Soluzioni seconda esercitazione ASD

Alessandra Raffaetà

Esercizio 1. Dati due numeri interi x ed y definiamo la distanza tra x ed y come $d(x, y) = |x - y|$. Sia T un albero binario di ricerca le cui chiavi sono numeri interi e avente almeno due nodi. Scrivere una funzione **efficiente** che restituisca la distanza minima fra le chiavi di due nodi di T.

Qual è la complessità della funzione?

Il prototipo della funzione è

`int distanzaminima(Tree t)`

Non si possono usare strutture ausiliarie di dimensione $\Theta(n)$ dove n è il numero dei nodi dell'albero.

Soluzione: Poiché l'albero è un albero binario di ricerca possiamo sfruttare il fatto che visitando l'albero tramite una chiamata a `treemin` e $n-1$ chiamate a `treesucc`, le chiavi dell'albero sono elencate in ordine non decrescente. Perciò calcolando il minimo delle distanze fra le chiavi di due nodi consecutivi visitati in questo ordine troveremo la distanza minima fra le chiavi di due nodi qualsiasi dell'albero.

```
/*pre: u <> NULL */
/*post: restituisce il minimo del sottoalbero radicato nel nodo u */
Node treemin(Node u){
  while (u->left != NULL)
    u = u->left;

  return u;
}

/*pre: u <> NULL */
/*post: restituisce il successore di u nell'ordine stabilito in una visita
    simmetrica dell'albero a cui u appartiene. Se u e' il massimo restituisce
    NULL*/
Node treesucc(Node u){
  Node y;
  if (u->right != NULL)
    return treemin(u->right);

  y = u->p;
  while (y != NULL && u == y->right){
    u = y;
    y = y->p;
  }
  return y;
}
```

```
/*La funzione restituisce la distanza minima fra le chiavi di due nodi in t*/
int distanzaminima(Tree t) {
  int mind, diff;
  Node temp, succ;

  temp = treemin(t->root);
  succ = treesucc(temp);

  mind = succ->key - temp->key;
  while ((temp = treesucc(succ))!= NULL){
    diff = temp->key - succ->key;
    if (mind > diff)
      mind = diff;
    succ = temp;
  }
  return mind;
}
```

Come abbiamo dimostrato a lezione, una visita effettuata con una chiamata a `treemin` e $n-1$ chiamate a `treesucc`, ha costo $\Theta(n)$ dove n è il numero dei nodi dell'albero, in quanto ogni arco dell'albero è attraversato al più due volte. Di conseguenza, la funzione `distanzaminima(t)` ha complessità $\Theta(n)$.

Esercizio 2. Sia dato un albero binario di ricerca T con n nodi e chiavi naturali. Si scriva una funzione **efficiente** che restituisca un albero binario di ricerca T' contenente tutte e sole le chiavi pari di T. La soluzione deve essere in loco. Si analizzi la complessità in termini di tempo della funzione proposta.

Il prototipo della funzione è

`Tree BSTpari(Tree t)`

Soluzione:

```
/* Funzione ausiliaria che rimuove dall'albero radicato in u i nodi contenenti
    chiavi dispari e restiuisce tale albero */
Node BSTpariaux(Node u, Node *min, Node *max){
  Node minsx, maxsx, mindx, maxdx;

  if (u == NULL){
    *min = NULL;
    *max = NULL;
    return NULL;
  }

  u->left = BSTpariaux(u->left, &minsx, &maxsx);
  u->right = BSTpariaux(u->right, &mindx, &maxdx);
  if (u->key % 2 == 0){
    if (minsx != NULL)
      *min = minsx;
    else
      *min = u;
```

```
    if (maxdx != NULL)
      *max = maxdx;
    else
      *max = u;
    return u;
  }
  if (maxsx!= NULL){
    if (maxsx->p != u){
      maxsx->p->right = maxsx->left;
      if (maxsx->left != NULL)
        maxsx->left->p = maxsx->p;
      maxsx->left = u->left;
      u->left->p = maxsx;
    }
    maxsx->p = u->p;
    maxsx->right = u->right;
    if (u->right != NULL)
      u->right->p = maxsx;
    *min = minsx;
    if (maxdx != NULL)
      *max = maxdx;
    else
      *max = maxsx;
    free(u);
    return maxsx;
  }
  if (mindx!= NULL){
    if (mindx->p != u){
      mindx->p->left = mindx->right;
      if (mindx->right != NULL)
        mindx->right->p = mindx->p;
      mindx->right = u->right;
      u->right->p = mindx;
    }
    mindx->p = u->p;
    mindx->left = u->left;
    *max = maxdx;
    *min = mindx;
    free(u);
    return mindx;
  }
  *min = NULL;
  *max = NULL;
  free(u);
  return NULL;
}

Tree BSTpari(Tree t){

  Node min, max;
  t -> root = BSTpariaux(t->root, &min, &max);
  return t;
}
```

