Soluzioni Esercitazione I ASD 2024-25

Lorenzo Cazzaro Riccardo Maso Alessandra Raffaetà

Esercizio 1. Si consideri un albero ternario completo in cui ogni nodo ha i seguenti campi: (i) **key** chiave intera, (ii) **fruitful** valore booleano, (iii) **left** puntatore al figlio sinistro, (iv) **center** puntatore al figlio centrale, (v) **right** puntatore al figlio destro.

1. Si scriva una procedura efficiente in **C++** che assegni **True** al campo **fruitful** del nodo se e solo se la *somma delle chiavi* dei nodi di ciascuno dei *sottoalberi radicati nei figli* è maggiore di una costante **k** fornita in input.

   Il prototipo della procedura è:

   `void set_fruitful(PTNode r, int k)`

2. **Analizzare e motivare in modo chiaro, preciso ed approfondito la complessità della procedura**, indicando eventuali relazioni di ricorrenza e mostrando la loro risoluzione.

Il tipo PTNode è così definito:

```
struct TNode {
       int key;
       bool fruitful;
       TNode* left;
       TNode* center;
       TNode* right;
};

typedef TNode* PTNode;
```

Soluzione:

```
/*Funzione ausiliaria che restituisce la somma delle chiavi dell'albero
    radicato in u*/
int set_fruitfulAux(PTNode u, int k){
    int sumsx, sumdx, sumc;

    if (u == nullptr){
      return 0;
    }

    sumsx = set_fruitfulAux(u->left, k);
    sumdx = set_fruitfulAux(u->right, k);
```

```
    sumc = set_fruitfulAux(u->center, k);

    u->fruitful = (sumsx > k) && (sumdx > k) && (sumc > k);

    return sumsx + sumdx + sumc + u->key;
}

/*Funzione che restituisce se l'albero e' fruitful o meno.*/
void set_fruitful(PTNode r, int k){
    set_fruitfulAux(r, k);
}
```

La procedura `set_fruitful` chiama `set_fruitfulAux` senza eseguire altre operazioni. Di conseguenza, il calcolo della complessità si concentra su `set_fruitfulAux`. Per il calcolo della complessità temporale asintotica, la ricorrenza di `set_fruitfulAux` è:

$$T(n) = \begin{cases} c & n = 0 \\ 3T\left(\frac{n}{3}\right) + d & n > 0 \end{cases}$$

dove n è il numero dei nodi dell'albero. La ricorrenza sfrutta l'ipotesi per cui l'albero in input è *completo*, quindi, dato un nodo u radice di un albero di n nodi, ciascun sottoalbero contiene approssimativamente $\frac{n}{3}$ nodi.

Possiamo quindi sfruttare il teorema master per risolvere la 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 2. Si scriva una funzione efficiente per stabilire se un albero binario è **quasi completo**, cioè tutti i livelli dell'albero sono completamente riempiti, tranne eventualmente l'ultimo che ha le foglie addossate a sinistra. Calcolare la complessità al caso pessimo dell'algoritmo indicando, e risolvendo, la corrispondente relazione di ricorrenza.

La rappresentazione dell'albero binario utilizza esclusivamente i campi **left**, **right** e **key**. Il prototipo della funzione è:

```
bool isQuasiCompleto(PNode r)
```

Restituisce true se la proprietà è verificata altrimenti false.
Analizzare e motivare in modo chiaro, preciso ed approfondito la complessità della funzione.

Il tipo `PNode` è così definito:

```
struct Node {
       int key;
       Node* left;
       Node* right;
};

typedef Node* PNode;
```

Soluzione:

```
/* Resituisce 0 se l'albero radicato in u e' completo, 1 se l'albero non e'
    completo ma e' quasi completo, 2 se non e' quasi completo */
int auxquasicompleto(PNode u, int& h){

  int hsx, hdx, risx, risdx;

  if (u == nullptr){
    h = -1;
    return 0;
  }

  risx = auxquasicompleto(u->left, hsx);
  risdx = auxquasicompleto(u->right, hdx);

  //calcola l'altezza dell'albero radicato in u
  h = (hsx < hdx ? hdx : hsx) + 1;
  /* L'albero radicato in u e' quasi completo se:
  - entrambi i sottoalberi sono completi e il sottoalbero destro differisce in
    altezza dal sottoalbero sinistro al massimo di 1.
  - il sottoalbero sinistro e' completo mentre il sottoalbero destro e' quasi
    completo, ma i due sottoalberi hanno la stessa altezza.
  - il sottoalbero sinistro e' quasi completo mentre il sottoalbero destro e'
    completo, ma il sottoalbero sinistro e' piu' alto di 1 del sottoalbero
    destro.
  */
  if ((risx == 0 && risdx == 0 && hdx <= hsx && hsx <= hdx + 1) || (risx == 0
    && risdx == 1 && hsx == hdx) || (risx == 1 && risdx == 0 && hsx == hdx +
    1))
    // se l'altezza del sottoalbero destro e' minore del sottoalbero sinistro
    allora l'albero radicato in u e' quasi completo, altrimenti entrambi i
    sottoalberi presentano la stessa altezza e l'albero radicato in u e'
    completo o quasi completo in base alla proprieta' del sottoalbero destro.
    return (hdx < hsx ? 1 : risdx);
  return 2;
}

/* Restituisce se l'albero radicato in u e' quasi completo */
bool quasicompleto(PNode u){

  int h;
  return auxquasicompleto(u, h) < 2;

}
```

