Soluzioni Esercitazione I ASD 2022-23

Alessandra Raffaetà Lorenzo Cazzaro

Esercizio 1. Scrivere una funzione **EFFICIENTE** `blackHeight(u)` che dato in input la radice `u` di un albero binario, i cui nodi `x` hanno, oltre ai campi `key`, `left` e `right`, un campo `col` che può essere `'B'` (per "black") oppure `'R'` (per "red"), verifica se per ogni nodo, il cammino da quel nodo a qualsiasi foglia contiene lo stesso numero di nodi neri (altezza nera del nodo `x`). In caso negativo, restituisce -1, altrimenti restituisce l'altezza nera della radice.

Il prototipo della funzione è:

```
int blackHeight(PNode u)
```

Valutare la complessità della funzione, indicando eventuali relazioni di ricorrenza.
Il tipo PNode è così definito:

```
struct Node {
       int key;
       char col;
       Node* left;
       Node* right;
       Node(int k, char c, Node* sx = nullptr, Node* dx = nullptr)
            :  key{k}, col{c}, left{sx}, right{dx} {}
 };

typedef Node* PNode;
```

Soluzione:

```
/* Ritorna l'altezza nera dell'albero radicato in u se la proprieta' vale per
    ogni nodo dell'albero, altrimenti ritorna -1 */
int blackHeight(PNode u){
  int rissx, risdx, ris;

  if (u == nullptr)
    return 0;

  /*Altezza nera del sottoalbero sinistro e destro*/
  rissx = blackHeight(u->left);
  risdx = blackHeight(u->right);

  if (rissx == -1 || risdx == -1 || (u->left && u->right && rissx != risdx))
    return -1;
```

```
/*N.B.: ritorna -1 nel caso in cui la proprieta' non e' verificata in un
   sottoalbero oppure le altezze nere sono diverse E entrambi i figli
   esistono */

  if (u->left == nullptr)
    ris = risdx;
  else
    ris = rissx;

  if (u->col == 'B')
    ris++;

  return ris;
}
```

Per il calcolo della complessità temporale asintotica nel caso peggiore, la relazione di ricorrenza di `blackHeight` 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 nel caso peggiore la complessità di questa funzione è $\Theta(n)$.

Esercizio 2. Sia `T` un albero generale i cui nodi hanno campi: `key`, `left_child` e `right_sib`. Scrivere una funzione **EFFICIENTE** che dato in input la radice dell'albero `T` e un intero `k` (`k`$\geq$2) verifica se `T` è un albero `k`-completo.

Il prototipo della funzione è:

```
bool k_Completo(PNodeG r, int k)
```

Analizzare la complessità della funzione. Il tipo `PNodeG` è così definito:

```
struct NodeG {
       int key;
       NodeG* left_child;
       NodeG* right_sib;
       NodeG(int k, NodeG* sx = nullptr, NodeG* dx = nullptr)
             :  key{k}, left_child{sx}, right_sib{dx} {}
};

typedef NodeG* PNodeG;
```

Soluzione:

```
/* Ritorna true se l'albero radicato in r e' k-completo, altrimenti false.
h e' utilizzata per restituire l'altezza dell'albero. */
bool k_CompletoAux(PNodeG r, int k, int& h){

  int grado{0}, hf{-1}, temp;
  PNodeG figlio;
  bool ris{true};

  /*Un albero vuoto e' k-completo e la sua altezza e' -1*/
  if (r == nullptr){
    h = -1;
    return true;
  }

  figlio = r->left_child;

  /* Esamina i sottoalberi dei figli fin tanto che i sottoalberi sono k-
     completi ed esistono ancora figli */
  while (ris && figlio){
    grado++; //tiene traccia del grado del padre.
    ris = (grado <= k) && k_CompletoAux(figlio, k, temp);
    if (hf == -1)
      hf = temp;
    else
      ris = ris && (hf == temp);
      /* se le altezze dei sottoalberi radicati nei figli sono diverse, l'
     albero non puo' essere k-completo */

    figlio = figlio->right_sib;
  }

  h = hf + 1;

  return ris && (grado == 0 || grado == k);
}

bool k_Completo(PNodeG r, int k){
  int h;
  return k_CompletoAux(r, k, h);
}
```

Ogni nodo dell'albero è visitato al più una volta, perciò la complessità è $O(n)$ dove n è il numero dei nodi dell'albero.

