Soluzioni prova intermedia

Testo A

Esercizio 2a

a. Si confronti $10n^2$ con $2000\lg^2 n$ e si provi o si confuti che
$2000\lg^2 n = O(n^2)$ e $10n^2 = O(\lg^2 n)$.

$2000\lg^2 n = O(n^2)$ è vero perchè esistono due costanti $c, n_0 \geq 0$, tali che $2000\lg^2 n \leq c\, n^2$, per ogni $n \geq n_0$, basta prendere $c = 2000$ e $n_0 = 1$. Infatti si ha
$2000\lg^2 n \leq 2000\, n^2 \Leftrightarrow \lg^2 n \leq n^2 \Leftrightarrow \lg n \leq n$

$10n^2 = O(\lg^2 n)$ è falsa perché per ogni scelta di c si trova un valore di n per cui la condizione $10n^2 \leq c\, \lg^2 n$ è falsa.
Infatti per $n = 2^c$ si dovrebbe avere che $10\,(2^c)^2 \leq c\, \lg^2 2^c$
$\Leftrightarrow 10\, 2^{2c} \leq c^3$ che è falsa per ogni $c \geq 1$.

Esercizio 2b

2b.
Se si dimostra che un algoritmo ha tempo di esecuzione $\Theta(n^2)$ nel caso migliore, è possibile che in qualche caso l'algoritmo termini in $\Theta(n \lg n)$ passi? Si motivi la risposta.

No non è possibile perché il tempo di esecuzione nel caso migliore fornisce un limite inferiore ai tempi in tutti i casi.

Soluzioni prova intermedia

Testo B

Esercizio 2a

a.Si confronti 20n con 5000lg n e si provi o si confuti che 5000lg n = O(n) e 20n = O(lg n).

5000lg n = O(n) è vero perchè esistono c ed n_0 tali che 5000lgn ≤ c n, basta prendere c = 5000 e la disuguaglianza è vera per ogni n ≥ 1. Infatti si ha 5000lg n ≤ 5000 n ⇔ lg n ≤ n.

20n = O(lg n) è falsa perché per ogni scelta di c si trova un valore di n per cui la condizione 20n ≤ c lg n è falsa. Infatti per n = 2^c si dovrebbe verificare che 20 2^c ≤ c lg 2^c ⇔ 20 2^c ≤ c^2 , che è banalmente falsa per ogni c ≥ 1

Esercizio 2b

2b.
Se si dimostra che un algoritmo ha tempo di esecuzione $\Theta(n^2)$ nel caso peggiore, è possibile che in qualche caso l'algoritmo termini in $\Theta(n \lg n)$ passi? Si motivi la risposta.

Sì è possibile perché il tempo di esecuzione nel caso peggiore fornisce un limite superiore ai tempi in tutti i casi.

Esercizio 3 testo A

Dato un array di interi A, diversi tra loro, diciamo che due elementi A[i] e A[j] di A, con A[i]<A[j], sono adiacenti se in A non ci sono elementi x compresi tra A[i] e A[j], cioè tali che A[i] < x < A[j] . Chiamiamo distanza tra due elementi di A la loro differenza in valore assoluto, cioè |A[i] - A[j]|.
Si determini la coppia di indici che individua gli elementi adiacenti a distanza massima in A. L'algoritmo dovrebbe terminare in O(nlgn).

Esempio:

A=

5	-10	2	-15	3	15	-8	12
0	1	2	3	4	5	6	7

Qui una coppia di elementi adiacenti a distanza massima, pari a 10, sono A[2] e A[6]

soluzioni

Per determinare una coppia di adiacenti a distanza massima, basterebbe ordinare l'array e calcolare la massima differenza tra elementi successivi. Ma si chiede di fornire in output gli indici dei due elementi e questa informazione andrebbe persa con l'ordinamento.
Si può risolvere il problema in due modi diversi.
In ogni caso si dovrà fare una copia degli elementi dell'array A in un secondo array, B, su cui eseguire l'ordinamento.
I soluzione:
passo 1: copia A in B, un nuovo array della stessa dimensione di A
passo 2: ordina B, in ordine crescente con l'heapsort
passo 3: trova i due elementi successivi in B la cui differenza in valore assoluto è massima, siano B[i] e B[i+1] .
passo 4: cerca B[i] e B[i+1] in A e dai in output la coppia di indici determinati.
II soluzione:
passo 1: considera un array B i cui elementi sono le coppie (A[i],i).
passo 2: ordina B, in ordine crescente sulla prima componente con l'heapsort
passo 3: trova i due elementi successivi in B la cui differenza in valore assoluto sulle prime componenti è massima, siano B[i] e B[i+1].
passo 4: dai in output la coppia delle seconde componenti di B[i] e B[i+1].

