Corso di laurea in Informatica
Introduzione agli Algoritmi
A.A. 2024/2025

Esercizi riepilogativi

Tiziana Calamoneri

SAPIENZA
UNIVERSITÀ DI ROMA

Slides realizzate sulla base di quelle preparate da T. Calamoneri e G. Bongiovanni per il corso di Informatica Generale tenuto a distanza nell'A.A. 2019/20

*Esercizio svolto 1 (1)

Esercizio 1. Dato un array che contiene solo numeri negativi e positivi (nessun valore pari a zero), riorganizzarlo in modo che tutti i numeri negativi stiano a sinistra di quelli positivi.

Nota: Ordinare l'array risolve il problema, ma è inutilmente costoso. Usare il counting sort può non essere possibile (non abbiamo ipotesi sul range dell'input).

IDEA 1: Basta adattare la `Partition` del `Quicksort` in modo che scambi un positivo in posizione *i* con un negativo in posizione *j*

Esercizio svolto 1 (2)

```
def SeparaPosNeg1 (A):
i,j = 0,len(A)-1
while (i < j):
   while (A[j] > 0) and (i ≤ j):
      j -= 1
   while ((A[i] < 0)and(i ≤ j):
      i += 1
   if (i < j)
     A[i], A[j]= A[j], A[i]
```

Costo computazionale: Θ(n)

Esercizio svolto 1 (3)

IDEA 2. Il problema di separare gli elementi positivi dai negativi si può ridurre a quello di ordinare n numeri interi nel range [0,1]; si può quindi applicare una modifica dell'algoritmo di Counting Sort, in cui NON USIAMO I VALORI per cui non violiamo l'ipotesi, non serve l'array ausiliario C, perché non abbiamo dati satellite, ma serve comunque B

```
def  SeparaPosNeg2(A):
   B=[0]*len(A)
   neg, pos = 0,len(A)-1
   for i in range len(A):
    if A[i]>0:
       B[pos] = A[i]
       pos -= 1
    else:
       B[neg] = A[i]
       neg += 1
```

Costo computazionale: $\Theta(n)$ ma spazio $\Theta(n)$

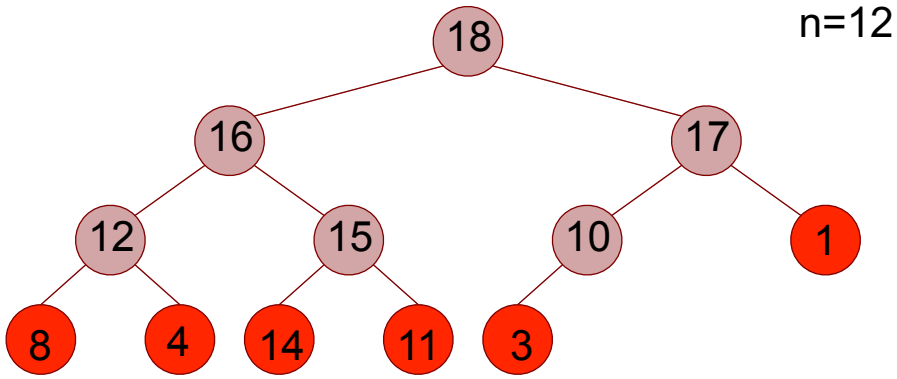
*Esercizio svolto 2 (1)

Esercizio 2. Progettare un algoritmo che, dato in input un array che rappresenta uno heap, restituisca il valore minimo. Fare le opportune considerazioni sul costo computazionale.

