Soluzioni Esercitazione I ASD 2023-24

Lorenzo Cazzaro Riccardo Maso Alessandra Raffaetà

Esercizio 1. Un albero binario si dice **t-bilanciato** se per ogni suo nodo vale la proprietà: le altezze dei sottoalberi radicati nei suoi figli differiscono per al più **t** unità.

1. Dato un albero binario, scrivere una funzione **EFFICIENTE** che restituisca il minimo valore **t** per cui l'albero risulti **t**-bilanciato.
2. Discutere la complessità della soluzione trovata.

Il prototipo della funzione è:

```
int tBil(PNode u)
```

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:

```
/*Funzione ausiliaria che tiene anche traccia dell'altezza di u.*/
int tBilAux(PNode u, int& h){

  int tsx, tdx, hsx, hdx, diff, t;

  /*caso base*/
  if (u == nullptr){
    h = -1;
    return 0;
  }

  /*trova l'altezza e il valore t per il sottoalbero sinistro e destro*/
  tsx = tBilAux(u->left, hsx);
  tdx = tBilAux(u->right, hdx);
```

```
  diff = abs(hsx - hdx);

  /*trova t per u*/
  t = max(diff, max(tsx, tdx));

  /*ottiene l'altezza di u*/
  h = max(hsx, hdx) + 1;

  return t;
}

/*Funzione che restituisce il minimo valore t per cui u e' t-bilanciato.*/
int tBil(PNode u){
    int h;

    return tBilAux(u, h);
}
```

Per il calcolo della complessità temporale asintotica, la relazione di ricorrenza di `tBilAux` 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 $\Theta(n)$.

La complessità di `tBil` dipende unicamente da quella di `tBilAux`, dato che viene eseguita un'operazione dal costo costante e viene poi chiamata `tBilAux`, quindi la complessità asintotica temporale di `tBil` 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 verifichi la seguente proprietà: per ogni nodo `u`, le chiavi dei figli del nodo `u` devono avere valori non decrescenti considerando i figli di `u` da sinistra verso destra.

Il prototipo della funzione è:

`bool isNonDec(PNodeG 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 `PNodeG` è così definito:

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

```
};

typedef NodeG* PNodeG;
```

Soluzione:

```
/*restituisce se u e' non-decrescente o meno*/
bool isNonDec(PNodeG r){
    /*caso base*/
    if (r == nullptr)
        return true;
    /*c'e' solo il figlio sinistro: verifica la proprieta' sul figlio.*/
    if (r->right_sib == nullptr)
        return isNonDec(r->left_child);
    /*verifica la proprieta' sul figlio e verifica che la chiave rispetti la
    proprieta'.*/
    return (r->key <= r->right_sib->key) && isNonDec(r->right_sib) && isNonDec(
    r->left_child);
}
```

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

Esercizio 3. Dati due vettori contenenti rispettivamente i valori dei nodi tutti **DISTINTI** ottenuti da una visita in ordine anticipato e da una visita in ordine simmetrico di un albero binario, si ricostruisca l'albero binario.

Il prototipo della funzione è:

```
PNode ricostruisci(const vector<int>& va, const vector<int>& vs)
```

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

Soluzione:

```
/*Ricostruisce sottoalberi a partire dai sottovettori delle visite considerati.
    */
PNode ricostruisciAux(const vector<int>& va, int infa, int supa, const vector<
    int>& vs, int infs, int sups){

    int i;
    PNode r;

    if (infa > supa)
        return nullptr;

    /*la chiave della radice e' l'elemento va[infa] */
    r = new Node{va[infa],nullptr,nullptr};
```

```
    i = infs;

    /*trova la posizione della chiave della radice nel vettore vs */
    while (vs[i] != va[infa])
        i++;

    /*Ricostruisci i sottoalberi sinistro e destro.*/
    r->left = ricostruisciAux(va, infa+1, infa + (i - infs), vs, infs, i-1);
    r->right = ricostruisciAux(va, infa+(i - infs) + 1, supa, vs, i+1, sups);

    return r;
}

/*Ricostruisce l'albero*/
PNode ricostruisci(const vector<int>& va, const vector<int>& vs){
    return ricostruisciAux(va, 0, va.size()-1, vs, 0, vs.size()-1);
}
```

