ESERCIZIO 1 (Immagine 2) - BST con Conteggio Foglie

Soluzione

Struttura Nodo: Ogni nodo deve contenere:

- `key` : chiave del nodo
- `left` : puntatore al figlio sinistro
- `right` : puntatore al figlio destro
- `leaves` : numero di foglie nel sottoalbero radicato in questo nodo

Pseudocodice LEAVES(x):

```
LEAVES(x)
1.  if x = NIL
2.      return 0
3.  return x.leaves
```

Pseudocodice INSERT(T, z):

```
INSERT(T, z)
1.  z.left = NIL
2.  z.right = NIL
3.  z.leaves = 1          // nuovo nodo è foglia
4.  y = NIL
5.  x = T.root
6.  while x ≠ NIL
7.      y = x
8.      if z.key < x.key
9.          x = x.left
10.     else
11.         x = x.right
12. z.p = y
13. if y = NIL
14.     T.root = z
15. else if z.key < y.key
16.     y.left = z
17. else
18.     y.right = z
19. UPDATE-LEAVES-PATH(T, z)
```

Pseudocodice UPDATE-LEAVES-PATH(T, z):

```
UPDATE-LEAVES-PATH(T, z)
1.  x = z.p
2.  while x ≠ NIL
3.      was_leaf = (x.left = NIL and x.right = NIL prima dell'inserimento)
4.      left_leaves = LEAVES(x.left)
5.      right_leaves = LEAVES(x.right)
6.      x.leaves = left_leaves + right_leaves
7.      if was_leaf and x.leaves = 1
8.          x.leaves = 0    // x non è più foglia
9.      x = x.p
```

Nota: il calcolo del campo `was_leaf` richiede attenzione. In realtà, dopo l'inserimento, se `x` aveva zero figli e ora ne ha uno, deve aggiornare il conteggio sottraendo 1 (se stesso come foglia) e aggiungendo le foglie del sottoalbero.

Versione corretta UPDATE-LEAVES-PATH:

```
UPDATE-LEAVES-PATH(T, z)
1.  x = z.p
2.  while x ≠ NIL
3.      old_leaves = x.leaves
4.      left_leaves = 0
5.      right_leaves = 0
6.      if x.left ≠ NIL
7.          left_leaves = x.left.leaves
8.      if x.right ≠ NIL
9.          right_leaves = x.right.leaves
10.     if x.left = NIL and x.right = NIL
11.         x.leaves = 1    // x è foglia
12.     else
13.         x.leaves = left_leaves + right_leaves
14.     x = x.p
```

Complessità:

- **LEAVES(x):** O(1)
- **INSERT(T, z):** O(h) dove h è l'altezza dell'albero (inserimento BST standard + aggiornamento path dalla foglia alla radice)

ESERCIZIO 1 (Immagine 3) - SortJoin con Max-Heap

Soluzione

Pseudocodice:

```
SORTJOIN(A, B, n)
1.  i = 1
2.  j = 1
3.  k = 1
4.  while i ≤ 2n and j ≤ n
5.      if HEAP-MAXIMUM(A) ≥ HEAP-MAXIMUM(B)
6.          A[k] = HEAP-EXTRACT-MAX(A)
7.          i = i + 1
8.      else
9.          A[k] = HEAP-EXTRACT-MAX(B)
10.         j = j + 1
11.     k = k + 1
12. while i ≤ 2n
13.     A[k] = HEAP-EXTRACT-MAX(A)
14.     i = i + 1
15.     k = k + 1
16. while j ≤ n
17.     A[k] = HEAP-EXTRACT-MAX(B)
18.     j = j + 1
19.     k = k + 1
```

Correttezza: L'algoritmo estrae ripetutamente il massimo tra i due max-heap e lo inserisce in `A`. Poiché estraiamo sempre il massimo globale tra i due heap, costruiamo l'array ordinato in modo decrescente. Alla fine, gli elementi rimanenti nell'heap non vuoto vengono estratti e aggiunti.

