Complementi di Algoritmi e Strutture Dati

(III anno Laurea Triennale - a.a. 2016/17)

Prova scritta 7 settembre 2017

NB: I punteggi sono indicativi.

Esercizio 1 - Analisi di algoritmi (punti 7) Consideriamo una versione dell'algoritmo di quicksort progettata per ordinare array dove gli elementi hanno *chiavi tutte distinte*. Questo significa che, scelto il pivot p, ogni altro elemento può essere solo $< p$ oppure $> p$.
Sotto tale condizione, quando partizioniamo una porzione di array lunga n caselle, dove ci sono m elementi $< p$ ($0 \leq m \leq n-1$ in quanto una casella è occupata dal pivot), alla fine del partizionamento gli elementi $< p$ dovranno occupare le prime m caselle.
In particolare, qui usiamo un algoritmo di partizionamento che prima conta quanti elementi sono $< p$ (suppomiano siano m), e poi scorre le prime m caselle dell'array spostando avanti (mediante scambio) gli elementi che sono $> p$.
Lo pseudocodice che partiziona la porzione di array $a[\texttt{inf}..\texttt{sup}]$ è il seguente:

```
p=a[inf] //scelta del pivot
//conto elementi < p
m=0
for(k=inf+1; k<=sup; k++)
  if (a[k]<p) m++
//fine conta
i=inf+1; j=inf+m+1
//Pre
while (i≤ inf+m) //Inv
 if (a[i]<p) i++
 else
    swap(a, i, j)
    j++
//Post
swap(a, inf, inf+m) //metto a posto il pivot
```

1. Assumendo che la conta sia giusta, si dimostri la correttezza del ciclo di scambi, ovvero che alla fine (prima di mettere a posto il pivot) vale la postcondizione

$$Post = a[\texttt{inf} + 1..\texttt{inf} + m] < p \wedge a[\texttt{inf} + m + 1..\texttt{sup}] > p$$

   A tale scopo, si dica quale precondizione *Pre* vale all'inizio del ciclo, si descriva un'opportuna invariante *Inv* (eventualmente anche a parole o graficamente), e si provi che vale all'inizio, che si preserva a ogni iterazione e che congiuntamente alla condizione di uscita dal ciclo implica la postcondizione.

2. Dimostrare che tutta la procedura di partizionamento ha complessità temporale nel caso peggiore in $O(n)$ con $n = \texttt{sup} - \texttt{inf} + 1$ la lunghezza della porzione di array partizionata.

Esercizio 2 - Sorting (punti 5) Si consideri il seguente array:

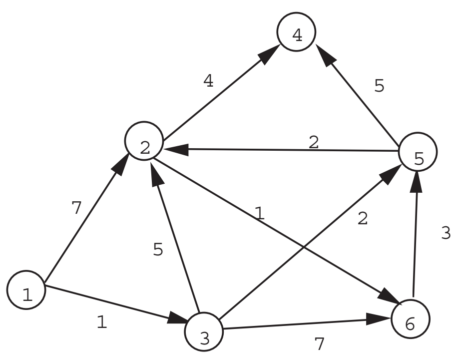
12	30	10	13	40	13	40	3	20	17

Eseguire la prima fase dell'algoritmo heapsort, cioè quella che trasforma l'array in uno heap a massimo. Si chiede di eseguirla

- PREFERIBILMENTE con la procedura *heapify*.
  Ad ogni passo disegnare tutto l'array come albero ed indicare quali sotto-parti sono già heap.
- IN ALTERNATIVA (ma allora l'esercizio vale un punto in meno) con una serie di chiamate a *insert*.
  Per ogni chiamata indicare l'elemento che viene inserito e disegnare lo heap in cui viene inserito prima e dopo l'operazione (ovviamente il "dopo" di un inserimento è il "prima" dell'inserimento seguente e non occorre ripetere il disegno).

Ricordare che deve essere uno heap *a massimo*.

Esercizio 3 - Grafi (Punti 6) Si consideri il seguente grafo orientato e pesato.



Applicando l'algoritmo di Dijkstra, si determinino i pesi dei cammini minimi che collegano il vertice 1 con tutti gli altri vertici. Più precisamente, si completi la seguente tabella:

	1	2	3	4	5	6
	0	∞	∞	∞	∞	∞
	...					

dove ogni riga corrisponde a un'iterazione, e ogni casella contiene: per i nodi per i quali è già stata trovata la distanza definitiva, un simbolo speciale (per esempio -), per gli altri la distanza provvisoria corrente.

Esercizio 4 - NP-completezza (Punti 7) Si considerino i seguenti due problemi decisionali: dato in input un grafo non orientato G, in *ODD-MAX-CLIQUE* ci chiediamo se la dimensione massima di una clique in G è dispari, mentre in *EVEN-MAX-CLIQUE* ci chiediamo se la dimensione massima di una clique in G è pari.

1. Si dia una riduzione polinomiale da *ODD-MAX-CLIQUE* in *EVEN-MAX-CLIQUE*.
2. Si provi che se *EVEN-MAX-CLIQUE* è NP-completo allora lo è anche *ODD-MAX-CLIQUE*.

Esercizio 5 - Tecniche algoritmiche (Punti 8) Si consideri il problema di trovare, data una sequenza $X[1..n]$, la lunghezza della massima sottosequenza palindroma di X, dove una sequenza è palindroma se resta la stessa letta da destra a sinistra. Per esempio, se $X = [A, B, C, D, C, E, A]$, la massima lunghezza di una sottosequenza palindroma è 4, corrispondente alla sottosequenza $[A, C, C, A]$. Indichiamo con $P[i, j]$, con $1 \leq i \leq j \leq n$, il sottoproblema di trovare la la lunghezza della massima sottosequenza palindroma di $X[i..j]$.

1. Si definisca induttivamente $P[i, j]$ giustificando la correttezza della definizione.
2. Si descriva un algoritmo di programmazione dinamica basato sulla definizione data al punto precedente che calcoli $P[1, n]$.
3. Si valuti la complessità di tale algoritmo.