La funzione ausiliaria `BSTpariaux` restituisce non solo la radice dell'albero T' ma anche i nodi minimo e massimo dell'albero T'. Supponiamo che l'albero T abbia una radice u contenente una chiave dispari. Tale nodo deve essere rimosso. Al suo posto metteremo il massimo del sottoalbero di sinistra, `maxsx`, che è calcolato dalla chiamata ricorsiva `BSTpariaux(u->left, &minsx, &maxsx)`, se esiste, altrimenti il minimo del sottoalbero di destra, `mindx`, che è calcolato dalla chiamata ricorsiva `BSTpariaux(u->right, &mindx, &maxdx)`. Infatti sia `maxsx` che `mindx`, possono rimpiazzare u perché soddisfano la proprietà degli alberi di ricerca.

La funzione `BSTpari` invoca la funzione `BSTpariaux`, dunque per calcolare la complessità della prima funzione si dovrà trovare la complessità della funzione ausiliaria. La relazione di ricorrenza di `BSTpariaux` è:

$$T(n) = \begin{cases} c & n = 0 \\ T(k) + T(n-k-1) + d & n > 0 \end{cases}$$

dove n è il numero dei nodi dell'albero e k è il numero dei nodi del sottoalbero sinistro della radice.

Come abbiamo dimostrato a lezione utilizzando il metodo di sostituzione

$$T(n) = (c+d)n + c$$

Quindi la complessità di questa funzione è $\Theta(n)$.

Esercizio 3. Un vettore A è detto vettore di intervalli se ogni suo elemento $A[i]$ ha due campi interi $A[i].start$ e $A[i].end$ tali che $A[i].start \leq A[i].end$. Intuitivamente $A[i].start$ e $A[i].end$ rappresentano l'intervallo chiuso di interi $[A[i].start, A[i].end]$. Un intero k è coperto da A se esiste un intervallo di A che contiene k. Formalmente k è coperto da A se esiste un indice i di A tale che $A[i].start \leq k \leq A[i].end$.

Dato un vettore di intervalli A di lunghezza n, si consideri il problema di determinare un nuovo vettore di intervalli A' di lunghezza $n' \leq n$ che copra gli stessi interi di A, e in cui gli intervalli siano disgiunti. Scrivere una funzione di complessità $O(n \log n)$ che calcoli A'.

Il tipo `Intervallo` è così definito:

```
typedef struct {
        int start;
        int end;
} Intervallo;
```

Il prototipo della funzione è

```
Intervallo * copertura(Intervallo v[], ind dimV, int *dimout)
```

La funzione restituisce il vettore A' e in `*dimout` è memorizzata la sua dimensione.

Soluzione:

```
Intervallo * copertura(Intervallo v[], int dimV, int *dimout){
  int i, j, endtemp;
  Intervallo *ris;
```

```
  if (dimV == 0)
    return 0;

  ordina(v, dimV); /*Ordino gli intervalli in ordine non decrescente con
    heapsort o merge sort */

  ris = malloc(sizeof(Intervallo) * dimV);
  i = 0;
  j = 0;
  while (i < dimV){
    ris[j] = v[i];
    endtemp = v[i].end;
    i = i+1;
    while (i < dimV && endtemp >= v[i].start){
      if (v[i].end > endtemp)
        endtemp = v[i].end;
      i++;
    }
    ris[j].end = endtemp;
    j++;
  }

  if (j < dimV)
    ris = realloc(ris, sizeof(Intervallo)*j);
  *dimout = j;
  return ris;
}
```

La procedura `ordina` ordina gli intervalli in base al primo estremo dell'intervallo e se sono uguali li ordina rispetto al secondo estremo (ordinamento lessicografico). In questo modo per verificare se due intervalli consecutivi sono disgiunti è sufficiente verificare se $A[i].end < A[i+1].start$. Sfruttando questa condizione costruiamo una copertura di intervalli disgiunti.