Invariante del ciclo principale (righe 4-11): All'inizio dell'iterazione t-esima:

- `A[1..k-1]` contiene i k-1 elementi più grandi estratti dai due heap in ordine decrescente
- `i` conta gli elementi estratti da A
- `j` conta gli elementi estratti da B

Complessità:

- Ogni `HEAP-EXTRACT-MAX` costa O(log n)
- Totale: 2n + n = 3n estrazioni
- **Tempo totale:** O(n log n)
- **Spazio:** O(1) (modifiche in place, B può essere modificato)

ESERCIZIO 1 (Immagine 4) - BST: max, merge

Soluzione

Pseudocodice MAX(T, n):

```
MAX(T, n)
1.  if T.root = NIL
2.      return un array vuoto
3.  A = nuovo array di dimensione n
4.  MAX-RECURSIVE(T.root, A, 0)
5.  return A
```

```
MAX-RECURSIVE(x, A, index)
1.  if x = NIL
2.      return index
3.  index = MAX-RECURSIVE(x.left, A, index)
4.  A[index] = x.key
5.  index = index + 1
6.  index = MAX-RECURSIVE(x.right, A, index)
7.  return index
```

Questo esegue una visita in-order del BST, riempiendo l'array con le chiavi in ordine crescente. Poiché il BST è completo con n nodi, l'array risultante contiene tutte le chiavi ordinate.

Pseudocodice MERGE(T1, T2, k):

```
MERGE(T1, T2, k)
1.  A1 = MAX(T1, n)           // array ordinato da T1
2.  A2 = MAX(T2, n)           // array ordinato da T2
3.  A = nuovo array di dimensione 2n+1
4.  i = 1, j = 1, idx = 1
5.  // Merge dei due array ordinati
6.  while i ≤ n and j ≤ n
7.      if A1[i] < A2[j]
8.          A[idx] = A1[i]
9.          i = i + 1
10.     else if A1[i] > A2[j]
11.         A[idx] = A2[j]
12.         j = j + 1
13.     else  // A1[i] = A2[j], aggiungiamo solo uno
14.         A[idx] = A1[i]
15.         i = i + 1
16.         j = j + 1
17.     idx = idx + 1
18. // Copia elementi rimanenti
19. while i ≤ n
20.     A[idx] = A1[i]
21.     i = i + 1
22.     idx = idx + 1
23. while j ≤ n
24.     A[idx] = A2[j]
```

```
25.     j = j + 1
26.     idx = idx + 1
27. // Aggiungi k
28. A[idx] = k
29. idx = idx + 1
30. // Ordina l'array finale (k potrebbe essere in qualsiasi posizione)
31. INSERTION-SORT(A, idx-1)
32. return A
```

Nota: per efficienza, invece di ordinare alla fine, possiamo inserire k durante il merge nella posizione corretta.

Versione ottimizzata:

```
MERGE(T1, T2, k)
1.  A1 = MAX(T1, n)
2.  A2 = MAX(T2, n)
3.  A = nuovo array di dimensione 2n+1
4.  i = 1, j = 1, idx = 1
5.  k_inserted = false
6.  while i ≤ n and j ≤ n
7.      // Inserisci k se è il prossimo in ordine
8.      if not k_inserted and k < A1[i] and k < A2[j]
9.          A[idx] = k
10.         k_inserted = true
11.         idx = idx + 1
12.     if A1[i] < A2[j]
13.         if not k_inserted and k < A1[i]
14.             A[idx] = k
15.             k_inserted = true
16.             idx = idx + 1
17.         if A1[i] < A2[j]
18.             A[idx] = A1[i]
19.             i = i + 1
20.         else
21.             // gestisci duplicati e inserimento k...
```

Questa versione diventa complessa. La versione precedente con sort finale è più chiara e comunque O(n).

Complessità:

- **MAX(T, n):** O(n) (visita in-order)
- **MERGE:** O(n) per ogni MAX, O(n) per merge, O(n) per insertion sort finale
- **Totale:** O(n)

ESERCIZIO 2 (Immagine 5) - Selezione Attività (Greedy)

Soluzione

(a) Algoritmo Greedy:

```
ACTIVITY-SELECTOR-LAST(s, f, n)
1.  A = {an}                    // inizia con l'ultima attività
2.  k = n
3.  for i = n-1 downto 1
4.      if f[i] ≤ s[k]         // attività i compatibile con k
5.          A = A ∪ {ai}
6.          k = i
7.  return A
```

(b) Esecuzione su input dato:

- s = (1, 2, 3, 6, 7, 9)
- f = (4, 5, 7, 8, 12, 11)