Idea:

- Il minimo va cercato nelle foglie dello heap; anche nel caso di ripetizioni del valore minimo, una di esse deve per forza trovarsi in una foglia;
- non c'è alcuna proprietà dello heap che ci aiuti a trovare il minimo fra i valori contenuti nelle foglie

Esercizio svolto 2 (2)



18	16	17	12	15	10	1	8	4	14	11	3

Esercizio svolto 2 (3)

- sappiamo solo che le foglie stanno negli elementi dell'array che occupano le posizioni da $\lfloor n/2 \rfloor$ (compreso) in avanti;
- quindi si deve usare un ciclo che, ispezionando le posizioni dell'array da $\lfloor n/2 \rfloor$ a $n-1$ compresi, individui il minimo.

```
def minInHeap(A,n):
    return min([A[i] for i in range(n//2,n)])
```

Costo computazionale: Θ(n).

*Esercizio svolto 3 (1)

Esercizio 3. Scrivere le funzioni Enqueue e Dequeue per una coda con priorità implementata su Max Heap (in cui la chiave massima ha priorità più alta).

IDEE:

- **Dequeue:** estraiamo la radice dello heap, che è il massimo; poi bisogna ripristinare lo heap (che ha un elemento in meno) usando esattamente la **Heapify()** già vista con lo Heapsort;

Esercizio svolto 3 (2)

- **Enqueue:** dobbiamo aggiungere un elemento allo heap.

NOTA: non va bene usare **`BuildHeap()`** perché lì partiamo dall'array disordinato e lo rendiamo interamente un heap, mentre qui abbiamo già uno heap e dobbiamo inserire un singolo elemento.

- Inseriamo il valore come nuova foglia (a destra dell'ultimo elemento presente nell'heap)
- Ripristiniamo l'heap dal basso verso l'altro, partendo dalla nuova foglia e risalendo (ci vuole una nuova funzione **`Heapify_bottom_up()`**).

Esercizio svolto 3 (3)

In sintesi:

Dequeue:

- Scambiare la chiave nella radice e la chiave in posizione **heap_size-1**;
- Il massimo estratto si trova in posizione **heap_size-1**
- Diminuire **heap_size** di 1
- Chiamare **Heapify()** sulla radice

…

Esercizio svolto 3 (4)

…

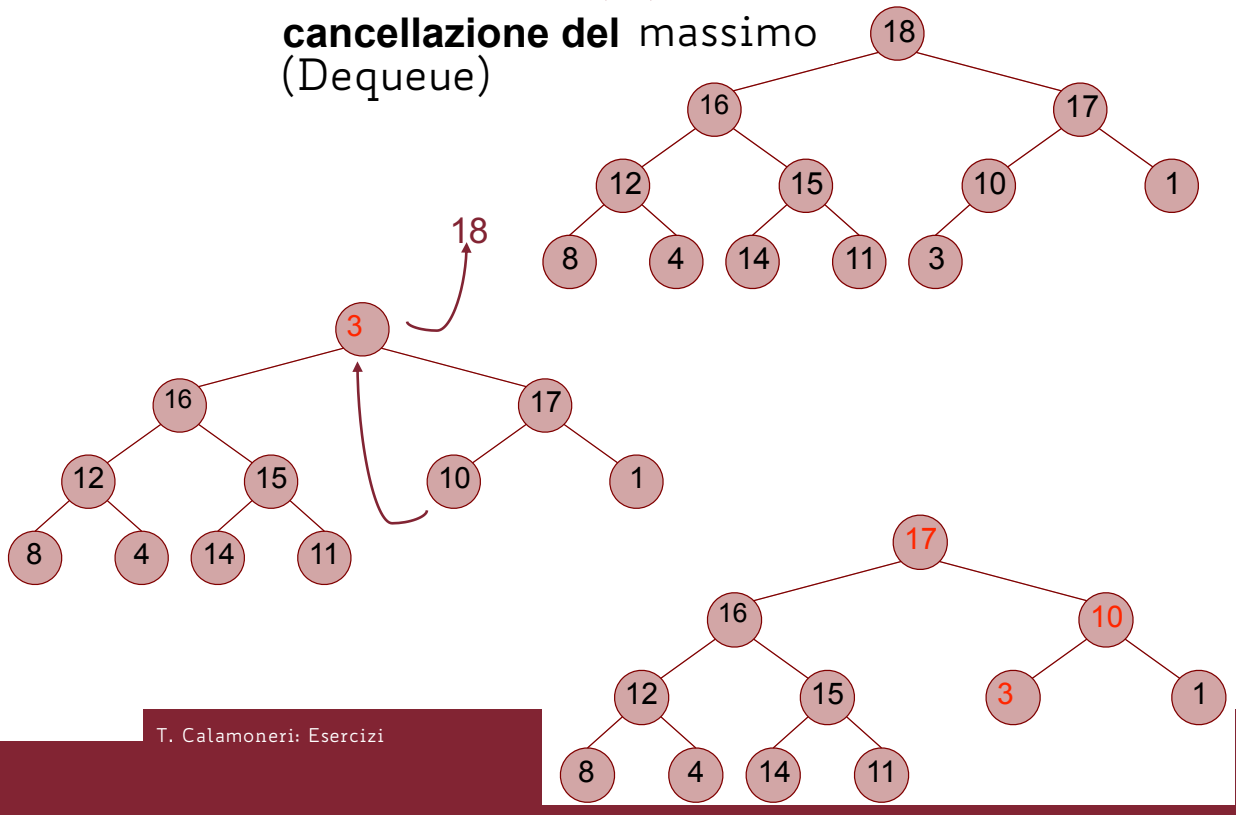
Enqueue:

