Introduzione agli Algoritmi
luglio 2014
Prof. Emanuela Fachini (canale 1)
Parte I

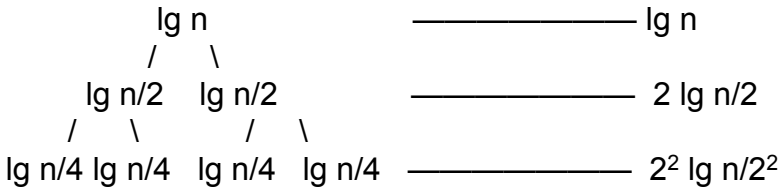
Esercizio 1
Si consideri la seguente funzione:

```
test (A,lo,hi)
 if hi ≤ lo then return 1
 m = (lo+hi)/2
 k = m
 a = 1
  while k ≥ 1 do
          a = 2*a
          k = k/2
  return test(A,lo,m) and test(A,m+1,hi)
```

Scrivere la relazione di ricorrenza che descrive il tempo di esecuzione T della funzione *test* e risolverla.

Soluzione:
Il ciclo while viene eseguito lg n/2 volte quindi la relazione è:
$T(n) = 2T(n/2) + \Theta(\lg n/2)$.
Per risolverla stabiliamo il caso base e eliminiamo la notazione asintotica semplificando:
$T(n) = a$ se $n \leq 1$
$T(n) = 2T(n/2) + b \lg n$ altrimenti

l'albero della ricorsione:



….

Al livello i si ha un costo totale pari a $b2^i \lg n/2^i$ quindi, posto $n = 2^k$ si ottiene:
$T(n) = 2^k T(1) + \Sigma_{i=0,\ldots,k-1} b2^i \lg n/2^i \leq 2^k T(1) + b\Sigma_{i=0,\ldots,k-1} 2^i \lg n = 2^k T(1) + b \lg n(2^k - 1) = O(n \lg n)$

Prova induttiva che $T(n) = O(n \lg n)$ e cioè che esistono due costanti c ed n_0 tali che $T(n) \leq c\, n \lg n$ per ogni $n \geq n_0$

Passo induttivo. Assumiamo che $T(m) \leq c\, m \lg m$ per $m < n$ e consideriamo $T(n)$.
$T(n) = 2T(n/2) + b \lg n \leq 2\, cn/2 \lg n/2 + b \lg n = cn(\lg n - \lg 2) + b \lg n = cn\lg n - cn + b \lg n$.
Vogliamo che $cn\lg n - cn + b \lg n \leq c\, n \lg n$ e questo è banalmente vero per ogni scelta di $c \geq b$ e $n_0 = 1$, visto che $n > \lg n$, per ogni $n > 0$.
Il caso base è lasciato per esercizio.

Introduzione agli Algoritmi
19 giugno 2014
Prof. Emanuela Fachini (canale 1) e Prof. Irene Finocchi (canale 2)

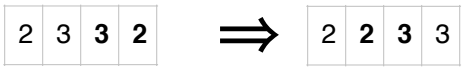
Esercizio 2

Un algoritmo di ordinamento è stabile se al termine della sua esecuzione eventuali elementi uguali sono ancora nell'ordine relativo iniziale.
Il selectionSort studiato a lezione non è stabile.

```
SelSort(A)
n = len(A)
for (j = 0; j ≤ n-2; j++)
            minInd = Min(A,j,n-1)
            scambia A[j] con A[minInd]
```

```
Min(A,lo,hi)
minInd = lo
for (k = lo+1;k ≤ hi; k++)
    if A[k] < A[minInd ]
    then minInd = k
return minInd
```

Per esempio:



A. Si modifichi l'algoritmo in modo da renderlo stabile. Si analizzi la complessità dell'algoritmo proposto nel caso migliore e peggiore.
B. Il mergesort studiato a lezione è stabile? Si motivi la risposta.

Soluzione.

Per il punto A si deve modificare l'algoritmo sostituendo allo scambio l'inserimento di A[minInd] nella posizione j. L'inserimento avviene dopo aver "creato lo spazio" spostando di una posizione a destra gli elementi A[j],…, A[minInd-1], avendo salvato A[minInd] in una variabile di appoggio. Se quindi due elementi uguali si trovano nelle posizioni j e k con j < k, quello nella posizione j sarà quello trovato per primo e quello nella posizione k sarà sistemato nella posizione successiva. Quindi questa modifica rende il SelectionSort stabile.

```
SelSort(A)
input: un arry A
postc: A è ordinato in ordine crescente
n = len(A)
for (j = 0; j ≤ n-2; j++)
            minInd = Min(A,j,n-1)
            insert(A,j,minInd)
```

```
insert(A,j,m)
input: un array A e due indici
postc: l'elemento A[m] è spostato nella posizione j, facendo slittare di una
posizione a destra gli elementi tra j e m-1
x = A[m]
```

Introduzione agli Algoritmi
luglio 2014
Prof. Emanuela Fachini (canale 1)

```
for (k = m;k > j;k- -)
      A[k] = A[k-1]
A[j] = x
```

La complessità asintotica non cambia perché la ricerca del minimo e la insert hanno costo lineare e quindi, sia nel caso peggiore che in quello migliore si ha $O(n^2)$.