La funzione `quasicompleto` chiama `auxquasicompleto` e poi effettua un confronto il cui costo è costante. Di conseguenza, il calcolo della complessità si concentra su `auxquasicompleto`. Per il calcolo della complessità temporale asintotica nel caso peggiore, la ricorrenza di `auxquasicompleto` 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 3. La larghezza di un albero è il numero massimo di nodi che stanno tutti al medesimo livello. Si fornisca una funzione che calcoli in tempo ottimo la larghezza di un albero generale T di n nodi. I nodi hanno campi **key**, **left_child** e **right_sib**.

Il prototipo della funzione è:

```
int larghezza(PNodeG r)
```

Analizzare e motivare in modo chiaro, preciso ed approfondito la complessità della funzione. Il tipo `PNodeG` è così definito:

```
struct NodeG {
       int key;
       NodeG* left_child;
       NodeG* right_sib;
};

typedef NodeG* PNodeG;
```

Soluzione: Di seguito sono proposte due soluzioni una iterativa e una ricorsiva, entrambe le soluzioni sono valutate allo stesso modo in quanto hanno la stessa complessità.

Soluzione I:

```
/*Ritorna il numero massimo di nodi che stanno tutti al medesimo livello*/
int larghezza(PNodeG r){

  queue<PNodeG> q;
  int contaNodiLiv;
  int larg;
  PNodeG u, iter;

  if (r == nullptr)
    return 0;

  contaNodiLiv = 1; //numero di nodi al livello corrente
  larg = 1;  //il valore massimo della larghezza

  //visita in ampiezza dell'albero
  q.push(r);
```

```
  while(!q.empty()){
    u = q.front();
    q.pop();
    iter = u->left_child;
    while (iter){
      q.push(iter);
      iter = iter->right_sib;
    }

    contaNodiLiv--;

    /*se il contatore e' a zero ho visittato tutti i nodi di un livello
    quindi nella coda sono presenti tutti i nodi del nuovo livello */
    if (contaNodiLiv == 0){
      contaNodiLiv = q.size();
      larg = (larg<contaNodiLiv?contaNodiLiv:larg); //aggiorno larg se
    necessario
    }
  }

  return larg;
}
```

Nella soluzione viene implementata una visita in ampiezza dove ogni nodo viene visitato esattamente una sola volta. La complessità delle operazioni eseguite durante la visita di un nodo è costante. Quindi la complessità finale è $\Theta(n)$.

Soluzione II:

```
void larghezzaBisaux(PNodeG u, vector<int>& levelsWidth, int i){

  if (u != nullptr){
    if (i >= levelsWidth.size())
      levelsWidth.resize(levelsWidth.size()*2, 0);

    levelsWidth[i]+= 1;
    larghezzaBisaux(u->left_child, levelsWidth, i+1);
    larghezzaBisaux(u->right_sib, levelsWidth, i);
  }
}

int larghezzaBis(PNodeG r){

  vector<int> levelsWidth;
  int max;
  if (r == nullptr)
    return 0;

  levelsWidth.resize(1, 0);

  larghezzaBisaux(r, levelsWidth, 0);

  max = levelsWidth[0];
  for (int i = 1; i < levelsWidth.size(); i++)
```

```
    if (max < levelsWidth[i])
      max = levelsWidth[i];

  return max;
}
```

Anche in questa soluzione viene implementata una visita in ampiezza dove ogni nodo viene visitato esattamente una sola volta. Se si assume che la funzione `resize` abbia tempo di esecuzione costante, la complessità delle operazioni eseguite durante la visita di un nodo è costante. In alternativa, si può calcolare preliminarmente la profondità dell'albero in tempo $\theta(n)$ con n numero di nodi dell'albero e allocare `levelsWidth` con la dimensione appropriata. Quindi la complessità finale è $\Theta(n)$.