Per il calcolo della complessità temporale asintotica, la relazione di ricorrenza di `ricostruisciAux` è:

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

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

Il caso peggiore si presenta quando i nodi nei due sottoalberi sono fortemente sbilanciati. Per esempio $k = 0$. Dunque la ricorrenza diventa

$$T(n) = T(0) + T(n-1) + \mathcal{O}(n) = T(n-1) + \mathcal{O}(n)$$

Usando l'albero di ricorsione si può dimostrare che $T(n) = \mathcal{O}(n^2)$.

La complessità di `ricostruisci` dipende unicamente da quella di `ricostruisciAux`, dato che viene subito chiamata `ricostruisciAux`, quindi nel caso peggiore la complessità asintotica temporale di `ricostruisci` è $\mathcal{O}(n^2)$.

Esercizio 4. Dato un albero binario, scrivere una funzione che costruisce un array bidimensionale **mat** tale che, per ogni coppia di nodi **u** e **v**, l'elemento **mat[u][v]** sia il minimo antenato comune di **u** e **v**. La funzione deve richiedere tempo $\mathcal{O}(\mathbf{n^2})$ e il suo prototipo è:

```
void minAntCom(PNode r, vector<vector<int>>& mat)
```

Per semplicità si assuma che i nodi abbiano chiavi numerate da **0** a **n-1** e che queste chiavi identifichino il nodo. Di conseguenza gli indici della matrice rappresentano i nodi e in posizione **mat[u][v]** c'è la chiave del nodo che è il minimo antenato comune di **u** e **v**.

[Il minimo antenato comune di due nodi **u** e **v** è l'antenato comune di **u** e **v** che si trova più lontano dalla radice dell'albero.]

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

Soluzione:

```
/*struttura per un elemento della lista*/
struct Link {
    int num;
    Link * next;
};

typedef Link * Plink;

/*struttura per gestire la lista*/
struct listaTestaCoda{
    Plink testa;
    Plink coda;
};

typedef listaTestaCoda *PlistaTestaCoda;

/*inserisce un elemento nella lista*/
PlistaTestaCoda inserisciTesta(PlistaTestaCoda l, int val){

    Plink p = new Link{val, nullptr};
    if (!l){
        l = new listaTestaCoda;
        l->coda = p;
    }
    else
        p->next = l->testa;
    l->testa = p;

    return l;
}

/*concatena due liste*/
PlistaTestaCoda concatena(PlistaTestaCoda l1, PlistaTestaCoda l2){
    if (!l1)
        return l2;
    if (!l2)
        return l1;

    PlistaTestaCoda temp = l2;
    l1->coda->next = l2->testa;
    l1->coda = l2->coda;
    delete temp;
    return l1;
}

/*funzione ausiliara per calcolare il minimo antenato comune di ogni nodo dell'
    albero*/
```

```
PlistaTestaCoda minAntCom(PNode r, vector<vector<int>>& mat){
    Plink iterdx, itersx;
    /*caso base*/
    if (!r)
        return nullptr;
    /*lista contenente i nodi del sottoalbero sinistro*/
    PlistaTestaCoda lsx = minAntCom(r->left, mat);
    /*lista contenente i nodi del sottoalbero destro*/
    PlistaTestaCoda ldx = minAntCom(r->right, mat);
    mat[r->key][r->key] = r->key;

    if (ldx){
        iterdx = ldx->testa;

        /*combinazioni tra i nodi del sottoalbero destro e del nodo r*/
        while (iterdx){
            mat[iterdx->num][r->key] = r->key;
            mat[r->key][iterdx->num] = r->key;
            iterdx = iterdx->next;
        }
    }
    if (lsx){
        itersx = lsx->testa;

        /*combinazioni tra i nodi del sottoalbero sinistro e del nodo r*/
        while (itersx){
            mat[itersx->num][r->key] = r->key;
            mat[r->key][itersx->num] = r->key;
            if (ldx){
                iterdx = ldx->testa;

                /*Combinazioni tra i nodi del sottoalbero sinistro e del sotto
    albero destro*/
                while (iterdx){
                    mat[iterdx->num][itersx->num] = r->key;
                    mat[itersx->num][iterdx->num] = r->key;
                    iterdx = iterdx->next;
                }
            }
            itersx = itersx->next;
        }
    }

    /*ritorna l'unione della lista sx e dx aggiungendo anche il nodo corrente*/
    return inserisciTesta(concatena(lsx, ldx), r->key);
}

void minAntCom(PNode r, vector<vector<int>>& mat){
    minAntComAux(r, mat);
}
```

