Algoritmica — Raccolta Pseudocodici (Laboratorio)

Nicolas Manini, Davide Rucci

26 maggio 2019

Sommario

Questo documento raccoglie vari esami precedenti della parte di laboratorio Algoritmica, risolti in pseudocodice. Consigliamo, dopo aver letto il testo di un esame, di leggere la sua soluzione prima in pseudocodice e poi in codice C; in questo modo si riuscirà a comprendere meglio l'idea utilizzata per risolvere un esame prescindendo dai dettagli implementativi. Ogni esame è identificato da nome e data, ed è corredato da un link alla repository di GitHub verso il file C corrispondente.
Si prega di segnalare eventuali errori a `n.manini@studenti.unipi.it` e/o `d.rucci1@studenti.unipi.it`.

1 K stringhe più frequenti (28/05/2009) http://bit.ly/ALL_K_Stringhe

Input: N stringhe di lunghezza al più 100; il numero K di stringhe da restituire.
Output: Le K stringhe più frequenti, in ordine lessicografico.
Assunzioni: Stringhe di lunghezza al più 100 e di frequenze distinte; esistono almeno K stringhe distinte.

```
K-Stringhe(S, N, K)
   // Ordino le stringhe lessicograficamente
   MergeSort(S, 1, N)
   // Array contenente le frequenze di ogni stringa
   C = []

   i = 2 // Utilizzato per scorrere S
   j = 0 // Lunghezza di C

   while i < N // Scorro le stringhe
      count = 1
      // Per ogni stringa conto le occorrenze consecutive
      while i < N && S[i] = S[j]
         count = count + 1
         i = i + 1

      // Ho contato le occorrenze della stringa corrente
      // Inserisco in coda a C la coppia <stringa, frequenza>
      C.append(<S[i-1], count>)
      j = j + 1

      // Avanzo nell'array S
      i = i + 1

   // Ordino l'array C in base alla seconda componente
   MergeSort(C.frequenze, 1, j)

   // Ordino C[1..K] lesicograficamente per la prima componente
   MergeSort(C.stringhe, 1, K)

   for i = 1 to K
      stampa C[i].stringa
```

2 Punti colorati (inefficiente) (28/05/2009) http://bit.ly/ALL_Punti_Ineff

Input: N punti nella forma $\langle x, y, c\rangle$; M query nella forma $\langle x_1, y_1, x_2, y_2\rangle$.
Output: Per ogni query il numero di colori distinti dei punti nel quadrato di estremi $\langle x_1, y_1\rangle$ e $\langle x_2, y_2\rangle$.
Assunzioni: Le componenti x, y, c sono non negative; per ogni query $x_1 \leq x_2 \wedge y_1 \leq y_2$.

```
Punti-Colorati-Ineff(P, N, Q, M)
      // Scorro le query
      for j = 1 to M
         // Estraggo una query
         ⟨x1, y1, x2, y2⟩ = Q[j]

         // Ordino i punti in base alla componente x
         MergeSort(P.x, 1, N)

         start = 1 // Trovo il primo punto che soddisfa il vincolo inferiore sulla x
         while start < N && P[start].x < x1
            start = start + 1

         end = N // Trovo l'ultimo punto che soddisfa il vincolo superiore sulla x
         while end ≥ start && P[end].x > x2
            end = end - 1

         // Ordino il sottoarray individuato in base alla componente y
         MergeSort(P.y, start, end)

         // Trovo il primo punto che soddisfa il vincolo inferiore sulla y
         while start < N && P[start].y < y1
            start = start + 1

         // Trovo l'ultimo punto che soddisfa il vincolo superiore sulla y
         while end ≥ start && P[end].y > y2
            end = end - 1

         // Ordino il sottoarray individuato in base al colore
         MergeSort(P.c, start, end)

         colori = 0 // Contatore dei colori distinti
         ultimo = -1 // Ultimo colore considerato

         // Scorro i punti
         for i = start to end
            if P[i].c ≠ ultimo
               ultimo = P[i].c
               colori = colori + 1

         stampa colori
```

$$T(n, m) = m(n + n\lg(n) + n_{x_1}^{x_2}\lg(n_{x_1}^{x_2}) + n_{x_1 y_1}^{x_2 y_2}\lg(n_{x_1 y_1}^{x_2 y_2}))$$

dove:

$$n_{x_1}^{x_2} = |\{p \in P|\, x_1 \leq p.x \leq x_2\}|$$
$$n_{x_1 y_1}^{x_2 y_2} = |\{p \in P|\, x_1 \leq p.x \leq x_2 \ \wedge y_1 \leq p.y \leq y_2\}|$$

maggiorando poi $n_{x_1}^{x_2} = n$ e $n_{x_1 y_1}^{x_2 y_2} = n$ osserviamo che al caso pessimo:

$$T(n, m) = m(n + 3n\lg(n)) = \mathcal{O}(mn\lg(n))$$

3 Punti colorati (28/05/2009) http://bit.ly/ALL_Punti_Colorati

Input: N punti nella forma $\langle x, y, c\rangle$; M query nella forma $\langle x_1, y_1, x_2, y_2\rangle$.
Output: Per ogni query il numero di colori distinti dei punti nel quadrato di estremi $\langle x_1, y_1\rangle$ e $\langle x_2, y_2\rangle$.
Assunzioni: Le componenti x, y, c sono non negative; per ogni query $x_1 \leq x_2 \wedge y_1 \leq y_2$.

```
Punti-Colorati(P, N, Q, M)
   // Ordino i punti in base al colore
   MergeSort(P.c, 1, N)

      // Scorro le query
      for j = 1 to M
         // Estraggo una query
         ⟨x1, y1, x2, y2⟩ = Q[j]

         colori = 0 // Contatore dei colori distinti
         ultimo = -1 // Ultimo colore considerato

         // Scorro i punti
         for i = 1 to N
            ⟨x, y, c⟩ = P[i]
            // Appartenenza al rettangolo
            if (x1 ≤ x ≤ x2) && (y1 ≤ y ≤ y2)
               // Controllo se non ho mai incontrato questo colore
               if c ≠ ultimo
                  colori = colori + 1
                  ultimo = c
         stampa colori
```

$$T(n, m) = n \lg(n) + mn$$

4 Punti colorati (extra)

Ricerca binaria GEQ*

```
BinSearch-GEQ(A, sx, dx, e)
   // Se l'array e' vuoto ritorno l'estremo maggiore
   if sx > dx
         return sx

      // Calcolo il punto medio
      cx = (sx + dx) / 2

      // Controllo se ho trovato il valore che cerco
      if A[cx] = e
         // Verifico che sia il primo valore e nell'array
         if cx ≠ 0 && A[cx-1] = e
            return BinSearch-GEQ(A, sx, cx - 1, e)
         else
            return cx

      // Ricorsione classica
      if A[cx] > e
         return BinSearch-GEQ(A, sx, cx - 1, e)
      return BinSearch-GEQ(A, cx + 1, dx, e)
```

*trova il primo indice x tale per cui $A[x] \geq e$.

```
Punti-Colorati(P, N, Q, M)
   // Ordino i punti in base al colore, e a parita' per coordinata x
   MergeSort(<P.c, P.x>, 1, N)

   // Indici a cui il colore dei punti in P cambia
   C = []

   num_colori = 0 // Numero di colori distinti
   ultimo = -1
   for i = 1 to N // Calcolo gli indici a cui cambia il colore
      if P[i].c ≠ ultimo
         C.append(i)
         ultimo = P[i].c
         num_colori = num_colori + 1
   C.append(N+1)

   // Scorro le query
   for j = 1 to M
      ⟨x1, y1, x2, y2⟩ = Q[j] // Estraggo una query
      colori = 0

      for i = 1 to num_colori - 1 // Controllo i sottoarray di colore uniforme
         // Primo indice che soddisfa il vincolo sulla x, nel sottoarray di colore i-esimo
         start = BinSearch-GEQ(P.x, C[i], C[i+1], x1)
         // Scorro i punti che soddisfano il vincolo sulla x
         while start < C[i+1] && P[start].x ≤ x2
            // Verifico il vincolo sulla coordinata y
            if (y1 ≤ P[start].y ≤ y2)
               colori = colori + 1
               // Falsifico la guardia del while
               start = C[i+1]

      stampa colori
```

Assumendo che ci siano c colori distinti e che ogni colore abbia n/c punti.

$$T(n,m) = n\lg(n) + n + mc(\lg\left(\frac{n}{c}\right) + \left(\frac{n}{c}\right)_{x_1}^{x_2})$$