Ordinato per tempo di fine:

- a1: [1, 4]
- a2: [2, 5]
- a3: [3, 7]
- a4: [6, 8]
- a5: [7, 12]
- a6: [9, 11]

Partendo da a6 (ultima): [9, 11]

- a5: f[5] = 12 > s[6] = 9, NON compatibile
- a4: f[4] = 8 ≤ s[6] = 9, compatibile → aggiungi a4: [6, 8]
- a3: f[3] = 7 ≤ s[4] = 6, NON compatibile
- a2: f[2] = 5 ≤ s[4] = 6, compatibile → aggiungi a2: [2, 5]
- a1: f[1] = 4 ≤ s[2] = 2, NON compatibile

Risultato: {a6, a4, a2} = {[9,11], [6,8], [2,5]}

(c) Dimostrazione proprietà di scelta greedy:

Teorema: Esiste una soluzione ottima che contiene l'attività che inizia per ultima (an).

Dimostrazione: Sia O una soluzione ottima che non contiene an. Sia aj l'attività in O con tempo di inizio massimo. Poiché le attività sono ordinate per tempo di inizio, aj inizia prima o

al tempo di an, quindi s[j] ≤ s[n].

Consideriamo O' = (O \ {aj}) ∪ {an}.

Dobbiamo dimostrare che O' è una soluzione valida:

- an è compatibile con tutte le attività in O \ {aj} perché an inizia dopo tutte loro (s[n] ≥ s[j] ≥ s[i] per ogni ai in O \ {aj})
- Inoltre, se aj finiva prima che iniziasse un'altra attività ak in O (f[j] ≤ s[k]), allora anche an finisce prima che inizi ak se f[n] ≤ s[k]

In realtà questa dimostrazione non è completa. La scelta greedy "ultima attività" non è corretta in generale.

Correzione: La scelta greedy standard è selezionare l'attività che **finisce per prima**, non quella che inizia per ultima.

Se l'esercizio richiede specificamente "inizia per ultima", allora:

Controesempio:

- a1: [1, 10]
- a2: [11, 12]
- a3: [5, 6]

Inizia per ultima: a2 [11, 12]

- a1 non compatibile (finisce a 10, a2 inizia a 11, ma a1 inizia prima di a2)
- Soluzione: {a2, a3} se a3 è compatibile

Soluzione ottima: {a1} oppure {a2, a3} = 2 attività

La strategia "inizia per ultima" non garantisce ottimalità.

Interpretazione corretta: L'esercizio probabilmente intende la scelta greedy standard (finisce per prima), applicata in ordine inverso.

ESERCIZIO 2 (Immagine 6) - Longest Common Substring (LCS)

Soluzione

(a) Complessità esaustiva: Una sottostringa di X di lunghezza n ha O(n²) sottostringhe possibili (i possibili punti di inizio × lunghezze). Per ogni sottostringa di X, verificare se è sottostringa di Y richiede O(n) tempo.

Complessità totale: $O(n^3)$

(b) Modifica con substring checking in O(m+n): Se possiamo verificare in O(m+n) se una stringa di lunghezza m è sottostringa di una di lunghezza n (es. algoritmo KMP), possiamo:

- Generare tutte le $O(n^2)$ sottostringhe di X
- Per ognuna, verificare in O(n) se è sottostringa di Y
- **Complessità:** $O(n^3)$ → nessun miglioramento asintoto, ma costanti migliori

(c) Programmazione Dinamica:

Relazione di ricorrenza: Sia `L[i][j]` la lunghezza della LCS che termina in `X[i]` e `Y[j]`.

```
L[i][j] = { 0                    se i=0 o j=0
          { L[i-1][j-1] + 1      se X[i] = Y[j]
          { 0                    se X[i] ≠ Y[j]
```

Algoritmo bottom-up:

```
LCS-LENGTH(X, Y, n)
1.  L = matrice (n+1) × (n+1)
2.  max_length = 0
3.  for i = 0 to n
4.      L[i][0] = 0
5.  for j = 0 to n
6.      L[0][j] = 0
7.  for i = 1 to n
8.      for j = 1 to n
9.          if X[i] = Y[j]
10.             L[i][j] = L[i-1][j-1] + 1
11.             if L[i][j] > max_length
12.                 max_length = L[i][j]
13.         else
14.             L[i][j] = 0
15. return max_length
```