Per il punto B basta osservare che la merge fonde due sottoarrays L e R ordinati il primo contenente gli elementi della metà sinistra della porzione di array in analisi e la seconda quelli della metà destra e che nel confronto L[j] ≤ R[k] l'elemento L[j] è vincente e viene copiato prima dell'eventualmente uguale elemento R[k]. Quindi gli elementi uguali mantengono le posizioni relative originali.

Parte II

Esercizio 3

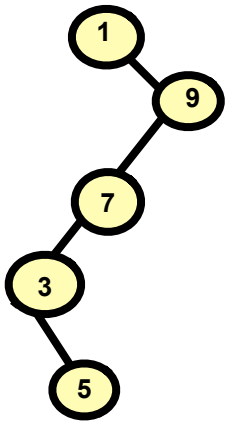
A. Qual'è il massimo numero di nodi che può avere un albero binario di altezza h?
B. Qual'è il minimo numero di nodi che può avere un albero binario di altezza h? Si motivi la risposta.
C. Se ne deducano i limiti inferiori e superiori per l'altezza, h, di un albero binario in termini del numero dei nodi, n.
D. Si dimostri che è possibile costruire un maxheap, a partire da un array di interi qualunque di n elementi, in O(n). Si descriva l'algoritmo e se ne analizzi la complessità.

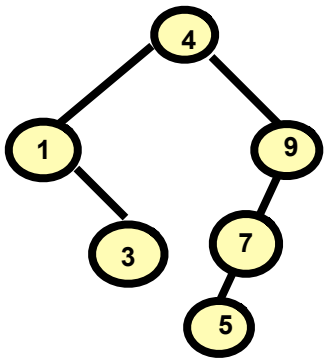
Vedere libro di testo e/o slides.

Esercizio 4

Si scriva un algoritmo di inserimento di un nuovo elemento in un ABR, facendo in modo che il nuovo elemento dopo l'inserimento sia alla radice dell'ABR dato. Esempio, nell'albero T si inserisce un nodo di chiave 4:

T=





Si giustifichi la correttezza dell'operazione e se ne fornisca la complessità asintotica.

Introduzione agli Algoritmi
19 giugno 2014
Prof. Emanuela Fachini (canale 1) e Prof. Irene Finocchi (canale 2)

Soluzione.
Basta far seguire al normale inserimento in un ABR, un ciclo all'interno del quale vengono eseguite delle rotazioni. Se il nodo inserito x è figlio sinistro si farà una rotazione a destra su suo padre, altrimenti una a sinistra, si sostituisce a x il padre di x e si ripete il processo fino a che il padre di x non è nullo. Così x va alla radice.
La complessità è ancora O(h) perché si fa un numero di rotazioni, ciascuna di costo costante, lungo il cammino dal nodo inserito fino alla radice, che nel caso peggiore è lungo proprio h.
Seguono le specifiche delle funzioni utilizzate e lo pseudocodice della funzione che porta un nodo x alla radice e della funzione richiesta.

InsertRoot(k, T)

input: T è il riferimento alla radice di un albero
prec: T è un ABR e k una chiave non presente in T
post: restituisce un ABR con i nodi di T e radice di chiave k
x = Insert(k, T)
SpostaaRadice(T,x)

Insert(k, T)

input: T è il riferimento alla radice di un albero
prec: T è un ABR e k una chiave non presente in T
post: inserisce la chiave k e restituisce il puntatore al nodo di chiave k in T

```
SpostaaRadice(T,x)
        Input: un puntatore alla radice, T e a un nodo, x
        prec: T è un ABR non nullo e x è un nodo di T
        postc: modifica T in modo che il nodo x sia la radice
        y = x.p
      while y ≠ nil do
          y è il padre di x
          if (y.left ≠ Null and y.left == x) then RightRot(y)
          if (y.right ≠ Null and y.right == x) then LeftRot(y)
          y = x.p
```

LeftRot(y)
input: y è un nodo di un albero binario

Prec: y è un nodo di un ABR non vuoto che ha il figlio destro

postc: il figlio destro, x, di y diventa suo padre e ne prende il posto come figlio;

conserva il suo figlio destro, e prende y come figlio sinistro mentre il figlio sinistro di x diventa figlio destro di y.

RightRot(y)

input: y è un nodo di un albero binario

Prec: y è un nodo di un ABR non vuoto, che ha il figlio sinistro

postc: il figlio sinistro, x, di y diventa suo padre e ne prende il posto come figlio; conserva il suo figlio sinistro,

e prende y come figlio destro mentre il figlio destro di x diventa figlio sinistro di y.