# Dati e Algoritmi I (Pietracaprina)

## Scritto Completo e Compitino 23/01/2020 (Seconda Parte)

## SOLUZIONI

**Problema.** Sia $A = A[0], A[1], \ldots, A[n-1]$ un array di interi distinti, ordinato in senso crescente, e $x$ un valore intero. Il seguente algoritmo restituisce l'indice di $x$ in $A$, se $x$ appare in $A$, e $-1$ altrimenti.

```
a ← 0; b ← n − 1;
while a ≤ b do
    c ← ⌊(a + b)/2⌋;
    if A[c] = x then return c;
    else
        if A[c] < x then a ← c + 1;
        else b ← c − 1;
return −1;
```

Identificare un opportuno invariante per il ciclo che serva per provare la correttezza dell'algoritmo (la prova di correttezza non è richiesta).

**Soluzione.** L'invariante è il seguente: $k \neq A[i]$ per ogni $i \in [0, a-1] \cup [b+1, n-1]$. □

**Problema.** Progettare un algoritmo *non ricorsivo* `SearchMinAB` che dato un albero binario di ricerca $T$ contenente entry con chiavi distinte e un intervallo $[A, B]$, restituisca la più piccola chiave $k \in [A, B]$ presente in $T$, se ne esiste una, e `null` altrimenti. Descrivere l'algoritmo in pseudocodice e analizzarne la complessità.

**Soluzione.** L'algoritmo si ottiene adattando l'algoritmo progettato per il Problema 9 della dispensa di esercizi sulle Mappe.

**Algoritmo** `SearchMinAB`$(T, A, B)$

**Input:** ABR $T$ con chiavi distinte e intervallo $[A, B]$

**Output:** Minima chiave $k \in [A, B]$ presente in $T$, se esiste, o null altrimenti

```
v ← T.root();
min-k ← null;
while (T.isInternal(v)) do
    k ← v.getElement().getKey();
    if (k < A) then v ← T.right(v);
    else
        if (k ∈ [A, B]) then min-k ← k;
        v ← T.left(v)
return min-k
```

Riguardo alla complessità, dato che il nodo $v$ scende di un livello ad ogni iterazione del **while**, il numero di iterazioni è proporzionale al più all'altezza di $T$, e in ciascuna iterazione

si esegue un numero costante di operazioni. Al di fuori del ciclo, si eseguono $O(1)$ operazioni. La complessità è quindi $O(h)$, con $h$ altezza di $T$. □

**Problema.** Sia $G = (V, E)$ un grafo che rappresenta una rete sociale con $n$ vertici ed $m$ archi. Ogni vertice $u \in V$ ha un campo $L_V[u]$.`influencer` che vale 1 se $u$ è un influencer e 0 altrimenti. Un vertice $x$ non influencer si dice *influenzabile* se esiste un influencer $y$ e un cammino tra $x$ e $y$. Progettare in pseudocodice un algoritmo `Influenzabili` che conti il numero di vertici influenzabili in $G$, e analizzarne la complessità. (Per avere punteggio pieno la complessità deve essere $O(n + m)$.)

**Soluzione.** L'idea è quella di invocare una BFS per ciascuna componente connessa che contiene almeno un influencer (partendo da uno degli influencer della componente) facendole restituire il numero di vertici non influencer della componente, che viene accumulato in un contatore. Si supponga di modificare `BFS`$(G, s)$ in modo che restituisca il numero di vertici non influencer visitati. L'algoritmo richiesto è il seguente.

**Algoritmo** `Influenzabili`$(G)$

**Input:** `Grafo` $G = (V, E)$ `non diretto con vertici etichettati come influencer/non influencer`
**Output:** `Numero di vertici influenzabili`

```
count ← 0;
for v ← 1 to n do
    if (L_V[v].ID = 0) AND (L_V[v].influencer = 1) then
        count ← count+BFS(G, v)
return count
```

È facile vedere che le modifiche richieste alla BFS non ne alterano la complessità. Dato che la BFS viene invocata al più una volta per ogni componente connessa la complessità è $\Theta(n + m)$. □