Esercizio 3. Sia `T` un albero ternario completo i cui nodi sono colorati di bianco (carattere `'w'`) o nero (carattere `'b'`). L'albero è rappresentato tramite un vettore posizionale. Scrivere una funzione **EFFICIENTE** che ritorni il numero dei nodi del cammino più lungo all'interno dell'albero con nodi che presentano tutti lo stesso colore (il cammino può partire da un nodo qualsiasi, non necessariamente dalla radice) e il colore dei nodi del cammino più lungo.

Il prototipo della funzione è:

`int max_l_cammino_monocolore(const vector<char>& tree, char& colore_max_cammino)`

Analizzare la complessità della funzione.

Soluzione:

```
/* Restituisce il numero dei nodi del cammino monocolore piu' lungo e max
    conterra' il colore di tale cammino. Il parametro i contiene la radice del
     sottoalbero in esame, mentre lungcurr contiene il numero dei nodi del
    cammino monocolore che inizia da i. */
int camminoMaxAux(const vector<char>& tree, int i, int& lungcurr, char& max){

    int lmax1, lmax2, lmax3, lungcurr1, lungcurr2, lungcurr3;
    char max1, max2, max3;

    /*L'albero radicato nel nodo di indice i e' vuoto (siamo fuori dal vettore)
    .*/
    if (i >= tree.size()){
        lungcurr = 0;
        return 0;
    }

    /*Il nodo di indice i e' una foglia.*/
    if (i*3 + 1 >= tree.size()){
        lungcurr = 1;
        max = tree[i];
        return 1;
    }

    /* Ho un albero di almeno 4 nodi*/

    lmax1 = camminoMaxAux(tree, i*3 + 1, lungcurr1, max1);
    lmax2 = camminoMaxAux(tree, i*3 + 2, lungcurr2, max2);
    lmax3 = camminoMaxAux(tree, i*3 + 3, lungcurr3, max3);

    lungcurr = 1;
    /* Incremento, se possibile, il numero dei nodi del cammino monocolore
    corrente che parte dal nodo con indice i */
    if (tree[i] == tree[i*3 + 1])
        lungcurr = lungcurr1 + 1;
    if (tree[i] == tree[i*3 + 2] && lungcurr < lungcurr2 + 1)
        lungcurr = lungcurr2 + 1;
    if (tree[i] == tree[i*3 + 3] && lungcurr < lungcurr3 + 1)
        lungcurr = lungcurr3 + 1;

    /* Se il numero dei nodi del cammino monocolore corrente e' il piu' grande
    */
    if (lungcurr >= lmax1 && lungcurr >= lmax2 && lungcurr >= lmax3){
        max = tree[i];
        return lungcurr;
    }
```

```
    /* Altrimenti, aggiorno max e il numero dei nodi del cammino piu' lungo
    utilizzando i valori restituiti dalle chiamate ai figli. */
    if (lmax1 >= lmax2){
        if (lmax1 >= lmax3){
            max = max1;
            return lmax1;
        }
        max = max3;
        return lmax3;
    }
    if (lmax2 >= lmax3){
        max = max2;
        return lmax2;
    }
    max = max3;
    return lmax3;
}

/* La funzione restituisce il numero dei nodi del cammino monocolore piu' lungo
     e lettera_max_cammino conterra' il colore di tale cammino. */
int max_l_cammino_monocolore(const vector<char>& tree, char&
    lettera_max_cammino){
    int lungcurr;
    return camminoMaxAux(tree, 0, lungcurr, lettera_max_cammino);
}
```