Assumendo che n sia il numero di intervalli in `v` e che il costo di `malloc` e `realloc` siano costanti, la complessità della funzione è data dalla somma di:

- complessità di `ordina`. Poiché abbiamo scelto heapsort o merge sort la complessità è $\Theta(n \log n)$;
- costo del ciclo `while` esterno. Si può osservare che ogni intervallo è esaminato una sola volta e le operazioni che vengono eseguite per costruire il vettore risultato sono costanti. Di conseguenza il costo è $\Theta(n)$.

Questo ci permette di concludere che la complessità è quella richiesta, cioè $O(n \log n)$.

Esercizio 4. Sia $A[0..n-1]$ un array di n numeri interi distinti. Se $i < j$ e $A[i] > A[j]$, allora la coppia (i, j) è detta **inversione** di A.

a) Elencare le inversioni dell'array $\langle 3, 4, 9, 7, 1 \rangle$.

b) Quale array con elementi estratti dall'insieme $\{1, 2, ..., n\}$ ha più inversioni? Quante inversioni ha?

c) Qual è la realazione fra il tempo di esecuzione di insertion sort e il numero di inversioni nell'array di input? Spiegare la vostra risposta.

d) Create un algoritmo che determina il numero di inversioni in una permutazione di n elementi nel tempo $\Theta(n \log n)$ nel caso peggiore. (Suggerimento: modificare il merge sort). Il prototipo della funzione è

```
int inversioni(int v[], int inf, int sup)
```

Soluzione:

a) Le inversioni sono $(0,4),(1,4),(2,3),(2,4),(3,4)$.

b) L'array con elementi appartenenti all'insieme $\{1,2,...,n\}$ con più inversioni è

$$\langle n, n-1, n-2, \ldots, 1\rangle.$$

Per ogni $0 \leq i < j \leq n-1$ c'è un'inversione (i,j). Il numero di tali inversioni è $\binom{n}{2} = n(n-1)/2$.

c) Ogni iterazione del ciclo interno di insertion sort (`while`) corrisponde all'eliminazione di una inversione.

Supponiamo che nell'input iniziale ci sia un'inversione (k,j). Allora $k < j$ e $A[k] > A[j]$. Quando il ciclo esterno dell'insertion sort (`for`) assegna `key = A[j]`, il valore che si trovava inizialmente in $A[k]$ è sempre in un qualche indice a sinistra di $A[j]$, cioè è in $A[i]$, con $0 \leq i < j$, e così l'inversione è diventata (i,j). Una qualche iterazione del ciclo interno (`while`) muove $A[i]$ una posizione a destra. All'uscita dal ciclo interno `key` sarà posto a sinistra dell'elemento in $A[i]$, eliminando così l'inversione. Poiché il ciclo interno sposta solo elementi che sono maggiori di `key`, sposta solo elementi che corrispondono a inversioni.

d)

```
int countinv(int v[], int inf, int med, int sup){
  int * aux;
  int app = 0, primo = inf, secondo = med + 1, count = 0, i;

  aux = (int *) malloc((sup - inf + 1)*sizeof(int));

  while (primo <= med && secondo <= sup){
    if (v[primo] > v[secondo]){
      count += med - primo + 1;
      aux[app] = v[secondo];
      secondo++;
    }
    else {
      aux[app] = v[primo];
      primo++;
    }
    app++;
  }
  if (primo <= med) {
```

```
        for (i = med; i >= primo; i--){
          v[sup] = v[i];
          sup--;
        }
      }
      for (i = 0; i < app; i++)
      v[i + inf] = aux[i];

      free(aux);
      return count;
    }

    int inversioni(int v[], int inf, int sup){
      int med;

      if (inf >= sup)
        return 0;

      med = (inf + sup)/2;

      return inversioni(v, inf, med) + inversioni(v, med + 1, sup) +
countinv(v, inf, med, sup);
    }
```

La relazione di ricorrenza di `inversioni` è:

$$T(n) = \begin{cases} \Theta(1) & n \leq 1 \\ 2T(n/2) + \Theta(n) & n > 1 \end{cases}$$

Applicando il teorema master, possiamo dimostrare che questa ricorrenza ha soluzione $T(n) = \Theta(n \log n)$.