

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.

```
1 K-Stringhe(S, N, K)
2     // Ordino le stringhe lessicograficamente
3     MergeSort(S, 1, N)
4     // Array contenente le frequenze di ogni stringa
5     C = []
6
7     i = 2 // Utilizzato per scorrere S
8     j = 0 // Lunghezza di C
9
10    while i < N // Scorro le stringhe
11        count = 1
12        // Per ogni stringa conto le occorrenze consecutive
13        while i < N && S[i] = S[j]
14            count = count + 1
15            i = i + 1
16
17        // Ho contato le occorrenze della stringa corrente
18        // Inserisco in coda a C la coppia <stringa, frequenza>
19        C.append(<S[i-1], count>)
20        j = j + 1
21
22        // Avanzo nell'array S
23        i = i + 1
24
25    // Ordino l'array C in base alla seconda componente
26    MergeSort(C.frequenze, 1, j)
27
28    // Ordino C[1..K] lessicograficamente per la prima componente
29    MergeSort(C.stringhe, 1, K)
30
31    for i = 1 to K
32        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 $x_1 \leq x_2 \wedge y_1 \leq y_2$.

```

1  Punti-Colorati-Ineff(P, N, Q, M)
2      // Scorro le query
3      for j = 1 to M
4          // Estraggo una query
5           $\langle x_1, y_1, x_2, y_2 \rangle = Q[j]$ 
6
7          // Ordino i punti in base alla componente x
8          MergeSort(P.x, 1, N)
9
10         start = 1 // Trovo il primo punto che soddisfa il vincolo inferiore sulla x
11         while start < N && P[start].x <  $x_1$ 
12             start = start + 1
13
14         end = N // Trovo l'ultimo punto che soddisfa il vincolo superiore sulla x
15         while end  $\geq$  start && P[end].x >  $x_2$ 
16             end = end - 1
17
18         // Ordino il sottoarray individuato in base alla componente y
19         MergeSort(P.y, start, end)
20
21         // Trovo il primo punto che soddisfa il vincolo inferiore sulla y
22         while start < N && P[start].y <  $y_1$ 
23             start = start + 1
24
25         // Trovo l'ultimo punto che soddisfa il vincolo superiore sulla y
26         while end  $\geq$  start && P[end].y >  $y_2$ 
27             end = end - 1
28
29         // Ordino il sottoarray individuato in base al colore
30         MergeSort(P.c, start, end)
31
32         colori = 0 // Contatore dei colori distinti
33         ultimo = -1 // Ultimo colore considerato
34
35         // Scorro i punti
36         for i = start to end
37             if P[i].c  $\neq$  ultimo
38                 ultimo = P[i].c
39                 colori = colori + 1
40
41         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:

$$n_{x_1}^{x_2} = |\{p \in P \mid x_1 \leq p.x \leq x_2\}|$$

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

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 $x_1 \leq x_2 \wedge y_1 \leq y_2$.

```
1 Punti-Colorati(P, N, Q, M)
2   // Ordino i punti in base al colore
3   MergeSort(P.c, 1, N)
4
5   // Scorro le query
6   for j = 1 to M
7     // Estraggo una query
8      $\langle x_1, y_1, x_2, y_2 \rangle = Q[j]$ 
9
10    colori = 0 // Contatore dei colori distinti
11    ultimo = -1 // Ultimo colore considerato
12
13    // Scorro i punti
14    for i = 1 to N
15       $\langle x, y, c \rangle = P[i]$ 
16      // Appartenenza al rettangolo
17      if  $(x_1 \leq x \leq x_2) \ \&\& \ (y_1 \leq y \leq y_2)$ 
18        // Controllo se non ho mai incontrato questo colore
19        if  $c \neq \text{ultimo}$ 
20          colori = colori + 1
21          ultimo = c
22    stampa colori
```

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

4 Punti colorati (extra)

Ricerca binaria GEQ*

```

1 BinSearch-GEQ(A, sx, dx, e)
2   // Se l'array e' vuoto ritorno l'estremo maggiore
3   if sx > dx
4       return sx
5
6   // Calcolo il punto medio
7   cx = (sx + dx) / 2
8
9   // Controllo se ho trovato il valore che cerco
10  if A[cx] = e
11      // Verifico che sia il primo valore e nell'array
12      if cx ≠ 0 && A[cx-1] = e
13          return BinSearch-GEQ(A, sx, cx - 1, e)
14      else
15          return cx
16
17  // Ricorsione classica
18  if A[cx] > e
19      return BinSearch-GEQ(A, sx, cx - 1, e)
20  return BinSearch-GEQ(A, cx + 1, dx, e)

