Note alla correzione dell'esame del 10 settembre 2020
Introduzione agli Algoritmi M-Z
Prof. Saverio Caminiti

Es. 1

Il costo di una singola esecuzione e dominato dal costo dei due for il primo eseguito n-volte e quello interno eseguito a-volte (non n come quello esterno). Il costo è descritto (usando la somma dei primi n numeri) così:

$\sum_{i=1..n} [\sum_{j=1..i} [c]] = \sum_{i=1..n} [ic] = c \sum_{i=1..n} [i] = c\, n(n+1)/2 = \Theta(n^2)$

La funzione fa una sola chiamata ricorsive su n/4, quindi T(n/4); molti hanno scritto 3T(n/4) questo è un errore **grave**: il x3 nel codice è semplicemente una moltiplicazione ed ha costo costante mentre scrivere 3T(n/4) significa triplicare il costo della chiamata ricorsiva.

In definitiva si otteneva:

$T(n) = T(n/4) + \Theta(n^2)$

$T(1) = \Theta(1)$ è sempre importante scrivere il costo del caso base

Risolvendo la versione semplificata (senza notazione asintotica) con il metodo iterativo (e non con albero della ricorsione o per sostituzione) si arrivava ad avere al generico passo i.

$T(n) = T(n/4^i) + \sum_{j=0..i-1} [c(n/4^j)^2]$

Al passo finale $k = \log_4 n$ si ottiene

$T(n) = T(1) + cn^2 \sum_{j=0..k-1} [(1/16)^j] = c' + cn^2 * d = \Theta(n^2)$

Infatti la serie geometrica di ragione 1/16 converge ad una costante.

Es. 2

Qui bastava adattare il merge per escludere dal risultato i valori che compaiono sia in A che in B. In pratica nel while si poteva aggiungere un if (A[i]==B[j]) {i++; j++;}. Così facendo in caso di valori che compaiono in entrambi gli array si incrementano entrambi gli indici per saltare quei valori senza includerli nel risultato.

In alternativa si poteva fare un merge normale e, scorrendo l'array risultante C, se si trovava C[i]==C[i+1] li si escludeva entrambi dall'array finale D (serviva un ulteriore array D in cui mettere i valori che apparivano in C senza duplicati).

Diversi hanno fatto ricorso (inutilmente) alla ricerca di valori – es. scorro A e per ogni valore x vedo se compare in B – questo è asintoticamente costo, quanto meno richiede una ricerca binaria.

Alcuni hanno calcolato solo A-B (gli elementi in A che non sono in B). Questa non è la differenza simmetrica: mancano gli elementi di B che non sono in A.

Qualcuno ha confuso indici e valori.

Gli indici sono numeri da 0 a n-1 (o da 1 a n se volte), i valori nell'array invece possono rappresentare molte cose, pensate ad esempio che siano il saldo di conti bancari espressi in €. Magari n = 200 e i valori invece possono raggiungere 1 milione di € o essere negativi. Confondere indici e valori quando si manipola un array è un errore grave.

Es 3.
A. basta una visita inorder per estrarre la sequenza ordinata delle chiavi di un ABR. Tempo Θ(n).

B. qui si poteva usare la seconda parte dello HeapSort: dato che lo Heap era già pronto non serviva la heapify, bastava fare il for che estrae il massimo, lo rimpiazza con una foglia e chiama fixHeap. Così si estraggono i valori dal maggiore al minore.
Tempo O(n log n).
Qui diversi hanno fatto cose strane tipo cercare ripetutamente il min nello heap o peggio ancora assumere che uno heap sia sempre un array ordinato al contrario: non è vero (es. [100, 50, 60, 11, 10, 4, 5] è uno heap valido).

Confrontando i tempi vediamo che la struttura migliore è l'ABR.