La funzione `max_l_cammino_monocolore` chiama la funzione ausiliaria `camminoMaxAux` una sola volta, di conseguenza è sufficiente stabilire la complessità temporale asintotica della funzione `camminoMaxAux` per ottenere anche quella di `max_l_cammino_monocolore`. Per il calcolo della complessità temporale asintotica nel caso peggiore, possiamo sfruttare la condizione per cui l'albero in input è ternario completo. Quindi, la relazione di ricorrenza di `camminoMaxAux` è:

$$T(n) = \begin{cases} c & n \leq 1 \\ 3T(\frac{n}{3}) + d & n > 1 \end{cases}$$

dove n è il numero dei nodi dell'albero.

Possiamo quindi sfruttare il teorema master per risolvere l'equazione di ricorrenza. Dato che $f(n) = d$, $a = 3$ e $b = 3$, si può verificare che si ricade nel primo caso del teorema master. Infatti, fissato $\epsilon = 1$, si ottiene $f(n) = \mathcal{O}(n^{log_3(3)-1})$, dunque $T(n) = \Theta(n)$.

Esercizio 4. `N` barre di marmo di lunghezza $L = [L_0, L_1, ..., L_{N-1}]$ in metri sono posizionate una accanto all'altra. Ci sono `K` macchine tagliatrici, posizionate una accanto all'altra, in grado di trasformare una barra in cubetti di marmo. Ogni macchina elabora barre di marmo a velocità diverse $S = [S_0, S_1, ..., S_{K-1}]$ in metri/sec.

Le macchine tagliatrici funzionano con i seguenti vincoli:

- una barra può essere elaborata da una sola macchina,
- una macchina può elaborare più barre, una dopo l'altra,

– una macchina, dopo aver completato una barra, può elaborare solo la barra adiacente alla sua destra (quindi una macchina può elaborare SOLO sequenze consecutive di barre),

– le macchine possono lavorare in parallelo,

– l'ordine di barre e macchine non può essere modificato.

Scrivere una funzione `minTime` che restituisce il tempo minimo richiesto per elaborare tutte le barre e inserisce nel vettore `ass` l'assegnazione delle barre alle macchine (ovvero `ass[i] = j` quando la barra `i` è assegnata alla macchina `j`).

Il prototipo della funzione è:

```
int minTime(const vector<int>& len, const vector<int>& speed,
                        vector<int>& ass)
```

Analizzare la complessità.

Soluzione:

```
/* Controlla che il tempo guess sia sufficiente per elaborare tutte le barre */
bool feasible(int guess, const vector<int>& len,  const vector<int>& speed){

  int barre, j, tempo, maxlen;

  barre = 0;
  j = 0;
  while (barre < len.size() && j < speed.size()){
    tempo = guess; /* Sono in parallelo le macchine */

    maxlen = tempo * speed[j];
    while (barre < len.size() && len[barre] <= maxlen){
      maxlen -= len[barre];
      barre += 1;
    }
    j++;
  }

  return barre == len.size();
}

/* Assegna le barre alle relative macchine conoscendo il tempo minimo tempomin
    */
void assegnazioneMacchine(int tempomin, const vector<int>& len, const vector<
    int>& speed, vector<int>& ass){

  int barre, j, tempo, maxlen;

  barre = 0;
  j = 0;
  while (barre < len.size() && j < speed.size()){
    tempo = tempomin; /* Sono in parallelo le macchine */
```

```
    maxlen = tempo * speed[j];
    while (barre < len.size() && len[barre] <= maxlen){
      ass[barre]= j;
      maxlen -= len[barre];
      barre += 1;
    }
    j++;
  }
}

/* Funzione che restituisce il tempo minimo richiesto per elaborare le barre e
    riempie ass con le assegnazioni delle macchine alle barre per ottenere il
    tempo minimo. */
int minTime(const vector<int>& len, const vector<int>& speed, vector<int>& ass)
    {

  int l = 0, r = 0, best, med;

  /* Il tempo massimo possibile si ottiene considerando una sola macchina a
    velocita' 1 che deve elaborare tutte le barre. */
  for (int i = 0; i < len.size(); i++)
    r += len[i];

  best = r;

  /* Applico un approccio divide et impera per individuare il tempo minimo. L'
    intervallo temporale all'interno del quale cercare la soluzione viene
    dimezzato progressivamente.*/
  while (l <= r){
    med = (l+r)/2;

    if (feasible(med, len, speed)){
        r = med - 1;
        best = med;
    }
    else
        l = med + 1;
  }

  assegnazioneMacchine(best, len, speed, ass);

  return best;
}
```