Complessità:

- **Tempo:** $O(n^2)$
- **Spazio:** $O(n^2)$

Esercizio 1 (10 punti) Diciamo che un array senza ripetizioni $A[1,n]$ è *semi-ordinato* se esiste un indice k, con $1 \leq k < n$, tale che $A[k+1..n]A[1..k]$ sia ordinato, ovvero i sottoarray $A[k+1..n]$ e $A[1..k]$ sono ordinati e $A[n] < A[1]$. In questo caso l'indice k viene detto il centro dell'array. Ad esempio l'array che segue è semi-ordinato con centro $k = 4$.

1	2	3	4	5	6	7
4	9	12	18	-1	1	2

Scrivere una funzione `centre(A)` che dato un array A semi-ordinato ne restituisce il centro. Giustificare la correttezza dell'algoritmo e valutarne la complessità.

Soluzione: L'idea è quella di procedere con un algoritmo divide et impera. A tal fine osservazione fondamentale è la seguente: dato un sottoarry $A[p..r]$ semi-ordinato, se lo divido in due sottoarray $A[p..q]$ e $A[p..r]$, allora uno solo dei due è semi-ordinato (quello che contiene il centro), mentre l'altro è ordinato. Quando la dimensione del sottoarray $A[p..r]$ diventa 2, il centro è p.

Lo pseudo-codice è quindi il seguente:

```
centre(A)
    return centre-rec(A,1,A.length)

centre-rec(A,p,r)
    if r == p+1
        return p
    else
        q = (p+r)//2
        if (A[q]<A[p])
            return centre-rec(A,p,q)
        else
            return centre-rec(A,q,r)
```

La prova di correttezza procede per induzione sul numero di elementi n dell'array $A[p..r]$

- Se $n = 2$, ovvero $r = p + 1$, allora $A[p+1] < A[p]$ e chiaramente p è il centro.
- Se invece $n > 2$, l'array viene diviso in due parti, ovvero si definisce $q = \lfloor (p+r)/2 \rfloor$ e si considerano gli array $A[p..q]$ e $A[q..r]$. Detto k il centro, ci sono due possibilità.
  - $(q \leq k)$ Questo accade se e solo se $A[p] > A[q]$, e correttamente in questo caso si ricorre sul sottoarray $A[q..r]$
  - $(q > k)$ Questo accade se e solo se $A[p] < A[q]$, e correttamente in questo caso si ricorre sul sottoarray $A[p..q]$

Esercizio 1 (10 punti) Un array di interi `A[1..n]` si dice *triangolare* se esiste $q \in [1,n]$ tale che `A[1..q]` è ordinato in modo crescente e `A[q..n]` è ordinato in modo decrescente. Realizzare una funzione `maxTr(A)` che dato un array triangolare `A[1..n]`, senza elementi ripetuti, ne trova il massimo. Valutarne la complessità.

Soluzione: Si realizza una procedura divide et impera che opera sul sotto-array $A[p, r]$, che inizialmente sarà l'intero array. Se $p = r$, ovvero il sotto-array ha dimensione 1, semplicemente si ritorna l'unico elemento $A[p]$. Altrimenti, si considera il punto intermedio $q = \lfloor p + r \rfloor$. Se $A[q] < A[q + 1]$ significa che la parte `A[p..q]` è crescente e la parte `A[q+1..r]` è ancora triangolare, per cui il massimo si troverà in quest'ultima e può essere cercato ricorsivamente. Dualmente se $A[q] > A[q + 1]$ significa che la parte `A[q+1..r]` è decrescente e la parte `A[p..q]` è ancora triangolare, per cui il massimo si troverà in quest'ultima.

12348675

```
maxTr(A,p,r)
   if p=r
      return A[p]
   else
      q = (p+r)/2
      if A[q]<A[q+1]
         return maxTr(A,q+1,r)
      else
         return maxTr(A,p,q)


maxTr(A)
   return maxTr(A,1,A.length)
```

La complessità è espressa dalla ricorrenza

$$T(n) = T(n/2) + c$$

Si può risolvere utilizzando il master theorem, confrontando $n^{\log_b^a} = n^{\log_2^2} = n$ con $f(n) = c$. Dato che $f(n) = O(n^{\log_b^a - \epsilon}) = O(n^{1-\epsilon})$ per $0 < \epsilon < 1$ si conclude che siamo nel primo caso del teorema che ci da' la soluzione $T(n) = \Theta(\log n)$.