- Ricopiare la nuova chiave nella posizione **heap_size**
- Aumentare **heap_size di 1**;
- Chiamare **Heapify_bottom_up()** sul nodo di indice **heap_size-1** (valore nuovo, già incrementato)

Come si vedrà, in questa implementazione entrambe le funzioni hanno costo *O(log n)*.

Esercizio svolto 3 (5)

cancellazione del massimo (Dequeue)



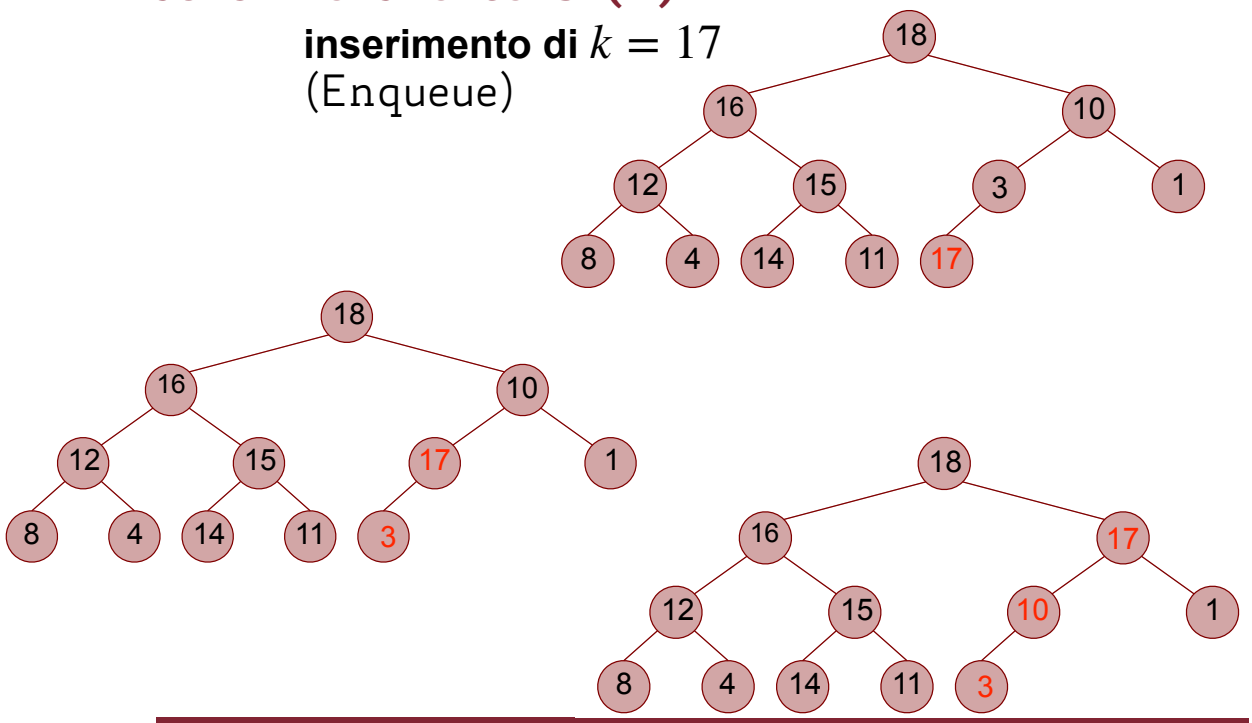
Esercizio svolto 3 (6)

```
def Dequeue (A,heap_size):
   if (heap_size == 0):
      scrivi "Errore: coda vuota"
      return
   k = A[0]
   A[0] = A[heap_size-1]
   heap_size = heap_size-1
   Heapify (A, heap_size)
   return k
```