Dato *diml* pari alla dimensione del vettore L, ad ogni tentativo di assegnazione tramite l'esecuzione della funzione `feasible` e della procedura `assegnazioneMacchine` dobbiamo cercare di riempire il vettore `ass`, la cui dimensione è *diml*. Quindi, la complessità temporale asintotica di `feasible` e `assegnazioneMacchine` è $\mathcal{O}(diml)$.

L'intervallo di valori all'interno del quale cerchiamo il tempo minimo contiene inizialmente r tempi da 0 a r, in quanto consideriamo solo tempi come numeri interi, dove r è il tempo richiesto nell'ipotetico caso peggiore per elaborare le barre, cioè

$r = \Sigma_{i=0}^{diml-1} L[i]$. L'intervallo viene dimezzato progressivamente tramite un approccio divide-et-impera.

L'equazione di ricorrenza che descrive la complessità temporale asintotica della funzione `minTime`, che richiama `feasible` e `assegnazioneMacchine`, è:

$$T(r, diml) = \begin{cases} \Theta(diml) & r \leq 1 \\ T(\frac{r}{2}, diml) + \Theta(diml) & r > 1 \end{cases}$$

Tramite il metodo iterativo o il metodo di sostituzione, si ottiene la complessità $\mathcal{O}(log_2(r) * diml)$

Esercizio 5. Ai campionati mondiali di maratona il numero di partecipanti è così alto che occorre organizzare la gara in due turni in giorni diversi. Si costruiscono così le due classifiche, una per turno, ciascuna delle quali riporta i partecipanti a quel turno in ordine di arrivo e il tempo corrispondente.

Dopo le gare, **SENZA** unificare le due classifiche, si vuole avere una tecnica efficiente per determinare, dato una posizione `k`, l'atleta che si è piazzato in quella posizione nella classifica generale. ovvero che ha ottenuto il `k`-mo tempo più basso.

Il prototipo della funzione è:

```
int estrai(const vector<int>& classifica1, const vector<int>&
                    classifica2, int k)
```

Restituire -1 se `k` non indica un elemento presente all'interno dell'unione delle sequenze. Analizzare la complessità della funzione.

Soluzione:

Di seguito sono proposte due soluzioni:

Soluzione I

```
/* La funzione ritorna il valore del k-esimo elemento del merge delle due
    classifiche */
int estrai(const vector<int>& classifica1, const vector<int>& classifica2, int
    k) {
    if (k < 1 || k > classifica1.size()+classifica2.size()) //se k e' fuori
    range
        return -1;

    int conta = 0, i = 0, j = 0, ris;
    int dim1 = classifica1.size();
    int dim2 = classifica2.size();
    bool trovato = false;

    /* Scorri le due classifiche considerando l'elemento minore fra gli
    elementi indicizzati delle due classifiche */
    while (i < dim1 && j < dim2 && !trovato) {
        conta++;
        if (classifica1[i] < classifica2[j]) {
```

```
            if (k == conta) {
                trovato = true;
                ris = classifica1[i];
            }
            else
                i++;
        }
        else if (k == conta) {
                trovato = true;
                ris = classifica2[j];
            }
            else
                j++;
    }

    /* Se classifica2 e' stato completamente esaminato */
    while (i < dim1 && !trovato) {
        conta++;
        if (k == conta) {
            trovato = true;
            ris = classifica1[i];
        }
        else
            i++;
    }

    /* Se classifica1 e' stato completamente esaminato */
    while (!trovato){
        conta++;
        if  (k == conta){
            trovato = true;
            ris = classifica2[j];
        }
        else
            j++;
    }

    return ris;
}
```