Esercizio 4. Dato un vettore di **n** interi, eventualmente ripetuti, ordinati in senso NON crescente, e un intero k, definire una funzione occ, di complessità ottima, che conta il numero di occorrenze di **k** in **v**. Il prototipo della funzione è:

```
int occ(const vector<int>& v, int k)
```

Analizzare e motivare in modo chiaro, preciso ed approfondito la complessità della funzione

Soluzione:

```
// Funzione di supporto per trovare il primo indice in cui appare k
int lower_bound(const vector<int>& v, int k, int inf, int sup){

    int med;

    if (inf > sup)
    return -1;

    med = (inf + sup)/2;

    if (v[med] == k){
        int ris = lower_bound(v, k, inf, med -1);
        if (ris != -1)
            return ris;

        return med;
    }

    if (v[med] < k)
        return lower_bound(v, k, inf, med -1);

    return lower_bound(v, k, med + 1, sup);
}

// Funzione di supporto per trovare l'ultimo indice in cui appare k
int upper_bound(const vector<int>& v, int k, int inf, int sup){
```

```
    int med;

    if (inf > sup)
    return -1;

    med = (inf + sup)/2;

    if (v[med] == k){
        int ris = upper_bound(v, k, med + 1, sup);
        if (ris != -1)
            return ris;

        return med;
    }

    if (v[med] < k)
        return upper_bound(v, k, inf, med - 1);

    return upper_bound(v, k, med + 1, sup);
}

// Funzione principale
int occ(const vector<int>& v, int k) {

    int first = lower_bound(v, k);
    if (first == -1) {
        return 0;  // k non e' presente nel vettore
    }
    int last = upper_bound(v, k);
    return last - first + 1;  // Numero di occorrenze
}
```

La complessità di `occ` è data dalla somma delle complessità di `lower_bound` e `upper_bound`. Analizziamo quindi la loro complessità. Entrambe le funzioni implementano una ricerca binaria sul vettore v di lunghezza n. La ricorrenza di ciascuna delle due funzioni è:

$$T(n) = \begin{cases} c & n = 0 \\ T\left(\frac{n}{2}\right) + d & n > 0 \end{cases}$$

dove n è la dimensione dell'array.

Possiamo quindi sfruttare il teorema master per risolvere la ricorrenza. Dato che $f(n) = d$, $a = 1$ e $b = 2$, si può verificare che si ricade nel secondo caso del teorema master. Infatti, $f(n) = \theta(n^{log_2(1)})$, dunque $T(n) = \Theta(log(n))$.

In conclusione, la complessità di `occ` è $\Theta(2 \log n) = \Theta(\log n)$.

Esercizio 5. Un videogioco prevede che il proprio avatar venga paracadutato su una mappa. La mappa è divisa in celle a cui è associato un numero indicante l'altitudine. Un giocatore furbo deve far atterrare il proprio avatar in una cella con un'altitudine superiore o uguale a quella delle celle adiacenti per non essere svantaggiato. Infatti, il giocatore furbo deve poter attaccare dall'alto i giocatori atterrati su celle con altitudine

più bassa o almeno deve poter fronteggiare gli altri giocatori ad armi pari, cioè a pari altitudine. Tuttavia, il giocatore deve decidere in fretta dove far atterrare il proprio avatar, in quanto aprire il paracadute troppo tardi causerà la morte del proprio avatar e la perdita della partita ancora prima di iniziarla. Supponi di essere il giocatore e di rappresentare la mappa nella tua mente come una matrice quadrata **n** x **n** di numeri positivi, mentre le locazioni esterne alla mappa sono rappresentate con un numero negativo. Sviluppa un algoritmo divide et impera EFFICIENTE per trovare le coordinate di una cella che non fornisca uno svantaggio una volta atterrato. Lo spazio aggiuntivo utilizzabile è O(1).

Il prototipo della procedura è:

```
void findGoodCell(const vector<vector<int>>& map, int columns, int&
                  index_row, int& index_column)
```

Analizzare e motivare in modo chiaro, preciso ed approfondito la complessità della funzione.

