```
D-Heap-Extract-Max (A,d)
Input: A è un array
Prec: A è un max-heap d-ario
Postc: estrae il massimo rispristinando la proprietà del max-heap
  if A.heap.size < 1 then error "l'heap è vuoto"
  max = A[1]
  A[1]  = A[A.heap.size]
```

```
A.heap.size = A.heap.size - 1
D-Max-Heapify (A,1)
return max

  D-Max-Heapify (A,i)
```
Input: A è un array e i è un indice

Prec: D-ARY-CHILD(i, j), per 1≤j≤d, sono radici di d-max-heap mentre A[i] può essere più piccolo di uno dei suoi figli

Postc: A[i] è radice di un d-max-Heap

```
  max = i
  while max == i do
     if (d(i − 1) + 2 ≤ A.heap.size) then max = A[d(i − 1) + 2]
```
il primo figlio, se c'è, è il massimo corrente
```
    for j = 2 to d-1 do
      if (d(i − 1) + j + 1 ≤ A.heap.size) then
          if (A[d(i − 1) + j + 1] < A[d(i − 1) + j + 2]) then max = d(i − 1) + j + 2
```
(tra i figli , se ci sono, si prende l'indice del massimo)
```
    if (A[max] > A[i]) then  scambia A[i]  e  A[max]; i = max
                       else max = i+1
```
(se max è l'indice di un figlio e A[max] ≤ A[i] si esce dal ciclo perché i≠max, se A[i] è una foglia, nell'ultimo if A[i]=A[max] e quindi ancora max≠i)

Introduzione agli algoritmi
Prova di esame del 19/9/2016
Prof.sse E. Fachini - R. Petreschi

Parte seconda

1. Disegnare l'albero binario, con chiavi alfabetiche, le cui visite inordine e preordine sono rispettivamente **GDHBAECJIKF** e **ABDGHCEFIJK**.

Sol. Dalla definizione di visita in preordine si sa che la radice è A, dalla definizione di quella in inordine GDHB sono nel sottoalbero sinistro di A e ECJIKF in quello destro. Riapplicando questo modo di individuare le radici e i sottoalberi ai sottoalberi sinistro e destro si giunge a costruire l'albero. Infatti B risulta la radice del sotto albero sinistro e C di quello destro. Inoltre B non ha il sotto albero destro perché nella inordine tutti gli altri nodi del sotto albero sinistro sono visitati prima di B. L'esercizio può essere completato facilmente a questo punto.

2. Si determini il tempo di esecuzione asintotico T(n) della seguente funzione:

```
analizzami(A,i,j)
n = j - i+1
c = 1
m = (i + j +1)/2
k = (i+m+1)/2
h = (m+j+1)/2
d = m
while d > i do
      for s = 1 to n do c++
      d = d - 2
if n > 1 then
         return analizzami(A,i,k) + analizzami(A,k+1,m) + analiz-
zami(A,m+1,h) + analizzami(A,h+1,j)
```

Sol. La relazione è $T(n) = 4T(n/4) + O(n^2)$, perché all'esterno delle chiamate oltre alle istruzioni di costo costante, ci sono due cicli annidati di cui il più interno eseguito n volte e il più esterno n/4 volte. Ci sono poi 4 chiamate, ciascuna su un quarto degli elementi dati in input.
L'albero della ricorsione:

cn^2

$c(n/4)^2$ $c(n/4)^2$ $c(n/4)^2$ $c(n/4)^2$

Al livello successivo troviamo 16 nodi ciascuno con un costo associato pari a $c(n/16)^2$. Possiamo quindi dedurre che il costo per ogni livello i è $4^i c(n/4^i)^2 = cn^2 (1/4)^i$. Sfruttando quindi il fatto che la serie di potenze con base minore di 1 converge a una costante, possiamo prevedere che $T(n) = O(n^2)$.
La prova induttiva che esistono n_0 e d tali che $T(n) \leq dn^2$, per ogni $n \geq n_0$
Sia vero che esistono n_0 e d tali che $T(m) \leq dm^2$, per ogni $m \geq n_0$ e $m<n$.
Consideriamo $T(n) = 4T(n/4) + cn^2 \leq 4d(n/4)^2 + cn^2 = dn^2/4 + cn^2$.
Verifichiamo se $dn^2/4 + cn^2 \leq dn^2$. Ma $dn^2 + 4cn^2 \leq 4dn^2$ sse $4cn^2 \leq 3dn^2$. Prendendo $n>1$ si ha che d deve essere maggiore di $4/3c$,qundi per esempio potremmo prendere $d=2c$ e $n_0=1$.
Controllando il caso base, si ha che $T(n) = a$ per una costante a e per ogni $n \leq 1$, Poiché $T(2) = 4T(0) + 4c = 4a+4c$, dobbiamo prendere $d=\max\{2c,2a\}$ e $n_0=1$.
Poiché il termine additivo nella relazione di ricorrenza è quadratico, possiamo concludere che $T(n) = \theta(n^2)$.

3. Sia T un albero binario radicato in cui ogni nodo ha un valore intero come chiave. Definiamo costo di un cammino radice-foglia la somma delle chiavi dei nodi che compongono il cammino. Il diametro di T è il massimo fra i costi dei cammini radice foglia in T. Si progetti un algoritmo per trovare il diametro di un albero T.

Sol. Il problema è una variante del calcolo dell'altezza. Si fa una visita in postorder dell'albero. Per ogni nodo i risultati delle chiamate ricorsive sui figli sono i diametri nei due sottoalberi, questi vanno confrontati e il maggiore tra i due, sommato alla chiave della radice, è il risultato della chiamata su quel nodo. Trattandosi di una visita, in cui per ogni nodo si eseguono solo confronti e assegnamenti, il tempo di esecuzione in ogni caso è $\theta(n)$, se n è il numero dei nodi dell'albero.

```
Diametro(T)
if T= NULL return 0
s = Diametro(T.left)
d = Diametro(T.right)
if (s≥d) return s+T.key else return d+T.key
```