Complementi di Algoritmi e Strutture Dati

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

Soluzioni prova scritta 7 settembre 2017

Esercizio 1 - Analisi di algoritmi 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.

   Diamo *Pre*, *Inv* e *Post*:

   Pre: $i = \texttt{inf}+1 \wedge j = \texttt{inf}+m$

graficamente:

inf	i ...	inf+m	j ...	sup
p	??		??	

Inv: $a[\texttt{inf}+1..i-1] < p \wedge a[\texttt{inf}+m+1..j-1] > p \wedge i \leq \texttt{inf}+m+1 \wedge j \leq \texttt{sup}+1$

graficamente:

inf	inf+1 ...	i ...	inf+m	inf+m+1 ...	j ...	sup
p	$< p$	??		$> p$	??	

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

graficamente:

inf	inf+1 ...	inf+m	inf+m+1 ...	sup
p	$< p$		$> p$	

Pre implica *Inv*:

Banalmente perché $a[\texttt{inf}+1..\texttt{inf}]$ e $a[\texttt{inf}+m+1..\texttt{inf}+m]$ sono vuote.

Inv si mantiene nel ciclo:

Se $a[i] < p$ la parte dei $< p$ guadagna un elemento, e correttamente faccio $i++$. Se $a[i] > p$ dopo lo scambio la la parte dei $> p$ guadagna un elemento, e correttamente faccio $j++$.

$Inv \wedge \neg(i \leq \texttt{inf}+m)$ implica *Post*:

Essendo (per *Inv*) $i \leq \texttt{inf}+m+1$, deve essere $i = \texttt{inf}+m+1$. Allora la prima parte di *Inv* diventa $a[\texttt{inf}+1..\texttt{inf}+m] < p$ che è la prima parte di *Post*. Essendoci questi m gli unici elementi $< p$, ne segue che gli elementi nelle caselle da $\texttt{inf}+m+1$ in poi dovranno essere $> p$.

Nota: in *Inv* la parte $a[\texttt{inf}+m+1..j-1] > p$ in realtà non è necessaria, ma è stata messa per chiarezza. Non la usiamo per dimostrare che alla fine vale *Post*, in quanto non sufficiente a tale scopo. Infatti j non necessariamente ha raggiunto la fine della porzione di array. Nel caso $m = n/2$ e tutti gli elementi $< p$ nella prima metà, j non viene incrementato nemmeno una volta!

TERMINAZIONE (non era richiesta):

Consideriamo la coppia $(\texttt{inf}+m-i, n-j)$ e l'ordinamento lessicografico sulle coppie.

Tale quantità decresce ad ogni ciclo poichè o aumenta i oppure aumenta j.

j può essere incrementato al più $n-m-1$ volte (perché ci sono in totale n elementi di cui m sono $< p$, uno $= p$ è il pivot), per cui $j \geq \texttt{inf}+m+1+n-m-1 = \texttt{inf}+n$. Quindi il primo elemento della coppia $(n-j)$ vale almeno $n - \texttt{inf} - n = -\texttt{inf}$.

Inoltre mentre la condizione del while $(i \leq \texttt{inf}+m)$ resta vera, il secondo elemento della coppia $(\texttt{inf}+m-i)$ vale almeno $0-(n-m-1) = n-m+1$.

Abbiamo dimostrato che la coppia decresce ad ogni ciclo ed è limitata inferiormente.

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.

   Le istruzioni al di fuori dei cicli sono solo assegnazioni e (alla fine) uno scambio, tutte di costo costante.

   Banalmente il ciclo for (che calcola m) viene eseguito $m-1$ volte ed esegue ad ogni giro un'istruzione di costo costante, per cui ha costo in $O(n)$.

   Il ciclo while viene eseguito al più $n-1$ volte perché:

   – se $a[i] > 0$ incrementiamo i per cui l'elemento viene scavalcato e non sarà esaminato più; quindi questo caso può capitare solo m volte in quanto ci sono m elementi $< p$;

   – altrimenti l'elemento viene messo in una posizione j che non sarà mai raggiunta da i (j parte già $> \texttt{inf}+m$ e viene incrementato), per cui l'elemento non sarà esaminato più; questo caso può capitare solo $n-m-1$ volte perché ci sono $n-m-1$ elementi $> p$.

Le istruzioni eseguite ad ogni giro hanno costo costante, per cui il costo del ciclo while è $O(n)$.

Nota: alcuni hanno esplicitamente o implicitamente assunto che il caso peggiore sia quello in cui $m = n-1$ (tutti gli elementi tranne il pivot sono $< p$) e poi hanno dato la complessità di questo caso.
Non è il caso peggiore. In questo caso, ho $n-1$ cicli che tutti trovano $a[i] < p$ ed eseguono un incremento $(i++)$; quindi in totale abbiamo $(n-1)$ volte una assegnazione.
Nel caso, per esempio, in cui $m = n/2$ e tutti gli elementi $< p$ sono in fondo, il ciclo itera m volte eseguendo ogni volta un incremento e uno scambio (che, per essere pignoli, sono 3 assegnazioni); in totale abbiamo $n/2$ volte 4 assegnazioni, quindi costa circa il doppio del caso "presunto" peggiore! All'ordine di grandezza sempre $O(n)$ ...