5 K occorrenze (02/11/2016) http://bit.ly/ALL_K_Occorrenze

Input: $A[1..N] \in \mathbb{Z}^N$ array in input; $K \in \mathbb{N}$ frequenza minima.
Output: Gli interi $x \in A$ con almeno K occorrenze, in ordine di apparizione in A.

```
K-Occorrenze(A, N, K)
   // Uso una tabella hash per tener traccia delle coppie (valore, frequenza)
   T = nuova tabella hash

   // Scorro l'array per contare le frequenze
   for i = 1 to N
      // Cerco il valore corrente nella tabella
      value = Hash-Search(T, A[i])

      // Controllo se la chiave era presente
      if value ≠ NIL
         // A[i] ha frequenza 'value' in A[1..i-1]
         Hash-Remove(T, A[i])
         Hash-Insert(T, A[i], value+1)
      else
         // A[i] non compare in A[1..i-1]
         Hash-Insert(T, A[i], 1)

   // Scorro l'array per ottenere i valori almeno K-frequenti in ordine
   for i = 1 to N
      // Cerco il valore corrente nella tabella
      value = Hash-Search(T, A[i])

      // Controllo se la chiave e' presente
      if value ≠ NIL
         // A[i] non compare in A[1..i-1]
         // Controllo se A[i] e' almeno K-frequente
         if value ≥ K
            stampa A[i]
```

$$T(n) = n + n = \Theta(n)$$

Sia $n' = |\{x \in \mathbb{Z} \,|\, x \in A\}|$ il numero degli elementi distinti in A, abbiamo che:

$$S(n) = \Theta(n') = O(n)$$

6 15 Oggetti (06/11/2014) http://bit.ly/ALL_15_Oggetti

Input: $A[1..N]$ array di coppie $\langle o, a \rangle$ di oggetti con rispettivi valori affettivi.
Output: I 15 oggetti distinti in A con valore affettivo maggiore.
Assunzioni: Gli oggetti sono stringhe di lunghezza al più 100, $\mathbb{N} \ni N \leq 30$, tabella di dimensione $2N$.

Tabella hash del caso

```
// Funzione hash per stringhe, assumiamo N visibile globalmente
h(Obj) // Obj stringa
   sum = 0
   // Scorro i caratteri della stringa
   for i = 1 to Obj.length
      // Sommo i valori dei caratteri
      sum = sum + valore_decimale(Obj[i])
   return sum % 2N

Hash-Search(T, o)
   ricerca una coppia ⟨o, a⟩ nella lista T(h(o)), altrimenti NIL

Hash-Insert(T, o, a)
   // Ricerco o nella tabella
   val = Hash-Search(T, o)
   if val = NIL
      inserisce ⟨o, a⟩ in testa alla lista T(h(o))
   else // L'oggetto era gia' presente
      if a > val // Mantengo oggetti distinti, con valore affettivo maggiore
         aggiorna la coppia ⟨o, val⟩ con ⟨o, a⟩ in T
```

```
15-Oggetti(A, N)
   // Uso una tabella hash T:oggetti->valore_affettivo per contenere gli oggetti distinti
   T = nuova tabella hash

   // Inserisco gli oggetti nella tabella
   for i = 1 to N
      ⟨o, a⟩ = A[i]
      Hash-Insert(T, o, a)

   // Al termine del ciclo T contiene una sola copia per gli oggetti ripetuti, con associato il valore
       affettivo massimo tra gli eventuali duplicati

   O = [] // Array che conterra' gli oggetti distinti nella tabella T
   dist = 0 // Numero di oggetti distinti

   // Ricopio gli elementi della tabella in O
   for i = 1 to 2N
      for each ⟨o, val⟩ nella lista T[i]
         O.append(⟨o, val⟩)
         dist = dist + 1

   // Ordino l'array O in base alla componente a (valore affettivo)
   MergeSort(O.a, 1, dist)

   // Stampo i primi 15 oggetti (o meno, in caso i distinti fossero <15)
   for i = 1 to min({15, dist})
      stampa O[i].o
```

Sia $n' = |\{o \mid \exists \langle o, a \rangle \in A\}| = \mathcal{O}(n)$ il numero degli oggetti distinti in A, abbiamo che:

$$T(n) = n + 2n + n' \lg(n') = \mathcal{O}(n \lg(n))$$

$$S(n) = \mathcal{O}(n)$$

7 Coda LRU (09/09/2016)

http://bit.ly/ALL_LRU_Array (versione con array)

http://bit.ly/ALL_LRU_Lista (versione con liste)