Costo computazionale: O(log n)

Esercizio svolto 3 (7)

inserimento di $k = 17$
(Enqueue)



Esercizio svolto 3 (8)

Pseudocodice (ricorsivo) della funzione **`Heapify_bottom_up()`:**

```
def Heapify_bottom_up (A, heap_size)
   padre = ⌈heap_size/2⌉ − 1
   if (A[padre] < A[heap_size-1]):
      A[padre],A[heap_size-1]=A[heap_size-1],A[padre]
      if (padre > 0)
         Heapify_bottom_up (A, padre)
   return
```

Costo computazionale: O(log n)

Esercizio svolto 3 (9)

```
def Enqueue (A,heap_size,k):
   if (heap_size = len(A))
      scrivi "Errore: coda piena"
      return;
   A[heap_size] = k
   heap_size = heap_size+1
   if (heap_size > 0) //la coda non era vuota
      Heapify_bottom_up(A, heap_size)
   return
```

Costo computazionale: O(log n)

Nota su `Heapify_bottom_up()` (1)

Si noti che la funzione **`Heapify_bottom_up()`** può essere sfruttata sia per **eliminare un elemento qualunque** A[x] (non necessariamente la radice) dallo heap che per **modificare arbitrariamente un qualunque elemento** A[x] dell'heap.

In entrambi i casi si deve:

- sostituire l'elemento da cancellare A[x] con la foglia più a destra A[n-1] oppure modificare A[x];
- riaggiustare lo heap a partire da A[x] sia verso l'alto che verso il basso; infatti:

Nota su `Heapify_bottom_up()` (2)

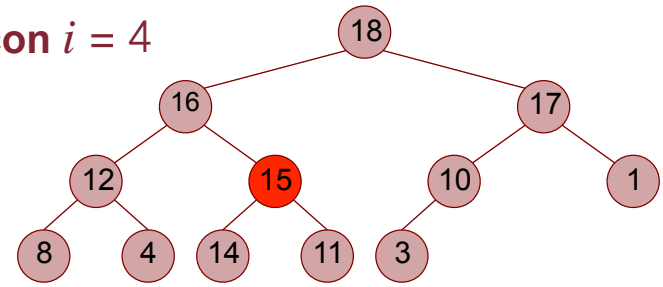
- se A[x] risulta minore del maggiore dei suoi figli, si può applicare la funzione di aggiustamento dell'heap top-down (**`Heapify`**);
- se A[x] sembra in posizione corretta rispetto ai suoi figli, non è ancora detto che l'heap sia corretto, bisogna infatti confrontarlo con il padre: se questo è minore di A[x] si deve procedere all'aggiustamento bottom-up (**`Heapify_bottom_up`**).