```

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

```

1 Punti-Colorati(P, N, Q, M)
2   // Ordino i punti in base al colore, e a parita' per coordinata x
3   MergeSort(<P.c, P.x>, 1, N)
4
5   // Indici a cui il colore dei punti in P cambia
6   C = []
7
8   num_colori = 0 // Numero di colori distinti
9   ultimo = -1
10  for i = 1 to N // Calcolo gli indici a cui cambia il colore
11      if P[i].c ≠ ultimo
12          C.append(i)
13          ultimo = P[i].c
14          num_colori = num_colori + 1
15  C.append(N+1)
16
17  // Scorro le query
18  for j = 1 to M
19      <x1, y1, x2, y2> = Q[j] // Estraggo una query
20      colori = 0
21
22      for i = 1 to num_colori - 1 // Controllo i sottoarray di colore uniforme
23          // Primo indice che soddisfa il vincolo sulla x, nel sottoarray di colore i-esimo
24          start = BinSearch-GEQ(P.x, C[i], C[i+1], x1)
25          // Scorro i punti che soddisfano il vincolo sulla x
26          while start < C[i+1] && P[start].x ≤ x2
27              // Verifico il vincolo sulla coordinata y
28              if (y1 ≤ P[start].y ≤ y2)
29                  colori = colori + 1
30              // Falsifico la guardia del while
31              start = C[i+1]
32
33  stampa colori

```

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

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

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 .

```
1 K-Occorrenze(A, N, K)
2   // Uso una tabella hash per tener traccia delle coppie (valore, frequenza)
3   T = nuova tabella hash
4
5   // Scorro l'array per contare le frequenze
6   for i = 1 to N
7       // Cerco il valore corrente nella tabella
8       value = Hash-Search(T, A[i])
9
10      // Controllo se la chiave era presente
11      if value  $\neq$  NIL
12          // A[i] ha frequenza 'value' in A[1..i-1]
13          Hash-Remove(T, A[i])
14          Hash-Insert(T, A[i], value+1)
15      else
16          // A[i] non compare in A[1..i-1]
17          Hash-Insert(T, A[i], 1)
18
19      // Scorro l'array per ottenere i valori almeno K-frequenti in ordine
20      for i = 1 to N
21          // Cerco il valore corrente nella tabella
22          value = Hash-Search(T, A[i])
23
24          // Controllo se la chiave e' presente
25          if value  $\neq$  NIL
26              // A[i] non compare in A[1..i-1]
27              // Controllo se A[i] e' almeno K-frequente
28              if value  $\geq$  K
29                  stampa A[i]
```

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

Sia $n' = |\{x \in \mathbb{Z} \mid x \in A\}|$ 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

```

1 // Funzione hash per stringhe, assumiamo N visibile globalmente
2 h(Obj) // Obj stringa
3     sum = 0
4     // Scorro i caratteri della stringa
5     for i = 1 to Obj.length
6         // Sommo i valori dei caratteri
7         sum = sum + valore_decimale(Obj[i])
8     return sum % 2N
9
10 Hash-Search(T, o)
11     ricerca una coppia  $\langle o, a \rangle$  nella lista T(h(o)), altrimenti NIL
12
13 Hash-Insert(T, o, a)
14     // Ricorro o nella tabella
15     val = Hash-Search(T, o)
16     if val = NIL
17         inserisce  $\langle o, a \rangle$  in testa alla lista T(h(o))
18     else // L'oggetto era già presente
19         if a > val // Mantengo oggetti distinti, con valore affettivo maggiore
20             aggiorna la coppia  $\langle o, val \rangle$  con  $\langle o, a \rangle$  in T

```

```

1 15-Oggetti(A, N)
2     // Uso una tabella hash T:oggetti->valore_affettivo per contenere gli oggetti distinti
3     T = nuova tabella hash
4
5     // Inserisco gli oggetti nella tabella
6     for i = 1 to N
7          $\langle o, a \rangle = A[i]$ 
8         Hash-Insert(T, o, a)
9
10    // Al termine del ciclo T contiene una sola copia per gli oggetti ripetuti, con associato il valore
11    // affettivo massimo tra gli eventuali duplicati
12
13    O = [] // Array che conterra' gli oggetti distinti nella tabella T
14    dist = 0 // Numero di oggetti distinti
15
16    // Ricopio gli elementi della tabella in O
17    for i = 1 to 2N
18        for each  $\langle o, val \rangle$  nella lista T[i]
19            O.append( $\langle o, val \rangle$ )
20            dist = dist + 1
21
22    // Ordino l'array O in base alla componente a (valore affettivo)
23    MergeSort(O.a, 1, dist)
24
25    // Stampo i primi 15 oggetti (o meno, in caso i distinti fossero <15)
26    for i = 1 to min({15, dist})
27        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).