Analisi. A ha n elementi

I soluzione:
passo 1: copia A in B, un nuovo array della stessa dimensione di A
passo 2: ordina B, in ordine crescente con l'heapsort
passo 3: trova i due elementi successivi in B la cui differenza in valore assoluto è massima, siano B[i] e B[i+1] .
passo 4: cerca B[i] e B[i+1] in A e dai in output la coppia di indici determinati.

Il tempo di esecuzione asintotico è in O(nlgn), poiché i passi 1,3 e 4 sono tutti eseguiti in tempo lineare e il passo 2 in tempo O(nlgn). Si ha un'occupazione di spazio Θ(n).

II soluzione:
passo 1: considera un array B i cui elementi sono le coppie (A[i],i).
passo 2: ordina B, in ordine crescente sulla prima componente con l'heapsort
passo 3: trova i due elementi successivi in B la cui differenza in valore assoluto sulle prime componenti è massima, siano B[i] e B[i+1].
passo 4: dai in output la coppia delle seconde componenti di B[i] e B[i+1].

Il tempo di esecuzione asintotico è in O(nlgn), poiché i passi 1,3 e 4 sono tutti eseguiti in tempo lineare e il passo 2 in tempo O(nlgn). Si ha un'occupazione di spazio Θ(n).

Asintoticamente sono equivalenti, nel primo caso c'è un tempo di scorrimento dell'array A in più, nel secondo lo spazio necessario per il secondo array è maggiore.

Pseudocodice 1

```
AdMaxDiff1(A)
    Input: A è un  array
    postc: dà in output la coppia di indici di due elementi adiacenti a distanza massima.
    n =  A.size
    copia A in un nuovo array B di n elementi
    ordina B in ordine crescente, usando l'heapsort
    calcoliamo la massima distanza nel ciclo successivo:
    maxCorr = |B[0] - B[1]|
    j = 0
    for i = 1 to n-2 do
          if |B[i] -  B[i+1]| > maxCorr then
                maxCorr = |B[i] -  B[i+1]|
                j=i
          endif
    enfor
    i = 0
    while i≤n do
          if (B[j] = A[i]) then h = i
          if (B[j+1] = A[i]) then k = i
    endwhile
    return (h,k)
```

Pseudocodice 2

```
AdMaxDiff2(A)
    Input: A è un  array
    postc: dà in output la coppia di indici di due elementi adiacenti a distanza massima.
    n =  A.size
    crea un nuovo array B di n elementi, in cui B[i] = (A[i],i)
    B[i].1 è la prima componente e B[i].2 la seconda
    ordina B sulla prima componente in ordine decrescente, usando l'heapsort
    calcoliamo la massima distanza:
    maxCorr = |B[0].1 - B[1].1|
    j = 0
    for i = 1 to n-2 do
          if |B[i].1 -  B[i+1].1| > maxCorr then
                maxCorr = |B[i].1 -  B[i+1].1|
                j=i
          endif
return (B[j].2,B[j+1].2)
```

II soluzione migliorata

La seconda soluzione può essere migliorata, dal punto di vista del tempo di calcolo, inserendo il calcolo della distanza massima nel ciclo dell'heapsort.
II soluzione migliorata:
passo 1: considera un array B i cui elementi sono le coppie (A[i],i).
B[i].1 è la prima componente e B[i].2 la seconda
passo 2: trasforma B in un maxheap
sia n = A.length e d=0
passo 3: ripeti i seguenti passi per i = n fino a 1

x = B[1]
scambia B[1] e B[i] e decrementa i
B.heapsize = B.heapsize -1
esegui la maxheapify(B,1), che ripristina la proprietà di essere maxheap violata nella posizione 1 in B, modificata in modo da tenere conto solo della prima componente degli elementi di B.
y = B[1], y.1 è il massimo tra gli elementi
ancora da ordinare e il precedente di x.1 in B
se |x.1 - y.1| < d allora
h = x.2,
k = y.2,

Return (h,k)

I passi 1 e 2 sono lineari nel numero degli elementi di A, il passo 3 comporta O(n lg n) passi perché la maxheapify è eseguita in O(lg n) e n volte.