Input: $N \in \mathbb{N}$ capacità massima della coda, una serie di coppie operazione-argomento nella forma $\langle op, x\rangle \in \{0, 1, 2\} \times (\mathbb{Z} \cup \{*\})$, dove $op = 0$ indica il comando di terminazione, $op = 1$ indica l'accesso all'elemento x in coda (se non presente ne include l'inserimento) e $op = 2$ indica il comando di stampa della coda.
Output: Il contenuto della coda terminato da un carattere $ per ogni operazione di stampa richiesta.
Note: La coda deve implementare una politica *Last Recently Used*; l'argomento assume un valore non significativo $*$ per le operazioni che non lo richiedono (0 e 2).

```
CodaLRU(N)
   // Istanzio una coda di lunghezza massima N
   Q = nuova coda di lunghezza N // Che struttura dati?
   do
      ⟨op, x⟩ = Leggi nuova operazione

      case based on op
         case 0: // Terminazione
            // Non fa nulla
         case 1: // Richiesta
            Richiedi(Q, x)
            // La richiesta opera nel modo seguente:
            //  Rimuove x (se presente)
            //  Inserisce x in testa
            //  Se la dimensione ha ecceduto N, rimuove in coda
         case 2: // Stampa
            Stampa(Q)

   while op ≠ 0 // Operazione di terminazione ricevuta
```

```
Richiedi(Q, x) // Implementazione con Q lista
   if x ∈ Q // Rimuovo x se presente
      Rimuovi(Q, x)

   // Se la coda e' satura, rimuovo in coda
   if Q.size = N
      RimuoviCoda(Q)

   // Reinserisco x in testa
   InserisciTesta(Q, x)
```

```
Richiedi(Q, x) // Implementazione con Q array
   if (Q.size > 0) && (Q[1] = x)
      return // Se l'elemento richiesto e' gia' in testa non devo far nulla

   tmp1 = Q[1]
   // Inserisco x in testa
   Q[1] = x

   inserito = true // Ho aggiunto un nuovo elemento alla coda?
   for i = 2 to min{Q.size + 1, N}
      // Sposto i valori a destra fino a trovare x o ad uscire dalla coda
      tmp2 = Q[i]
      Q[i] = tmp1

      if tmp2 = k // Ho trovato il valore spostato
         inserito = false
         break

      tmp1 = tmp2

```

```
    // Se ho inserito un nuovo elemento incremento size
    if inserito && (Q.size < N)
       Q.size = Q.size + 1
```

8 Mediana in un ABR (25/07/2016) http://bit.ly/ALL_ABR_Mediana

Input: $n_i \in \mathbb{Z}$, $i = 1 \ldots N$ elementi.
Output: La mediana dei valori.
Note: Gli elementi devono essere inseriti in un ABR.

```
MedianaABR({n_i}, N)
   // Nuovo ABR vuoto
   T = nuovo ABR

   // Inserisco i valori in T
   for i = 1 .. N
      Inserisci(T, n_i)

   // Calcolo la mediana in modo ricorsivo
   // Result e' una variabile in cui ritorniamo l'output
   ⟨m, [out]result⟩ = Mediana(T, 0, N)

   stampa result
```

```
// Effettua una visita simmetrica
// Ritorna il conteggio dei nodi contenenti un numero minore
// Restituisce la mediana assegnando result
Mediana(T, count, N)
   // Visito solo la prima meta' dei nodi
   if (count > N/2) || (count < 0)
      return -1

   if T.sx ≠ ∅ // Conto nel sottoalbero sinistro
      count = Mediana(T.sx, count, N)

   // Controllo se sono sul nodo contenente la mediana
   if count = N/2
      // Ritorno il valore in result
      result ← T.key
      // Ritorno il valore n/2 + 1 per non proseguire oltre
      return count + 1

   // Includo il nodo corrente nel conteggio
   count = count + 1

   if T.dx ≠ ∅ // Conto nel sottoalbero destro
      count = Mediana(T.dx, count, N)

   return count
```

La funzione Mediana opera effettuando una visita simmetrica e termina non appena raggiunge il nodo contenente la mediana, esplorando in ordine crescente di chiave, da cui la complessità è $\Theta(n/2) = \mathcal{O}(n)$.