Esercizio 2 - Sorting 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*.
Versione con heapify:

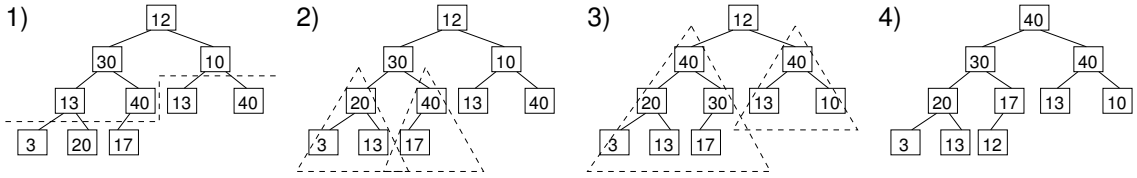
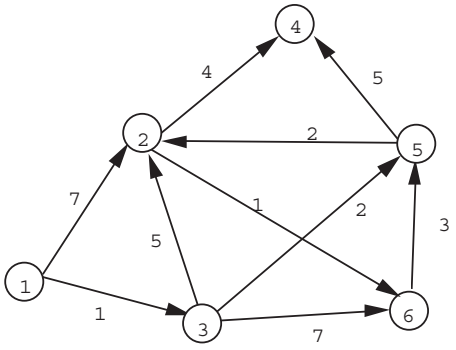


Fig.1) mostra la rappresentazione ad albero dell'array. I sottoalberi con radici le foglie (nodi sotto la linea tratteggiata) sono già heap.
Ora, per livelli risalenti, renderemo heap gli altri sottoalberi, per ognuno eseguendo `moveDown` della sua radice.
Nota: eseguita su un nodo n che contiene una chiave k, la `moveDown` fa scendere k scambiandola con la chiave k' che si trova nel figlio di n con chiave massima, e iterando il procedimento fino a che o tutti i figli del nodo corrente hanno chiavi $< k$ oppure il nodo corrente è foglia.
Prima rendiamo heap i sottoalberi con radici i nodi non foglie del penultimo livello (indicati da triangoli in Fig.2). Per far questo eseguiamo `moveDown`(13), che scambia 13 con 20, e `moveDown`(40) che non fa scambi.
Poi rendiamo heap i sottoalberi con radici i nodi non foglie del terzultimo livello (indicati da triangoli in Fig.3). Per far questo eseguiamo `moveDown`(30) che scambia 30 con 40, e `moveDown`(10) che scambia 10 con 40.
Poi l'intero albero. Eseguiamo `moveDown`(12) che scambia 12 con uno dei due 40: qui suppongo quello di sinistra (va bene anche l'altra scelta), poi con 30 e infine con 17. Otteniamo il risultato in Fig.4).

Esercizio 3 - Grafi 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. Nella prima colonna indichiamo il nodo estratto.

	1	2	3	4	5	6
	0	∞	∞	∞	∞	∞
1	-	7	1	∞	∞	∞
3	-	6	-	∞	3	8
5	-	5	-	8	-	8
2	-	-	-	8	-	6
6	-	-	-	8	-	-
4	-	-	-	-	-	-

Esercizio 4 - NP-completezza 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*.
   Dato un grafo G (istanza di *OMC*), consideriamo il grafo G' ottenuto aggiungendo a G un nuovo nodo, sia u, e archi da u a ogni nodo di G. Chiaramente questa trasformazione può essere eseguita in tempo polinomiale rispetto al numero di nodi e archi di G. Allora, è facile vedere che, data una clique di dimensione massima (sia k) in G, aggiungendo u si ha una clique di dimensione massima $(k+1)$ in G', e viceversa una clique di dimensione massima in G' contiene necessariamente u ed eliminandolo si ottiene una clique di dimensione massima in G. Quindi la dimensione massima di una clique in G è dispari se e solo se la dimensione massima di una clique in G è pari.

2. Si provi che se *EMC* è NP-completo allora lo è anche *OMC*.
   Supponiamo che *EMC* sia NP-completo. Avendo provato $OMC \leq_P EMC$, sappiamo che

OMC è in NP. Inoltre, dato che in modo analogo possiamo provare $EMC \leq_P OMC$, sappiamo che *OMC* è NP-hard.

Esercizio 5 - Tecniche algoritmiche 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 $1 \leq i, j \leq n$, 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.
   La massima sottosequenza palindroma di una sequenza vuota o di un elemento è chiaramente la sequenza stessa:

   $P[i, j] = 0$ per $i, j \in 1..n, j < i$
   $P[i, i] = 1$ per $i \in 1..n$

   Nel caso di una sequenza di almeno due elementi, abbiamo due casi.
   Se il primo e l'ultimo elemento sono uguali, la massima sottosequenza palindroma si ottiene aggiungendo tali elementi a una massima sottosequenza palindroma della sequenza formata dai restanti elementi:

   $$P[i, j] = P[i + 1, j - 1] + 2 \text{ se } X[i] = X[j], \text{ per } i, j \in 1..n,\ j > i + 1$$

   Se invece il primo e l'ultimo elemento sono diversi, una sottosequenza palindroma si ottiene solo considerando gli elementi da i a $j - 1$ oppure da $i + 1$ a j:

   $$P[i, j] = \max\{P[i, j - 1], P[i + 1, j]\} \text{ se } X[i] \neq X[j], \text{ per } i, j \in 1..n,\ j > i + 1$$

2. Si descriva un algoritmo di programmazione dinamica basato sulla definizione data al punto precedente che calcoli $P[1, n]$.
   L'algoritmo deriva direttamente dalla definizione induttiva data al punto precedente.

```
//P[1..n,1..n]
for (i=1; i<=n; i++)
  for (j=1; j<i; j++) P[i,j]=0
for (i=1; i<=n; i++) P[i,i]=1
for (i=1; i<=n-2; i++)
  for (j=i+2; j<=n; j++) P[i,j]=max(P[i,j-1]
```

3. Si valuti la complessità di tale algoritmo.
   L'algoritmo costruisce una matrice $n \times n$, quindi la sua complessità (temporale e spaziale) è chiaramente $\Theta(n^2)$.