Definiamo $dim1$ come la dimensione della prima classifica, $dim2$ come la dimensione della seconda classifica. La complessità asintotica è $T(dim1, dim2, k) = \Theta(k)$, dato che vengono visitati in tutto k elementi.

Soluzione II

```
/* La funzione ausiliaria ritorna il valore del k-esimo elemento del merge
    delle due classifiche */
int estrai_aux(const vector<int>& classifica1, int start1, int end1, const
    vector<int>& classifica2, int start2, int end2, int k) {
    /* start1 rappresenta il piu' basso indice della porzione di classifica1
    considerata;
```

```
    end1 rappresenta il piu' alto indice + 1 della porzione di classifica1
    considerata;
    se start1==end1 siamo fuori da classifica1, prendi quindi il k elemento da
    classifica2 */

    if (start1 == end1)
        return classifica2[start2 + k];

    /* start2 rappresenta il piu' basso indice della porzione di classifica2
    considerata;
    end2 rappresenta il piu' alto indice + 1 della porzione di classifica2
    considerata;
    se start2==end2 siamo fuori da classifica2, prendi quindi il k elemento da
    classifica1 */

    if (start2 == end2)
        return classifica1[start1 + k];

    int mid1 = (end1 - start1) / 2;
    int mid2 = (end2 - start2) / 2;

    /* se mid1+mid2 < k, allora non stiamo considerando abbastanza elementi.
    Puoi pero' escludere la meta' inferiore delle porzioni di una delle due
    classifiche considerate, siccome il k-esimo elemento non sara' contenuto
    in quella meta' */

    if (mid1 + mid2 < k) {
        if (classifica1[start1 + mid1] > classifica2[start2 + mid2])
            return estrai_aux(classifica1, start1, end1, classifica2, start2 +
    mid2 + 1, end2, k - mid2 - 1);
        else
            return estrai_aux(classifica1, start1 + mid1 + 1, end1, classifica2
    , start2, end2, k - mid1 - 1);
    }
    /* se mid1+mid2 >= k, allora stiamo considerando abbastanza elementi. Puoi
    pero' escludere la meta' superiore delle porzioni di una delle due
    classifiche considerate, siccome il k-esimo elemento non sara' contenuto
    in quella meta' */
    else if (classifica1[start1 + mid1] > classifica2[start2 + mid2])
            return estrai_aux(classifica1, start1, start1 + mid1, classifica2,
    start2, end2, k);
        else
            return estrai_aux(classifica1, start1, end1, classifica2, start2,
    start2 + mid2, k);
}

/*La funzione ritorna il valore del k-esimo elemento del merge delle due
    classifiche*/
int estrai(const vector<int>& classifica1, const vector<int>& classifica2, int
    k){

    if (k < 1 || k > classifica1.size()+classifica2.size()) //se k e' fuori
    range
```

```
        return -1;

    return estrai_aux(classifica1, 0, classifica1.size(), classifica2, 0,
    classifica2.size(), k-1);
}
```

Definiamo $dim1$ come la dimensione della prima classifica, $dim2$ come la dimensione della seconda classifica. La complessità asintotica è $T(dim1, dim2, k) = \mathcal{O}(log(dim1) + log(dim2))$, dato che dimezziamo progressivamente le dimensioni dei due vettori al fine di individuare il k-esimo elemento.

Se k è costante rispetto a $dim1 + dim2$, la prima soluzione ha una complessità migliore, altrimenti, nel caso generico, dato che k può essere $\Theta(dim1)$ o $\Theta(dim2)$, è la seconda soluzione ad avere una complessità asintotica migliore.