Dati e Algoritmi (Pietracaprina)

Scritto 11/02/2026 (Seconda Parte)

SOLUZIONI

Esercizio 1 Il seguente algoritmo ricorsivo determina se un array di bit $S = S[0]S[1]\cdots S[n-1]$ contiene due 1 consecutivi.

Algoritmo `find`(S)

$n \leftarrow |S|$; $m = \lfloor n/2 \rfloor$;
if $(n < 2)$ **then return** `false`;
if $S[m-1] \cdot S[m] = 1$ **then return** `true`;
else
  $S_L \leftarrow S[0 \div m-1]$; $S_R \leftarrow S[m \div n-1]$;
  return `find`(S_L) `OR find`(S_R)

Dimostrare la correttezza dell'algoritmo per induzione su $n \geq 0$.

Soluzione.

- **BASE**. È immediato vedere che l'algoritmo è corretto per array di taglia $n = 0, 1$.
- **PASSO INDUTTIVO**. Fissiamo un valore $n \geq 1$ arbitrariamente e assumiamo, come ipotesi induttiva, che per tutti gli array di taglia $\leq n$ l'algoritmo sia corretto. Sia S un array di taglia $n+1 \geq 2$. È facile vedere che se $S[m-1] \cdot S[m] \neq 1$, allora se la coppia di 1 consecutivi esiste ed è contenuta in S deve comparire a sinistra di m, e quindi in S_L, oppure a destra di $m-1$, e quindi in S_R (o sia in S_L che in S_R). Dato che $|S_L|, |S_R| < n$, si può applicare l'ipotesi induttiva e quindi il risultato è corretto.

□

Esercizio 2 Sia T un Albero Binario di Ricerca in cui ogni nodo interno v contiene una entry (k_v, L_v), dove il valore L_v è una *lista non ordinata* di $m > 1$ interi e la chiave k_v è il minimo intero in L_v. Gli $n \cdot m$ interi nelle n liste sono tutti distinti. Inoltre, vale che se due entry (k_1, L_1) e (k_2, L_2) hanno $k_1 < k_2$, allora tutti gli interi in L_1 sono $<$ di tutti gli interi in L_2. Cioè, le liste definiscono intervalli disgiunti di $\mathbb{Z}$.

a. Progettare in pseudocodice un algoritmo ricorsivo `TreeListSearch` che dato T e un intero x restituisce un nodo $v \in T$ che contiene una entry (k_v, L_v) con $x \in L_v$, se ne esiste uno, e restituisce `null` altrimenti. Le operazioni sulle liste L_v possono essere espresse a parole. **Suggerimento:** usare una variante della TreeSearch vista a lezione.

b. Assumendo $m = 3$, analizzare la complessità di `TreeListSearch`.

c. Come cambia la complessità per m generico?

Soluzione.

a. L'algoritmo è una modifica della `TreeSearch` vista a lezione.

Algoritmo `TreeListSearch`(v, x)

Input: nodo $v \in T$, intero x
Output: $u \in T_v$ con $x \in L_u$, se esiste, o null altrimenti.

if (T.`isExternal`(v)) **then return** null;
$k_{\texttt{min}} \leftarrow$ minimo intero in L_v;
$k_{\texttt{max}} \leftarrow$ massimo intero in L_v;
if ($k_{\texttt{min}} \leq x \leq k_{\texttt{max}}$) **then**
  if $x \in L_v$ **then return** v;
  else return null;
else
  if ($x < k_{\texttt{min}}$) **then return** `TreeListSearch`(T.`left`$(v), x$);
  else return `TreeListSearch`(T.`right`$(v), x$);

b. L'algoritmo ha la stessa struttura della `TreeSearch` vista a lezione. In particolare, l'albero della ricorsione ha $\leq h + 1$ nodi, dove h è l'altezza di T, e il costo associato a ciascun nodo è dominato dal calcolo di $k_{\texttt{min}}$ e $k_{\texttt{max}}$, e alla eventuale ricerca di x in L_v, che è $O(1)$, dato che le liste hanno lunghezza 3. La complessità è quindi $O(h)$.

c. Se le liste hanno lunghezza m, il costo associato a ciascun nodo dell'albero della ricorsione diventa $O(m)$, e quindi la complessità diventa $O(m \cdot h)$.

□

Esercizio 3 Siano $S_1 = S_1[0]S_1[1] \ldots S_1[n-1]$ e $S_2 = S_2[0]S_2[1] \ldots S_2[n-1]$ due sequenze ordinate, ciascuna contenente n interi distinti, che rappresentano due insiemi. Progettare in pseudocodice un algoritmo `Union` che determini il numero di interi appartenenti all'unione $S_1 \cup S_2$. L'algoritmo deve avere complessità $O(n)$ e deve usare spazio costante oltre a quello occupato dall'input.

Soluzione. L'algoritmo è basato su un semplice adattamento della strategia di `Merge`.

Algoritmo `Union`(S_1, S_2)

Input: `sequenze ordinate` S_1, S_2`, ciascuna con` n `interi distinti`
Output: $|S_1 \cup S_2|$
$i \leftarrow 0$; $j \leftarrow 0$; $C \leftarrow 0$;
while `((`$i < n$`) AND (`$j < n$`))` **do**
  $C \leftarrow C + 1$;
  if `(`$S_1[i] = S_2[j]$`)` **then** $\{i \leftarrow i + 1;\ j \leftarrow j + 1\}$;
  else
    if `(`$S_1[i] < S_2[j]$`)` **then** $\{i \leftarrow i + 1\}$;
    else $\{j \leftarrow j + 1\}$;
return $C + (n - i) + (n - j)$

Oltre alle due sequenze di input, l'algoritmo usa $O(1)$ variabili intere che richiedono quindi spazio costante. La complessità è chiaramente $\Theta(n)$ in quanto ogni iterazione esegue un numero costante di operazioni e incrementa almeno uno dei due indici i e j, i quali possono crescere solo sino a n. La correttezza dell'algoritmo (non richiesta dall'esercizio) si basa sul seguente invariante che vale alla fine di ciascuna iterazione dei while:

- $C = |S_1[0 \div i - 1] \cup S_2[0 \div j - 1]|$.
- $S_1[0 \div i - 1] \cup S_2[0 \div j - 1]$ contengono gli $i + j$ interi più piccoli di $S_1 \cup S_2$, e nessuno di esse compare in anche in $S_1[i \div n - 1] \cup S_2[j \div n - 1]$.

□