Esercizio 2 (9 punti) Data una stringa di numeri interi $A = (a_1, a_2, \ldots, a_n)$, si consideri la seguente ricorrenza $z(i, j)$ definita per ogni coppia di valori (i, j) con $1 \leq i, j \leq n$:

$$z(i,j) = \begin{cases} a_j & \text{if } i = 1, 1 \leq j \leq n, \\ a_{n+1-i} & \text{if } j = n, 1 < i \leq n, \\ z(i-1,j) \cdot z(i,j+1) \cdot z(i-1,j+1) & \text{altrimenti.} \end{cases}$$

1. Si fornisca il codice di un algoritmo iterativo bottom-up Z(A) che, data in input la stringa A restituisca in uscita il valore $z(n, 1)$.
2. Si valuti il numero esatto $T_Z(n)$ di moltiplicazioni tra interi eseguite dall'algoritmo sviluppato al punto (1).

Soluzione:

1. Date le dipendenze tra gli indici nella ricorrenza, un modo corretto di riempire la tabella è attraverso una scansione "reverse column-major", in cui calcoliamo gli elementi della tabella in ordine decrescente di indice di colonna e, all'interno della stessa colonna, in ordine crescente di indice di riga. Il codice è il seguente.

```
Z(A)
n = length(A)
for i=1 to n do
   z[1,i] = a_i
   z[i,n] = a_{n+1-i}
for j=n-1 downto 1 do
   for i=2 to n do
      z[i,j] = z[i-1,j] * z[i,j+1] * z[i-1,j+1]
return z[n,1]
```

   Si osservi che un altro modo corretto di riempire la tabella è attraverso una scansione "reverse diagonal", che scansiona per diagonali parallele alla diagonale principale partendo da quella contenente solo $z[1, n]$.

2. Ogni iterazione del doppio ciclo dell'algoritmo esegue due moltiplicazioni tra interi, e quindi

$$\begin{aligned} T_Z(n) &= \sum_{j=1}^{n-1} \sum_{i=2}^{n} 2 \\ &= \sum_{j=1}^{n-1} 2(n-1) \\ &= 2(n-1)^2. \end{aligned}$$

   Equivalentemente, basta osservare che l'algoritmo esegue due moltiplicazioni per ogni elemento di una tabella $(n-1) \times (n-1)$.

Esercizio 2 (9 punti) Supponiamo di avere un numero illimitato di monete di ciascuno dei seguenti valori: 50, 20, 1. Dato un numero intero positivo n, l'obiettivo è selezionare il più piccolo numero di monete tale che il loro valore totale sia n. Consideriamo l'algoritmo greedy che consiste nel selezionare ripetutamente la moneta di valore più grande possibile.

(a) Fornire un valore di n per cui l'algoritmo greedy *non* restituisce una soluzione ottima.

(b) Supponiamo ora che i valori delle monete siano 10, 5, 1. In questo caso l'algoritmo greedy restituisce sempre una soluzione ottima: dimostrare che ogni insieme ottimo $M^\star$ di monete di valore totale n contiene la scelta greedy.

Soluzione:

(a) Per esempio $n = 60$, perché la soluzione ottima è 3 monete da 20, mentre l'algoritmo greedy restituisce 11 monete (una da 50 e 10 da 1).

(b) Sia $M^\star$ una soluzione ottima. Sia x il valore maggiore tra 10, 5, e 1 che sia non superiore a n. Se $M^\star$ contiene una moneta di valore x, la proprietà è dimostrata. Altrimenti, sia $M \subseteq M^\star$ un insieme di (2 o più) monete di valore totale x (si osservi che tale insieme esiste sempre quando i valori delle monete sono 10, 5, 1); consideriamo $M' = M^\star \setminus M \cup X$, dove X è l'insieme contenente una moneta di valore x. M' è un insieme di monete di valore totale n e di cardinalità inferiore a quella di $M^\star$: assurdo, quindi questo secondo caso non può verificarsi, e quindi $M^\star$ contiene necessariamente una moneta di valore x.