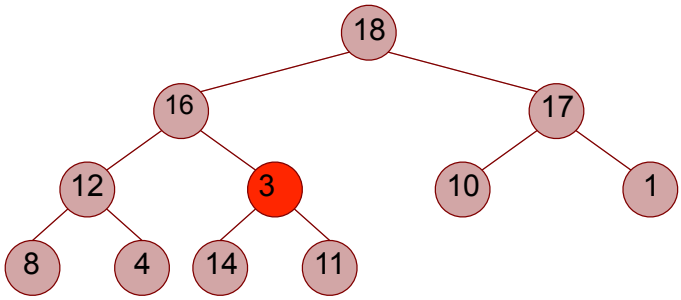
Nota: **si va verso l'alto <u>oppure</u> verso il basso, ma non è possibile che si debba andare in entrambe le direzioni.**

Costo computazionale: O(log n)

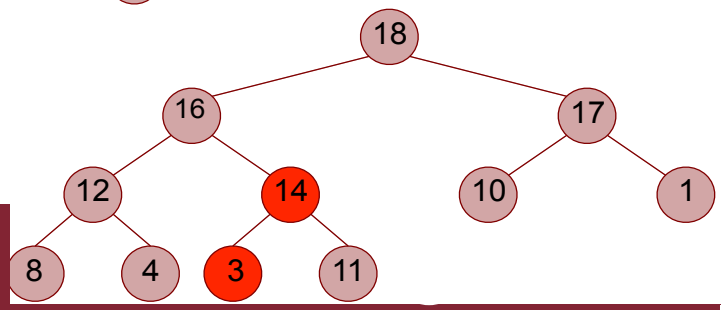
Nota su `heapify_bottom_up()` (3)

cancellazione di $A[i]$ con $i = 4$



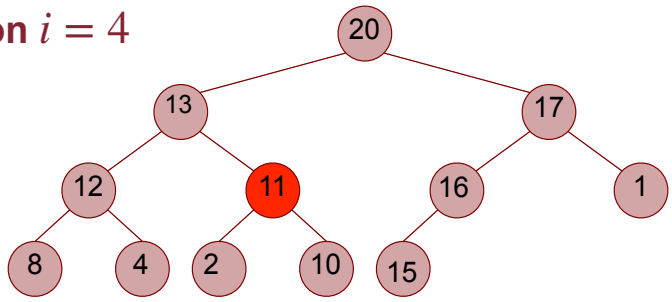


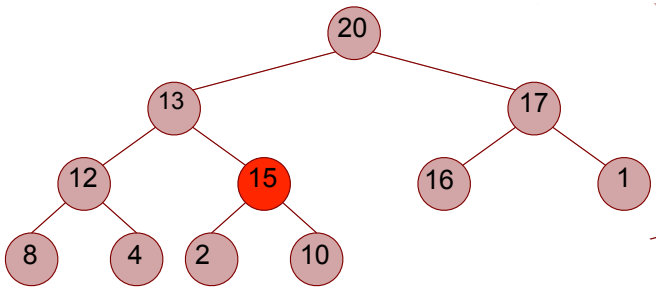
Dopo lo scambio, si aggiusta verso il basso:



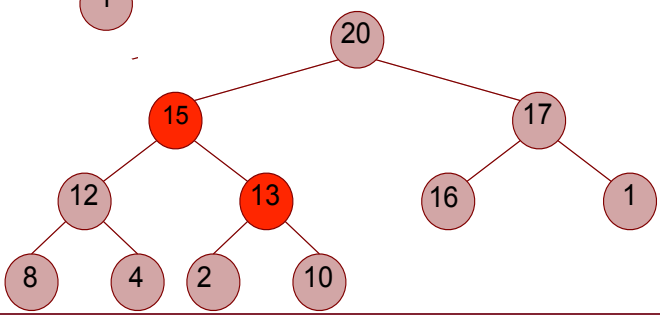
Nota su `heapify_bottom_up()` (4)

cancellazione di $A[i]$ con $i = 4$





Dopo lo scambio, si aggiusta verso l'alto:



Esercizio svolto 4 (1)

*Esercizio 4. Implementazione della visita in pre-order su un albero binario memorizzato con la notazione posizionale.