9 Prefissi in un ABR (25/01/2017) http://bit.ly/ALL_ABR_Prefissi

Input: N stringhe s_i lunghe al più 100 caratteri l'una, da inserire in un ABR.
Output: Le chiavi (ordinate) aventi come minimo nell'albero radicato in esse un proprio prefisso.
Note: Si consideri come ordinamento su stringhe l'ordinamento lessicografico, non si possono memorizzare informazioni aggiuntive nei nodi dell'albero, la stampa dell'output deve essere effettuato tramite una chiamata a una funzione lineare nel numero di nodi, si noti che il minimo in un nodo foglia è il valore in essa.

```
PrefissiABR({s_i}, N)
   // Nuovo ABR vuoto
   T = nuovo ABR

   // Inserisco le stringhe in T
   for i = 1 .. N
      Inserisci(T, s_i)

   // Stampo i risultati
   Prefix(T)
```

Soluzione inefficiente

```
m(T) // Ritorna il minimo dell'albero
   if T = ∅ // Se l'albero e' vuoto ritorno NIL
      return NIL

   if T.sx = ∅ // Se non ho sottoalbero sinistro
      return T.key

   // Ricerco nel sottoalbero sinistro
   return m(T.sx)

Prefix_Inefficiente(T)
   if T ≠ ∅ // Controllo se l'albero e' vuoto
      // Ricorro nell'albero sinistro
      Prefix_Inefficiente(T.sx)

      // Minimo dell'albero
      μ = m(T)

      // Controllo se il minimo trovato e' prefisso della chiave corrente
      if T.key = μ :: σ per qualche stringa σ
         stampa T.key

      // Ricorro nell'albero destro
      Prefix_Inefficiente(T.dx)
```

Complessità della soluzione inefficiente:

Poichè l'albero non è bilanciato:

$$T_m(n) = \mathcal{O}(n)$$

Da cui:

$$T(n) = n * T_m(n) = \mathcal{O}(n^2)$$

Soluzione lineare

```
// Ritorna il minimo nell'albero T, stampando durante la visita i nodi da restituire
Prefix(T)
   // Se l'albero e' vuoto ritorno la stringa vuota
   if T = ∅
      return ϵ // Si noti che la stringa vuota e' prefisso di ogni stringa

   // Minimo del sottoalbero sinistro
   μ = Prefix(T.sx)

   // Controllo se il minimo trovato e' prefisso della chiave corrente
   if T.key = μ :: σ per qualche stringa σ
      stampa T.key

   // Proseguo nel sottoalbero destro
   Prefix(T.dx)

   // Ritorno il minimo di questo albero
   if T.sx = ∅
      return T.key
   return μ
```

Questa soluzione segue la struttura di una visita simmetrica, la complessità è quindi $\mathcal{O}(n)$.

10 Quasi-Massimo (09/06/2016) http://bit.ly/ALL_ABR_Quasimax

Input: $\mathbb{N} \ni N \geq 2$ valori $n_i \in \mathbb{Z}$.
Output: Il penultimo valore nella sequenza ordinata degli n_i, ossia il valore di rango $N-1$.
Note: I valori vanno memorizzati in un ABR non bilanciato e l'algoritmo deve ricercare il Quasi-Massimo in tempo lineare nell'altezza dell'albero.

```
Quasi-Massimo({n_1,...,n_N})
   // Istanzio un albero vuoto
   T = ∅

   // Inserisco i valori in T
   for i = 1 to N
      Inserisci(T, n_i)

   // Stampo il risultato
   stampa QuasiMax(T)
```

```
QuasiMax(T)
   // Ricerco il nodo contenente il massimo di T
   m = Massimo(T)

   // Se m ha un figlio sinistro, il Quasi-Massimo e' il massimo dell'albero radicato in esso
   if m.sx ≠ ∅
      return Massimo(m.sx).key

   // Se m non ha figlo sinistro allora il Quasi-Massimo e' il padre di m
   return (m.padre).key

Massimo(T) // Ritorna il massimo nell'albero radicato in T
   if T = ∅
      return NIL

   if T.dx = ∅ // Se non c'e' un figlio destro ho trovato il massimo
      return T.key

   return Massimo(T.dx)
```

Sia h l'altezza dell'albero. Abbiamo che la complessità della procedura Massimo è:

$$T_{max}(n) = \mathcal{O}(h)$$

Si noti inoltre che seppur la funzione QuasiMax può effettuare due chiamate a Massimo abbiamo che la seconda chiamata viene effettuata su di un nodo a profondità maggiore di quella a cui la prima chiamata si è arrestata, da cui complessivamente le due chiamate effettuano al più una sola percolazione completa dell'albero:

$$T(n) = T^1_{max}(n) + T^2_{max} = \mathcal{O}(h_1) + \mathcal{O}(h_2) = \mathcal{O}(h)$$

dove $h_1 + h_2 \leq h$.