Soluzione:

Di seguito sono proposte due soluzioni:

Soluzione I: (Valutazione 1 punto).

```
int findMax(const vector<vector<int>>& map, int rows, int column, int& max) {

    int max_index = 0;
    for (int i = 0; i < rows; i++) {
        if (max < map[i][column]) {
            max = map[i][column];
            max_index = i;
        }
    }
    return max_index;
}

void findGoodCellAux(const vector<vector<int>>& map, int start, int end, int
    rows, int& index_row, int& index_column) {

    // Trova il massimo elemento nella colonna in mezzo -> candidato come good
    cell
    int max = 0;
    int max_index;

    // Caso base: siamo nell'unica colonna presente/rimasta
    if (start == end) {
        max_index = findMax(map, rows, start, max);
        index_column = start;
        index_row = max_index;
        return; // Fermati qui
    }
```

```
    int mid = (start + end) / 2;

    max_index = findMax(map, rows, mid, max);

    // Vai nella sottomatrice a sinistra se il vicino sulla stessa riga e'
    maggiore
    if (mid > start && map[max_index][mid - 1] > max) {
        return findGoodCellAux(map, start, mid - 1, rows, index_row,
    index_column);
    }
    // Vai nella sottomatrice a destra se il vicino sulla stessa riga e'
    maggiore
    else if (mid < end && map[max_index][mid + 1] > max) {
        return findGoodCellAux(map, mid + 1, end, rows, index_row, index_column
    );
    }
    // Trovata una good cell
    else {
        index_column = mid;
        index_row = max_index;
    }
}

/* Pre: map non e' vuota */
void findGoodCell(const vector<vector<int>>& map, int columns, int& index_row,
    int& index_column) {

    int rows = map.size();
    findGoodCellAux(map, 0, columns - 1, rows, index_row, index_column);
}
```

La procedura `findGoodCell` chiama `findGoodCellAux` dopo aver effettuato operazioni dal costo costante. Di conseguenza, il calcolo della complessità si concentra su `findGoodCellAux`. Siano n il numero di righe della matrice `map` e m il suo numero di colonne. Il costo di una chiamata a `findMax` è $\Theta(n)$, in quanto la funzione scorre le celle di una colonna della matrice. La ricorrenza di `findGoodCellAux` è:

$$T(n,m) = \begin{cases} cn & m = 1 \\ T\left(n, \frac{m}{2}\right) + dn & m > 1 \end{cases}$$

con c e d costanti.

La risoluzione della ricorrenza tramite srotolamento (supponendo $m > 1$) è:

$$
\begin{aligned}
T(n,m) &= T\left(n, \frac{m}{2}\right) + dn \\
&= T\left(n, \frac{m}{4}\right) + dn + dn = T\left(n, \frac{m}{2^2}\right) + 2dn \\
&= T\left(n, \frac{m}{2^3}\right) + 3dn \\
&\vdots \\
&= T\left(n, \frac{m}{2^{\log_2(m)}}\right) + \log_2(m)dn = T(n,1) + \log_2(m)dn \\
&= cn + \log_2(m)dn
\end{aligned}
$$

L'esecuzione della funzione `findGoodCellAux` può terminare prima di $\log_2(m)$ chiamate se viene trovata prima una cella vantaggiosa (possono esistere più celle vantaggiose all'interno della matrice). Dunque $T(n,m) = \mathcal{O}(n\log_2(m))$. Dato che si assume che la matrice in input sia quadrata, $n \times n$, quindi $m = n$, si ottiene $\mathcal{O}(n\log_2(n))$.

