Dati e Algoritmi (Pietracaprina)

Scritto 21/01/2026 (Seconda Parte)

SOLUZIONI

Esercizio 1 Sia A una matrice $n \times m$ di bit dove ogni riga è ordinata in senso crescente (prima gli 0 e poi gli 1). Il seguente algoritmo determina il massimo numero di 0 in una riga:

$i \leftarrow 0$; $j \leftarrow 0$;
while $((i < n)$ AND $(j < m))$ **do**
  if $(A[i, j] = 0)$ **then** $j \leftarrow j + 1$;
  else $i \leftarrow i + 1$;
return j

Determinare un opportuno invariante per il **while**, che sia utile per provare la correttezza dell'algoritmo.

Soluzione. Alla fine di una generica iterazione del **while**, quando il prossimo elemento da esaminare è $A[i, j]$, vale il seguente invariante

- j è pari al massimo numero di 0 in una riga della sottomatrice $(i + 1) \times j$ costituita dagli elementi $A[r, c]$ con $0 \leq r \leq i$ e $0 \leq c < j$;
- Nella colonna j vale che $A[r, j] = 1$, per ogni $0 \leq r \leq i - 1$.

□

Esercizio 2 Sia T un albero binario proprio dove ogni nodo $v \in T$ contiene un valore intero v.`val`. Un nodo v si dice *ordinato da sinistra* se il massimo intero contenuto nel sottoalbero sinistro di v è $< v$.`val`. Un nodo foglia è considerato sempre ordinato da sinistra.

a. Progettare in pseudocodice un algoritmo ricorsivo `SetLeftSorted` che per ogni nodo $v \in T$ imposta un campo v.`LeftSorted` a `true`, se v è ordinato da sinistra, e a `false` altrimenti. Specificare attentamente l'input e l'output della generica chiamata ricorsiva.

b. Analizzare la complessità dell'algoritmo `SetLeftSorted` progettato.

Soluzione.

a. L'algoritmo è basato su una visita in postorder di T. Lo pseudocodice è il seguente:

Algoritmo `SetLeftSorted`(v)

Input: Nodo $v \in T$
Output: max u.`val` con $u \in T_v$, e u.`LeftSorted` impostato correttamente $\forall u \in T_v$.

if (T.`isExternal`(v)) **then**
  v.`LeftSorted` $\leftarrow$ `true`;
  return v.`val`
$m_L \leftarrow$ `SetLeftSorted`(T.`left`(v));
$m_R \leftarrow$ `SetLeftSorted`(T.`right`(v));
if ($m_L < v$.`val`) **then** v.`LeftSorted` $\leftarrow$ `true`;
else v.`LeftSorted` $\leftarrow$ `false`;
return $\max\{m_L, m_R, v.\texttt{val}\}$

La prima invocazione è `SetLeftSorted`(T.`root`()).

b. L'algoritmo ha la stessa struttura della visita in postorder e in ogni chiamata ricorsiva esegue $\Theta(1)$ operazioni, oltre alle eventuali chiamate ricorsive al suo interno. Quindi la complessità di `SetLeftSorted`(T.`root`()) è $\Theta(n)$, dove n è il numero di nodi di T.

□

Esercizio 3 Sia $G = (V, E)$ un grafo non diretto con n vertici ed m archi. Un triangolo per G è una terna di vertici u, v, w tali che E contiene i 3 archi $(u, v), (u, w)$ e (v, w).

a. Se la terna u, v, w è un triangolo, come viene etichettato l'arco (v, w) da `BFS`(G, u)? Motivare la risposta.

b. Usando il punto precedente, progettare un algoritmo che determina, in tempo $O(n + m)$, il numero di triangoli di G che contengono u.

Soluzione.

a. Si consideri l'esecuzione di `BFS`(G, u). Il vertice u è inserito in L_0 e quando viene esaminato, sia v che w vengono inseriti in L_1 impostando il loro `ID` a 1. Nella successiva scansione di L_1, quando si esamina il primo tra v e w, ad esempio v, w.`ID` è già impostato a 1, e quindi l'arco (v, w), che a questo punto non è ancora stato visitato, viene etichettato come `CROSS EDGE`.

b. Dal punto precedente, è immediato dedurre che il numero di triangoli di G che contengono un dato vertice u è pari *esattamente* al numero di archi (v, w) che `BFS`(G, u) etichetta come `CROSS EDGE` e tali che $v, w \in L_1$. Supponiamo di fare una lieve modifica alla `BFS` che per ogni $v \in V$ imposta un campo v.`level` pari a i se v è inserito nel livello

L_i. Tale modifica chiaramente non ne altera la complessità. L'algoritmo richiesto è il seguente:

Algoritmo `TriangleCount`(G, u)

Input: `Grafo` $G = (V, E)$ `e vertice` $u \in V$
Output: `Numero di triangoli in` G `che contengono` u

forall $(v \in V)$ **do** $\{v$.`ID` $\leftarrow 0$; v.`level` $\leftarrow$ `null`$\}$;
forall $(e \in E)$ **do** e.`label` $\leftarrow$ `null` ;
`BFS`(G, u); `// BFS modificata`
`count` $\leftarrow 0$;
forall $(e \in E)$ **do**
  `Let` $e = (v, w)$;
  if $(v$.`level` $= 1)$ AND $(w$.`level` $= 1)$ AND $(e$.`label` $=$ CROSS EDGE$)$ **then**
    `count` $\leftarrow$ `count`$+1$;
return `count`

L'algoritmo ha la complessità desiderata dato che l'inizializzazione, la `BFS`, e il ciclo finale richiedono $O(n + m)$ operazioni.

□