IDEE

- Operiamo un semplice adattamento della visita ricorsiva già vista.
- La differenza è semplicemente che, anziché seguire puntatori, si calcolano gli indici dei figli (controllando che esistano).
- Bisogna però anche controllare di non superare la fine dell'array (che supponiamo contenere n elementi)

Esercizio svolto 4 (2)

```
def Visita_preordine_NotPos (A,i):
   if ((i < len(A)) AND (A[i] ≠ '-')):
      %accesso al nodo e operazioni conseguenti
      Visita_preordine_NotPos (A;2*i+1)
      Visita_preordine_NotPos (A;2*i+2)
   return
```

Il costo computazionale è identico a quello della visita preorder già vista, e va dimostrato formalmente, essendo questa una funzione ricorsiva.

*Esercizio svolto 5 (1)

Esercizio 5.

In un albero binario, si definisca *n-sbilanciamento* di un nodo il valore assoluto tra il numero di nodi nel suo sottoalbero sinistro ed il numero di nodi nel suo sottoalbero destro.

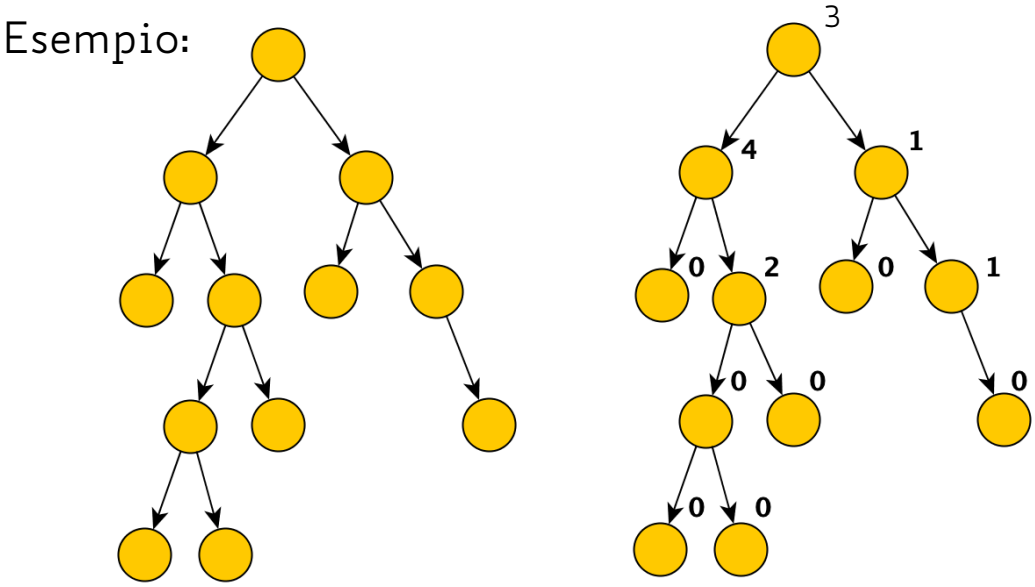
Dato un albero binario di cui si conosce il puntatore alla radice, trovare il massimo tra gli n-sbilanciamenti dei suoi nodi.

La funzione deve essere ricorsiva e NON fare uso di variabili globali.

…

Esercizio svolto 5 (2)

Esempio:



Esercizio svolto 5 (3)

IDEA

Ogni nodo riceve opportune informazioni da entrambi i suoi figli e poi effettua il calcolo seguendo la filosofia della visita in **post-ordine**.

Ogni nodo restituisce al padre sia l'n-sbilanciamento massimo per i nodi nel suo sottoalbero che il numero di nodi nel suo sottoalbero.

Al termine della visita la funzione restituirà l'n-sbilanciamento massimo (ed il numero di nodi dell'albero).