Soluzione II: (valutazione 2 punti)

```
// Trova il massimo nella window: riga centrale, colonna centrale e tutti i
    bordi
void findMaxInWindow(const vector<vector<int>>& matrix, int rowStart, int
    rowEnd, int colStart, int colEnd, int& maxRow, int& maxCol) {

    int maxVal = matrix[rowStart][colStart];
    int midRow = (rowStart + rowEnd) / 2;
    int midCol = (colStart + colEnd) / 2;

    // Scansiona la colonna centrale
    for (int i = rowStart; i <= rowEnd; i++) {
        if (matrix[i][midCol] > maxVal) {
            maxVal = matrix[i][midCol];
            maxRow = i;
            maxCol = midCol;
        }
    }

    // Scansiona la riga centrale
    for (int j = colStart; j <= colEnd; j++) {
        if (matrix[midRow][j] > maxVal) {
            maxVal = matrix[midRow][j];
            maxRow = midRow;
            maxCol = j;
        }
    }

    // Scansiona i bordi della finestra
    for (int i = rowStart; i <= rowEnd; i++) {
        if (matrix[i][colStart] > maxVal) {
```

```
            maxVal = matrix[i][colStart];
            maxRow = i;
            maxCol = colStart;
        }
        if (matrix[i][colEnd] > maxVal) {
            maxVal = matrix[i][colEnd];
            maxRow = i;
            maxCol = colEnd;
        }
    }

    for (int j = colStart; j <= colEnd; j++) {
        if (matrix[rowStart][j] > maxVal) {
            maxVal = matrix[rowStart][j];
            maxRow = rowStart;
            maxCol = j;
        }
        if (matrix[rowEnd][j] > maxVal) {
            maxVal = matrix[rowEnd][j];
            maxRow = rowEnd;
            maxCol = j;
        }
    }
}

void findGoodCellAux(const vector<vector<int>>& map, int rowStart, int rowEnd,
    int colStart, int colEnd, int& index_row, int& index_column) {

    if (rowStart > rowEnd || colStart > colEnd) return;

    // Trova il massimo nella window corrente
    int maxRow = rowStart, maxCol = colStart;
    findMaxInWindow(map, rowStart, rowEnd, colStart, colEnd, maxRow, maxCol);

    int maxVal = map[maxRow][maxCol];

    // Controlla i vicini
    int above = (maxRow > rowStart) ? map[maxRow - 1][maxCol] : -1;
    int below = (maxRow < rowEnd) ? map[maxRow + 1][maxCol] : -1;
    int left = (maxCol > colStart) ? map[maxRow][maxCol - 1] : -1;
    int right = (maxCol < colEnd) ? map[maxRow][maxCol + 1] : -1;

    // Se e' un picco
    if (maxVal >= above && maxVal >= below && maxVal >= left && maxVal >= right
    ) {
        index_row = maxRow;
        index_column = maxCol;
        return;
    }

    // Determina il vicino con valore massimo
    int realMaxRow = maxRow, realMaxCol = maxCol;
```

```
    if (above > maxVal) {
        realMaxRow = maxRow - 1;
        }
    else if (below > maxVal) {
            realMaxRow = maxRow + 1;
            }
         else if (left > maxVal) {
                realMaxCol = maxCol - 1;
                }
              else
                realMaxCol = maxCol + 1;

    // Determina il quadrante corretto
    int midRow = (rowStart + rowEnd) / 2;
    int midCol = (colStart + colEnd) / 2;

    if (realMaxRow <= midRow && realMaxCol <= midCol) { // Quadrante in alto a
    sinistra
        findGoodCellAux(map, rowStart, midRow, colStart, midCol, index_row,
    index_column);
    } else if (realMaxRow <= midRow && realMaxCol > midCol) { // Quadrante in
    alto a destra
        findGoodCellAux(map, rowStart, midRow, midCol + 1, colEnd, index_row,
    index_column);
    } else if (realMaxRow > midRow && realMaxCol <= midCol) { // Quadrante in
    basso a sinistra
        findGoodCellAux(map, midRow + 1, rowEnd, colStart, midCol, index_row,
    index_column);
    } else { // Quadrante in basso a destra
        findGoodCellAux(map, midRow + 1, rowEnd, midCol + 1, colEnd, index_row,
     index_column);
    }
}

/* Pre: map non e' vuota */
void findGoodCell(const vector<vector<int>>& map, int columns, int& index_row,
    int& index_column) {

    int rows = map.size();
    findGoodCellAux(map, 0, rows - 1, 0, columns - 1, index_row, index_column);
}
```

La procedura `findGoodCell` chiama `findGoodCellAux` dopo aver effettuato operazioni dal costo costante. Di conseguenza, il calcolo della complessità si concentra su `findGoodCellAux`. Sia n l'ordine della matrice quadrata `map` (cioè il numero delle righe o delle colonne). Il costo di una chiamata a `findMaxInWindow` è $\Theta(n)$, in quanto la funzione scorre le celle della colonna centrale, della riga centrale e dei bordi della matrice. La ricorrenza di `findGoodCellAux` è:

$$T(n) = \begin{cases} d_1 & n \leq 1 \\ T\left(\frac{n}{2}\right) + d_2 n & n > 1 \end{cases}$$

Possiamo sfruttare il teorema master per risolvere la ricorrenza. Dato che $f(n) = d_2 n$,

$a = 1$ e $b = 2$, si può verificare che si ricade nel terzo caso del teorema master. Infatti, fissato $\epsilon = 1$, si ottiene $f(n) = \Omega(n^{\log_2(1)+1}) = \Omega(n)$. Inoltre, occorre trovare una costante c tale che $af\left(\frac{n}{b}\right) \leq cf(n)$, cioè $d_2\frac{n}{2} \leq cd_2n$. Per esempio possiamo fissare $c = \frac{1}{2}$. Dunque $T(n) = \mathcal{O}(n)$, dato che l'esecuzione della funzione `findGoodCellAux` termina non appena viene trovata una cella vantaggiosa.