```
1 CodaLRU(N)
2   // Istanzio una coda di lunghezza massima N
3   Q = nuova coda di lunghezza N // Che struttura dati?
4   do
5      $\langle op, x \rangle$  = Leggi nuova operazione
6
7     case based on op
8       case 0: // Terminazione
9         // Non fa nulla
10      case 1: // Richiesta
11        Richiedi(Q, x)
12        // La richiesta opera nel modo seguente:
13        // Rimuove x (se presente)
14        // Inserisce x in testa
15        // Se la dimensione ha ecceduto N, rimuove in coda
16      case 2: // Stampa
17        Stampa(Q)
18
19  while op  $\neq$  0 // Operazione di terminazione ricevuta
```

```
1 Richiedi(Q, x) // Implementazione con Q lista
2   if  $x \in Q$  // Rimuovo x se presente
3     Rimuovi(Q, x)
4
5   // Se la coda e' satura, rimuovo in coda
6   if Q.size = N
7     RimuoviCoda(Q)
8
9   // Reinserisco x in testa
10  InserisciTesta(Q, x)
```

```
1 Richiedi(Q, x) // Implementazione con Q array
2   if (Q.size > 0) && (Q[1] = x)
3     return // Se l'elemento richiesto e' gia' in testa non devo far nulla
4
5   tmp1 = Q[1]
6   // Inserisco x in testa
7   Q[1] = x
8
9   inserito = true // Ho aggiunto un nuovo elemento alla coda?
10  for i = 2 to min{Q.size + 1, N}
11    // Sposto i valori a destra fino a trovare x o ad uscire dalla coda
12    tmp2 = Q[i]
13    Q[i] = tmp1
14
15    if tmp2 = x // Ho trovato il valore spostato
16      inserito = false
17      break
18
19  tmp1 = tmp2
20
```

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

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

Input: $n_i \in \mathbb{Z}$, $i = 1 \dots N$ elementi.

Output: La mediana dei valori.

Note: Gli elementi devono essere inseriti in un ABR.

```
1 MedianaABR({ $n_i$ }, N)
2   // Nuovo ABR vuoto
3   T = nuovo ABR
4
5   // Inserisco i valori in T
6   for i = 1 .. N
7       Inserisci(T,  $n_i$ )
8
9   // Calcolo la mediana in modo ricorsivo
10  // Result e' una variabile in cui ritorniamo l'output
11   $\langle m, [out]result \rangle = \text{Mediana}(T, 0, N)$ 
12
13  stampa result
```

```
1 // Effettua una visita simmetrica
2 // Ritorna il conteggio dei nodi contenenti un numero minore
3 // Restituisce la mediana assegnando result
4 Mediana(T, count, N)
5   // Visito solo la prima meta' dei nodi
6   if (count > N/2) || (count < 0)
7       return -1
8
9   if T.sx  $\neq \emptyset$  // Conto nel sottoalbero sinistro
10      count = Mediana(T.sx, count, N)
11
12  // Controllo se sono sul nodo contenente la mediana
13  if count = N/2
14      // Ritorno il valore in result
15      result  $\leftarrow$  T.key
16      // Ritorno il valore n/2 + 1 per non proseguire oltre
17      return count + 1
18
19  // Includo il nodo corrente nel conteggio
20  count = count + 1
21
22  if T.dx  $\neq \emptyset$  // Conto nel sottoalbero destro
23      count = Mediana(T.dx, count, N)
24
25  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.

```
1 PrefissiABR({si}, N)
2   // Nuovo ABR vuoto
3   T = nuovo ABR
4
5   // Inserisco le stringhe in T
6   for i = 1 .. N
7     Inserisci(T, si)
8
9   // Stampo i risultati
10  Prefix(T)
```

Soluzione inefficiente

```
1 m(T) // Ritorna il minimo dell'albero
2   if T = ∅ // Se l'albero e' vuoto ritorno NIL
3     return NIL
4
5   if T.sx = ∅ // Se non ho sottoalbero sinistro
6     return T.key
7
8   // Ricorco nel sottoalbero sinistro
9   return m(T.sx)
10
11 Prefix_Inefficiente(T)
12   if T ≠ ∅ // Controllo se l'albero e' vuoto
13     // Ricorro nell'albero sinistro
14     Prefix_Inefficiente(T.sx)
15
16   // Minimo dell'albero
17   μ = m(T)
18
19   // Controllo se il minimo trovato e' prefisso della chiave corrente
20   if T.key = μ :: σ per qualche stringa σ
21     stampa T.key
22
23   // Ricorro nell'albero destro
24   Prefix_Inefficiente(T.dx)
```