Esercizio svolto 5 (4)

```
def sbilanciamento(t):
      if (t == None):
            return 0,0
      max_sbil_sin,nodi_sin = sbilanciamento(t->left)
      max_sbil_des,nodi_des = sbilanciamento(t->right)
      massimo=max(abs(nodi_sin - nodi_des),
                              max_sbil_sin,max_sbil_des)
      return massimo,(nodi_sin + nodi_des +1)
```

Nota: La parte rossa calcola il numero di nodi dell'albero

Esercizio svolto 5 (5)

La funzione ricorsiva appena scritta, invocata su un nodo *x*, riceve ricorsivamente il numero di nodi ed il max n-sbilanciamento nei suoi due sottoalberi dai suoi figli, calcola il suo n-sbilanciamento ed il numero di nodi nel suo sottoalbero e restituisce tutto al padre.
Il costo computazionale è, come è ovvio, quello della visita di un albero, e l'equazione di ricorrenza relativa allo pseudocodice è:

$$T(n)=T(k)+T(n-1-k)+\Theta(1)$$
$$T(0)=\Theta(1)$$

che si può risolvere con il metodo di sostituzione dando come soluzione $\Theta(n)$.

*Esercizio svolto 6 (1)

Esercizio 6. Siano dati due array *A* e *B*, composti da $n \geq 1$ ed $m \geq 1$ numeri reali, rispettivamente. Gli array sono entrambi ordinati in senso crescente. *A* e *B* non contengono valori duplicati; tuttavia, uno stesso valore potrebbe essere presente una volta in *A* e una volta in *B*.
Progettare un algoritmo iterativo efficiente che stampi i valori che appartengono all'unione (intesa in senso insiemistico) di *A* e di *B*.
Ad esempio, se *A = [1,3,4,6]* e *B = [2,3,4,7]*, l'algoritmo deve stampare *1, 2, 3, 4, 6, 7*.
Di tale algoritmo:

- Si dia una descrizione a parole e si scriva lo pseudocodice;
- Si determini il costo computazionale, in funzione di *n* ed *m*.

Esercizio svolto 6 (2)

OSSERVAZIONE.

attenzione al problema dei duplicati: sappiamo che A e B contengono elementi tutti diversi tra loro ma possono esistere elementi che stanno sia in A che in B, e questi vanno stampati una volta sola.

IDEA 1.

- Prima considera gli elementi di A uno dopo l'altro e stampa solo quelli che non compaiono in B.
- Poi considera gli elementi di B non ancora stampati (?) uno dopo l'altro e stampali tutti.

Esercizio svolto 6 (3)

Implementazione 1: per ogni elemento di A, scorriamo tutto B per vedere se è presente: Costo computazionale: $\Theta(nm+m)=\Theta(nm)$, ma non stiamo usando l'ipotesi che i due array siano ordinati!

Implementazione 2: eseguiamo, per ciascun elemento di A, una ricerca binaria su B: costo computazionale $O(n \log m + m)$, ma ancora non stiamo sfruttando tutte le ipotesi perché così abbiamo usato il fatto che B sia ordinato ma l'ordinamento su A non ci serve a niente.

Esercizio svolto 6 (4)

IDEA 2.

Usiamo un algoritmo simile alla fusione di Mergesort, trascrivendo solo uno degli elementi uguali:

- *i* scorre *A* e *j* scorre *B* partendo entrambi da 0.
- Confrontiamo *A[i]* e *B[j]* :
  - se *A[i]<B[j]* si stampa *A[i]* e si incrementa *i*
  - se *A[i]>B[j]* si stampa *B[j]* e si incrementa *j*
  - se *A[i]=B[j]* si stampa *A[i]* e si incrementano *i* e *j*.
- Appena uno dei due array termina, stampiamo tutti gli elementi dell'altro.
- Oss. a differenza della fusione del MergeSort, non abbiamo qui bisogno di un array ausiliario.

Costo computazionale: $\Theta(n+m)$.