L'idea è di andare a fare una visita in post-ordine dell'albero. Di conseguenza quando l'algoritmo si trova in un nodo r, dopo le chiamate ricorsive della funzione su

`r->left` e `r->right`, esso avrà due liste contenenti tutti i valori dei nodi presenti nei suoi sottoalberi sinistro (`lsx`) e destro (`ldx`). Ora il valore delle seguenti celle della matrice verrà impostato ad `r->key`:

- `mat[r->key][r->key]`;
- `mat[r->key][itersx->num]` e `mat[itersx->num][r->key]` $\forall$ `itersx` $\in$ `lsx`;
- `mat[r->key][iterdx->num]` e `mat[iterdx->num][r->key]` $\forall$ `iterdx` $\in$ `ldx`;
- `mat[itersx->num][iterdx->num]` e `mat[iterdx->num][itersx->num]` $\forall$ `iterdx` $\in$ `ldx` $\wedge \forall$ `intersx` $\in$ `lsx`.

Infine la funzione andrà a concatenare le due liste (in tempo costante) ed aggiungere il nodo. Questa nuova lista verrà ritornata.

La complessità della funzione è $\Theta(n^2)$ in quanto ogni singola cella della matrice verrà toccata una ed una sola volta. Questo è garantito in quanto dopo che una combinazione verrà usata, entrambi i valori si troveranno nella stessa lista, quindi non potranno più essere usati per ricreare la stessa combinazione.

Esercizio 5. Nel centro di Preparazione Olimpica Acqua Acetosa "Giulio Onesti" si allenano la nazionale maschile e femminile di sollevamento pesi. Purtroppo, anche qui può accadere che i pesi vengano rimessi nei posti sbagliati. Date **n** coppie di manubri con pesi distinti, i manubri sono posizionati senza un ordine particolare su due rastrelliere. Inizialmente ogni fila ha un numero uguale di manubri. Trattandosi di una palestra professionale ben finanziata, c'è spazio infinito alle due estremità di ciascuna rastrelliera per sostenere eventuali pesi aggiuntivi. Per spostare un manubrio, si può farlo rotolare in uno spazio vicino libero sulla stessa rastrelliera quasi senza sforzo, oppure si può prenderlo, sollevarlo e posizionarlo in un altro posto libero; questo richiede forza proporzionale al suo peso. Qual è il peso più pesante che si deve sollevare per poter mettere pesi identici uno accanto all'altro? Si tenga presente che su ciascuna rastrelliera dopo la riorganizzazione potrebbe esserci sistemato un numero diverso di pesi; questo è consentito.

Il prototipo della funzione è:

```
int PesoMinimo(vector<int> pesi_1, vector<int> pesi_2)
```

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

Soluzione:

Di seguito sono proposte due soluzioni:

Soluzione I: (Valutazione 1.0 punto).

```
bool solve(vector<int>& v, int w) {
```

```
    //valore del peso (singolo) che non si puo' alzare
    int target = -1;
    size_t i = 0;
    bool result = true;

    while (i < v.size() && result) {
        int el = v[i];
        //se el <= w si puo' semplicemente spostare, non si fa niente
        // altrimenti se el > w
        if (el > w) {
            if (target == -1) {
                //prima volta che vedo un peso troppo pesante
                target = el;
            } else {
                if (target != el) {
                    //se il nuovo peso troppo pesante e' diverso dall'ultimo
    visto non si puo' risolvere
                    result = false;
                }
                //la coppia puo' essere appaiata
                else
                    target = -1;
            }
        }
        i++;
    }

    //se target e' uguale a -1 non ci sono pesi non spaiati
    return result && target == -1;
}

int PesoMinimo(vector<int> pesi_1, vector<int> pesi_2) {
    //valori iniziali per la bisection
    int low = 0, high = 0, best;

    /* ricerca del massimo dei pesi nei due vettori */
    for (int val : pesi_1)
      if (high < val)
         high = val;

    for (int val : pesi_2)
      if (high < val)
         high = val;

    best = high;
    //bisection
    while (low <= high) {
        int mid = (high + low) / 2;

        bool works = solve(pesi_1, mid) && solve(pesi_2, mid);
        /*se e' soluzione cerchiamo un valore piu' basso */
        if (works) {
            high = mid - 1;
```

```
            best = mid;
        } else {
            /* se non e' soluzione cerchiamo un peso piu' alto */
            low = mid + 1;
        }
    }
    return best;
}
```

Per andare a trovare la soluzione dell'esercizio vengono eseguiti due cicli for che eseguono n iterazioni dove n è la lunghezza dell'array, quindi la loro complessità è $\Theta(n)$. Dopo aver ottenuto i valori massimo e minimo (messo a 0 che è la soluzione nel caso di un vettore con le coppie già formate), viene eseguita una binary search sullo spazio delle soluzioni, quindi tra 0 ed il massimo peso trovato. La complessità della binary search è $\Theta(\log n)$, come visto a lezione, e ogni iterazione della ricerca va a chiamare la funzione `solve` sui due vettori.
La funzione `solve` esegue un ciclo for che itera n volte, avremmo quindi che la sua complessità è $\Theta(n)$.

Abbiamo quindi, che la complessità di `PesoMinimo` è $\Theta(2n \log n) = \Theta(n \log n)$.

Soluzione II: (valutazione 1.5 punti)

```
int solve(vector<int>& v) {
    // peso massimo da spostare
    int pesomax = 0;

    //elemento pendente da accoppiare (=0 nessun elemento pendente)
    int pending = 0;

    for (int i = 0; i < v.size(); i++)
            //se v[i] <= pesomax si puo' semplicemente spostare, non si fa
    niente
            // altrimenti se v[i]> pesomax
        if (v[i] > pesomax){
            // se non c'e' nessun elemento pendente, si setta v[i] come
    pendente
            if (pending == 0)
                pending = v[i];
            else {
                if (v[i] == pending)
                    //l'elemento puo' essere accoppiato non ci sono piu'
    elementi pendenti
                    pending = 0;
                else {
                    // si deve muovere v[i] oppure pending, si muove il piu'
    piccolo
                    if (v[i] < pending){
                        // se il piu' piccolo e' v[i], diventa il nuovo pesomax
                        pesomax = v[i];
                    }
                    else {
```

```
                        //se il piu' piccolo e' pending, pending e' il nuovo
    max
                        // e v[i] diventa pending
                        pesomax = pending;
                        pending = v[i];
                    }
                }
            }
        }
    //se non ci sono pesi pendenti, si restituisce pesomax
    if (pending == 0)
        return pesomax;
    //se ci sono pesi pendenti si restituisce il max fra pesomax e pending
    return max(pesomax,pending);
}

int PesoMinimo(vector<int> pesi_1, vector<int> pesi_2) {
    int res_1 = solve(pesi_1);
    int res_2 = solve(pesi_2);

    return max(res_1,res_2);
}
```

Per andare a risolvere il problema viene calcolata la soluzione locale per i due vettori $v1$ e $v2$ con la funzione `solve`, la cui complessità è $\Theta(n)$ dove n è il numero di elementi del vettore, in quanto viene eseguito un ciclo for che itera n volte e durante le iterazioni vengono eseguite unicamente operazioni con complessità costante.
Dato che la funzione `solve` viene chiamata due volte all'interno di `PesoMinimo`, la complessità di `PesoMinimo` è $\Theta(2n) = \Theta(n)$.