Complessità della soluzione inefficiente:

Poiché l'albero non è bilanciato:

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

Da cui:

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

```

1 // Ritorna il minimo nell'albero T, stampando durante la visita i nodi da restituire
2 Prefix(T)
3   // Se l'albero e' vuoto ritorno la stringa vuota
4   if T =  $\emptyset$ 
5       return  $\epsilon$  // Si noti che la stringa vuota e' prefisso di ogni stringa
6
7   // Minimo del sottoalbero sinistro
8    $\mu$  = Prefix(T.sx)
9
10  // Controllo se il minimo trovato e' prefisso della chiave corrente
11  if T.key =  $\mu :: \sigma$  per qualche stringa  $\sigma$ 
12      stampa T.key
13
14  // Proseguo nel sottoalbero destro
15  Prefix(T.dx)
16
17  // Ritorno il minimo di questo albero
18  if T.sx =  $\emptyset$ 
19      return T.key
20  return  $\mu$ 

```

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.

```

1 Quasi-Massimo( $\{n_1, \dots, n_N\}$ )
2   // Istanzio un albero vuoto
3   T =  $\emptyset$ 
4
5   // Inserisco i valori in T
6   for i = 1 to N
7       Inserisci(T,  $n_i$ )
8
9   // Stampo il risultato
10  stampa QuasiMax(T)

```

```

1 QuasiMax(T)
2   // Ricorro il nodo contenente il massimo di T
3   m = Massimo(T)
4
5   // Se m ha un figlio sinistro, il Quasi-Massimo e' il massimo dell'albero radicato in esso
6   if m.sx  $\neq \emptyset$ 
7       return Massimo(m.sx).key
8
9   // Se m non ha figlio sinistro allora il Quasi-Massimo e' il padre di m
10  return (m.padre).key

```

```

1 Massimo(T) // Ritorna il massimo nell'albero radicato in T
2   if T =  $\emptyset$ 
3       return NIL
4
5   if T.dx =  $\emptyset$  // Se non c'e' un figlio destro ho trovato il massimo
6       return T.key
7
8   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:

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

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.

```

1 Sotto-Massimo( $\langle k_i, v_i \rangle$ , s)
2     // Nuovo ABR vuoto
3     T =  $\emptyset$ 
4
5     // Inserisco i valori in T
6     for i = 1 .. N // Inserimento ordinato rispetto alla prima componente
7         Inserisci(T,  $\langle k_i, v_i \rangle$ )
8
9     // Ricerca il nodo di chiave s
10    Ts = Ricerca(T, s)
11
12    // Stampo il massimo nell'albero individuato
13    stampa TrovaMassimo(Ts)

```

```

1 RicercaKey(T, s)
2     if T =  $\emptyset$ 
3         return NIL
4
5      $\langle k_T, v_T \rangle$  = T.key
6
7     // Percolo l'albero in base all'ordinamento delle chiavi
8     if s > kT // Ordinamento lessicografico
9         return RicercaKey(T.dx, s)
10    else if s < kT
11        return RicercaKey(T.sx, s)
12
13    // Se ho trovato s, ritorno il nodo corrente
14    return T

```

```

1 TrovaMassimo(T) // Ricerca esaustiva, non ho la proprieta' di ABR
2     if T =  $\emptyset$ 
3         return -1
4
5      $\langle k_T, v_T \rangle$  = T.key
6
7     // Trovo i due sotto-massimi
8     maxR = TrovaMassimo(T.dx)
9     maxL = TrovaMassimo(T.sx)
10
11    // Ritorno il massimo
12    return max{maxR, maxL, vT}

```

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 $x, y \in [n_1, \dots, n_N]$.

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.

```
1 Lowest-Common-Ancestor({n_i}, x, y)
2   // Nuovo ABR vuoto
3   T = {}
4
5   // Inserisco i valori in T
6   for i = 1 .. N
7       Inserisci(T, n_i)
8
9   // Stampo il risultato
10  stampa lca(T, x, y)
```

```
1 lca(T, x, y)
2   if T = {}
3       return -1
4
5   // Controllo se x e y sono entrambi nel sottoalbero sinistro
6   if T.key > max{x, y}
7       return lca(T.sx, x, y)
8
9   // Controllo se x e y sono entrambi nel sottoalbero destro
10  if T.key < min{x, y}
11      return lca(T.dx, x, y)
12
13  // Se sono in sottoalberi distinti o se il valore corrente e' x o y ho finito
14  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:

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