Esercizio svolto 6 (5)

```
def Stampa_unione(A, B):
    n, m = len(A), len(B); i, j = 0, 0
    while i < n AND j <m:
        if A[i] < B[j]:
             print( A[i]); i+=1
        elif A[i] > B[j]:
             print(B[j]); j+=1
       else:
             print( A[i]); i+=1; j+=1
    while i < n: # stampa la parte finale di A
        print(A[i]); i+=1
    while j < m: # stampa la parte finale di B
        print(B[j]); j+=1
```

Il costo di questo algoritmo è $\Theta(n+m)$ in quanto effettua una scansione di entrambi gli array esaminando ciascun elemento una e una sola volta in tempo $\Theta(1)$.

Esercizio svolto 7 (1)

Esercizio 7. Si risolva la seguente equazione di ricorrenza con due metodi diversi, per ciascuno dettagliando il procedimento usato:

$T(n) = T(n/2) + \Theta(n^2)$ *se* $n > 1$

$T(1) = \Theta(1)$

tenendo conto che n è una potenza di 2.

Soluzione

Utilizziamo dapprima il metodo principale, che è il più rapido, e poi verifichiamo la soluzione trovata con il metodo iterativo.

Esercizio svolto 7 (2)

Metodo principale:

Le costanti a e b del teor. princ. valgono qui 1 e 2, quindi $log_b\ a = 0$ ed $n^{log_b\ a} = 1$: ponendo $\varepsilon < 2$, siamo nel caso 3. del teorema, se vale che $af(n/b) \leq cf(n)$.

Poiché $f(n)$ è espressa tramite notazione asintotica, per verificare la disuguaglianza, dobbiamo eliminare tale notazione, e porre ad esempio $f(n) = \Theta(n^2) = hn^2$ e $h \geq 0$

Ci chiediamo se esista una costante $c<1$ tale che $h(n/2)^2 \leq c(hn^2)$.

Risolvendo otteniamo: $hn^2(1/4 - c) \leq 0$ che è verificata ad esempio per $c = 1/2$.

Ne possiamo dedurre che $T(n) = \Theta(n^2)$.

Esercizio svolto 7 (3)

Metodo iterativo:

Dall'equazione di ricorrenza deduciamo che:

- $T\left(\frac{n}{2}\right) = T\left(\frac{n}{2^2}\right) + \Theta\left(\left(\frac{n}{2}\right)^2\right)$, $T\left(\frac{n}{2^2}\right) = T\left(\frac{n}{2^3}\right) + \Theta\left(\left(\frac{n}{2^2}\right)^2\right)$

e così via. Sostituendo:

- $T(n) = T\left(\frac{n}{2}\right) + \Theta(n^2) = T\left(\frac{n}{2^2}\right) + \Theta\left(\left(\frac{n}{2}\right)^2\right) + \Theta(n^2) =$

$$= T\left(\frac{n}{2^3}\right) + \Theta\left(\left(\frac{n}{2^2}\right)^2\right) + \Theta\left(\left(\frac{n}{2}\right)^2\right) + \Theta((n)^2) = \ldots$$

$$= T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} \Theta\left(\left(\frac{n}{2^i}\right)^2\right)$$

Esercizio svolto 7 (4)

Metodo iterativo (segue):

Continuiamo ad iterare fino al raggiungimento del caso base, cioè fino a quando $n/2^k = 1$, il che avviene se e solo se $k = \log n$.

L'equazione diventa così:

$$T(n) = T(1) + \sum_{i=0}^{\log n - 1} \Theta\left(\left(\frac{n}{2^i}\right)^2\right) = \Theta(1) + n^2 \sum_{i=0}^{\log n - 1} \Theta\left(\frac{1}{4^i}\right)$$

Ricordando che $\sum_{i=0}^{b} a^i = \frac{a^{b+1}-1}{a-1}$ otteniamo infine:

$$T(n) = \Theta(1) + n^2\, \Theta\left(\frac{1-\left(\frac{1}{4}\right)^{\log n}}{1-\frac{1}{4}}\right) = \Theta(n^2)$$