11 Sotto-Massimo (05/04/2016) http://bit.ly/ALL_ABR_Sottomax

Input: $N \in \mathbb{N}$ coppie $\langle k_i, v_i \rangle \in (\mathcal{S} \times \mathcal{V}) \subseteq (\Sigma^{100} \times \mathbb{N}^{+})$, dove $\mathcal{S} = \bigcup\{k_i\}$ e $\mathcal{V} = \bigcup\{v_i\}$ sono rispettivamente i domini delle chiavi (distinte, $|\mathcal{S}| = N$) e dei valori; una chiave $s \in \mathcal{S}$.
Output: Il valore massimo $\max(T_s)$ tra i valori nei nodi del sottoalbero radicato nel nodo di chiave s.
Note: Gli elementi devono essere inseriti in un ABR non bilanciato ordinato rispetto alle chiavi.

```
Sotto-Massimo(⟨k_i, v_i⟩, s)
   // Nuovo ABR vuoto
   T = ∅

   // Inserisco i valori in T
   for i = 1 .. N // Inserimento ordinato rispetto alla prima componente
      Inserisci(T, ⟨k_i, v_i⟩)

   // Ricerco il nodo di chiave s
   T_s = Ricerca(T, s)

   // Stampo il massimo nell'albero individuato
   stampa TrovaMassimo(T_s)
```

```
RicercaKey(T, s)
   if T = ∅
      return NIL

   ⟨k_T, v_T⟩ = T.key

   // Percolo l'albero in base all'ordinamento delle chiavi
   if s > k_T // Ordinamento lessicografico
      return RicercaKey(T.dx, s)
   else if s < k_T
      return RicercaKey(T.sx, s)

   // Se ho trovato s, ritorno il nodo corrente
   return T
```

```
TrovaMassimo(T) // Ricerca esaustiva, non ho la proprieta' di ABR
   if T = ∅
      return -1

   ⟨k_T, v_T⟩ = T.key

   // Trovo i due sotto-massimi
   max_R = TrovaMassimo(T.dx)
   max_L = TrovaMassimo(T.sx)

   // Ritorno il massimo
   return max{max_R, max_L, v_T}
```

La funzione TrovaMassimo esegue una postvisita dell'albero, da cui una complessità lineare nel numero di nodi

$$T_{max}(n) = \Theta(n)$$

La funzione RicercaKey esegue una ricerca percolando l'albero secondo la proprietà di ABR:

$$T_{RK}(n) = \mathcal{O}(n)$$

Complessivamente quindi avremo che, sia $n' < n$ la dimensione del sottoalbero T_s:

$$T(n) = T_{RK}(n) + T_{max}(n') = \mathcal{O}(n) + \Theta(n') = \mathcal{O}(n)$$

12 LCA — Lowest Common Ancestor (04/07/2016) http://bit.ly/ALL_ABR_Lca

Input: $\mathbb{N} \ni N \geq 2$ valori $n_i \in \mathbb{N}^+$; due valori $x, y \in [n_1, \ldots, n_N]$.
Output: Il *lowest common ancestor* dei nodi contenenti x e y più vicini alla radice.
Note: Sia T un albero, $x, y \in T$, si definisce $lca(T)$ la chiave del nodo di profondità maggiore tra quelli aventi negli alberi in essi radicati i due valori x e y; La soluzione deve essere lineare nell'altezza dell'ABR in cui si memorizzano i valori; I valori n_i possono contenere duplicati, in fase di inserimento un duplicato va inserito nel sottoalbero destro rispetto alle chiavi identiche incontrate.

```
Lowest-Common-Ancestor({n_i}, x, y)
   // Nuovo ABR vuoto
   T = ∅

   // Inserisco i valori in T
   for i = 1 .. N
      Inserisci(T, n_i)

   // Stampo il risultato
   stampa lca(T, x, y)
```

```
lca(T, x, y)
   if T = ∅
      return -1

   // Controllo se x e y sono entrambi nel sottoalbero sinistro
   if T.key > max{x,y}
      return lca(T.sx, x, y)

   // Controllo se x e y sono entrambi nel sottoalbero destro
   if T.key < min{x,y}
      return lca(T.dx, x, y)

   // Se sono in sottoalberi distinti o se il valore corrente e' x o y ho finito
   return T.key
```

La funzione lca ricorre in al più un solo sottoalbero, da cui al più esegue una percolazione di un ramo intero, sia h l'altezza dell'albero:

$$T(n) = \mathcal{O}(h)$$