Soluzioni Esercitazione II ASD 2023-24

Lorenzo Cazzaro   Riccardo Maso   Alessandra Raffaetà

Esercizio 1. Sia `t` un **BST** di dimensione **n** e sia **k** una chiave di `t` tale che `t→root→key < k`. Si vuole costruire un nuovo **BST** `t'` contenente tutte e sole le chiavi di `t` appartenenti all'intervallo [`t`→**root**→**key, k**].

1. Si scriva una funzione **EFFICENTE** per risolvere il problema.
2. Valutare e giustificare la complessità della funzione.

Il prototipo della funzione è:

```
PTree creaBSTInterval(PTree t, int k)
```

Il tipo PTree è così definito:

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

typedef Node* PNode;

struct Tree {
       PNode root;
 };

typedef Tree* PTree;
```

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

Soluzione:

```
/*Funzione ausiliaria per creare un BST contenente le chiavi con valori
    compresi nell'intervallo*/
PNode copyInRange(PNode u, int min, int max, PNode padre) {
    if (!u)
        return nullptr;
    if (u->key > max)
        return copyInRange(u->left, min, max, padre);
```

```
    if (u->key < min)
        return copyInRange(u->right, min, max, padre);
    PNode res = new Node{u->key, padre, nullptr, nullptr};
    res -> left = copyInRange(u->left, min, max, res);
    res -> right = copyInRange(u->right, min, max, res);
    return res;
}

PTree creaBSTInterval(PTree t, int k){
    PTree res = new Tree{nullptr};
    if (t->root)
        res->root = copyInRange(t->root, t->root->key, k, nullptr);
    return res;
}
```

La complessità temporale asintotica di `creaBSTInterval` dipende da quella di `copyInRange`. La relazione di ricorrenza di `copyInRange` è nel caso peggiore:

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

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

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

La complessità di questa funzione è quindi $\mathcal{O}(n)$, dato che non vengono necessariamente visitati tutti i nodi di `t`.

Questa soluzione può restituire un BST sbilanciato. Per ottenere un BST bilanciato, facendo uso di un vettore di appoggio, si può procedere come segue.

```
/*Funzione ausiliaria per creare un vettore contenente le chiavi dei nodi
    ordinate e con valori compresi nell'intervallo*/
void creaIntervallo(PNode u, int min, int max, vector<int>& ris){
    if (u){
        if (u->key < min)
            creaIntervallo(u->right, min, max, ris);
        else
            if (u->key > max)
                creaIntervallo(u->left, min, max, ris);
            else {
                creaIntervallo(u->left, min, max, ris);
                ris.push_back(u->key);
                creaIntervallo(u->right, min, max, ris);
            }
    }
}

PNode creaBST(vector<int> arr, int p, int r, PNode padre){
    if (p > r)
        return nullptr;
    PNode root;
    int med;
```

```
    med = (p+r)/2;
    root = new Node{arr[med], padre, nullptr, nullptr};
    root->left = creaBST(arr, p, med - 1, root);
    root->right = creaBST(arr, med+1, r, root);
    return root;
}

PTree creaBSTInterval(PTree t, int k){
    vector<int> ris;
    PTree tris;

    creaIntervallo(t->root, t->root-> key, k, ris);

    tris = new Tree();
    tris-> root = creaBST(ris, 0, ris.size()-1, nullptr);
    return tris;
}
```

La complessità temporale asintotica di `creaBSTInterval` dipende da quella di `creaIntervallo` e `creaBST`. La relazione di ricorrenza di `creaIntervallo` è nel caso peggiore:

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

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

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

La complessità di questa funzione è quindi $\mathcal{O}(n)$, dato che non vengono necessariamente visitati tutti i nodi di `t`.

La relazione di ricorrenza di `creaBST` è:

$$T(m) = \begin{cases} c & m = 0 \\ 2T(m/2) + d & m > 0 \end{cases}$$

dove m è il numero di chiavi contenute nel vettore `arr`.

Questa equazione di ricorrenza può essere risolta con il Master Theorem. Sia $a = 2$, $b = 2$, $f(m) = d$. Vogliamo dimostrare che ricadiamo nel primo caso del teorema, cioè esiste una costante $\epsilon > 0$ tale che $f(m) = \mathcal{O}(m^{(log_b a - \epsilon)}) = \mathcal{O}(m^{1-\epsilon})$. Poniamo $\epsilon = 1$. Dunque si ottiene che la complessità della funzione è $\Theta(m)$.

La complessità di `creaBSTInterval` è la somma delle complessità di `creaIntervallo` e `creaBST`, quindi la complessità di `creaBSTInterval` è $\mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n)$ perché la dimensione di `ris` è $m \leq n$.

Esercizio 2. Sia dato un albero binario di ricerca `t` con `n` nodi e chiavi naturali. Si scriva una funzione **EFFICENTE** che modifichi `t` in modo che contenga tutte e sole le chiavi pari di `t`. La soluzione deve essere in loco.

Il prototipo della funzione è:

```
void BSTpari(PTree t)
```

Dove il tipo PTree è definito nel seguente modo:

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

typedef Node* PNode;

struct Tree {
       PNode root;
 };

typedef Tree* PTree;
```

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

Soluzione:

```
/*Funzione ausiliaria che restitusce l'albero radicato in u contenente solo i
    nodi pari*/
PNode BSTpariaux(PNode u, PNode& min, PNode& max){

  PNode minsx, maxsx, mindx, maxdx;

  if (u == nullptr){
    min = nullptr;
    max = nullptr;
    return nullptr;
  }
  /*Filtra prima i sottoalberi sinistro e destro di u e salva tramite reference
     i nodi contenenti la chiave minima e massima.*/
  u->left = BSTpariaux(u->left, minsx, maxsx);
  u->right = BSTpariaux(u->right, mindx, maxdx);
  /*La radice u dell'albero contiene una chiave pari: aggiorna solo i puntatori
     ai nodi contenenti la chiave minima e massima dei sottoalberi*/
  if (u->key % 2 == 0){
    if (minsx != nullptr)
      min = minsx;
    else
      min = u;
    if (maxdx != nullptr)
      max = maxdx;
    else
      max = u;
```

```
    return u;
  }
  /*Se u contiene una chiave dispari, bisogna rimuovere u*/
  /*Se il massimo del sottoalbero sinistro esiste, rimpiazza u con il massimo
    del sottoalbero sinistro*/
  if (maxsx!= nullptr){
    if (maxsx->p != u){
      maxsx->p->right = maxsx->left;
      if (maxsx->left != nullptr)
          maxsx->left->p = maxsx->p;
      maxsx->left = u->left;
      u->left->p = maxsx;
    }
    maxsx->p = u->p;
    maxsx->right = u->right;
    if (u->right != nullptr)
      u->right->p = maxsx;
    min = minsx;
    if (maxdx != nullptr)
      max = maxdx;
    else
      max = maxsx;
    delete u;
    return maxsx;
  }
  /*Se il minimo del sottoalbero destro esiste, rimpiazza u con il minimo del
    sottoalbero destro*/
  if (mindx!= nullptr){
    if (mindx->p != u){
      mindx->p->left = mindx->right;
      if (mindx->right != nullptr)
          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;
    delete u;
    return mindx;
  }
  min = nullptr;
  max = nullptr;
  delete u;
  return nullptr;
}

void BSTpari(PTree t){

  PNode min, max;
  t -> root = BSTpariaux(t->root, min, max);
}
```

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

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

La complessità di questa funzione è quindi $\Theta(n)$.

Esercizio 3. Dato un **vettore v di n numeri naturali**, scrivere una procedura **EFFICENTE** che **ordini v** in modo tale che nel vettore risultante, dati **i** e **j** con $0 \leq i \leq j \leq n-1$, vale $mod(v[i], 3) \leq mod(v[j], 3)$, dove **mod(x, 3)** è il **resto** della divisione di **x** per **3**. Dire se la soluzione proposta è in loco e se è stabile. Valutare e giustificare la complessità della procedura proposta. Il prototipo della funzione è:

```
void ordinaMod3(vector<int>& v)
```

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

Soluzione:

```
void ordinaMod3(vector<int>& v){
    /* L'algoritmo va ad ordinare gli elementi dell'array tenendo conto del
    loro valore modulo 3.*/
    size_t eq0, eq1, eq2;
    eq2 = v.size();
    eq0 = 0;
    eq1 = 0;
    while (eq1 < eq2){
        if (v[eq1]%3 == 0){
            swap(v[eq0], v[eq1]);
            eq0++;
            eq1++;
        }
```

```
        else
            if (v[eq1]% 3 == 1)
                eq1++;
            else {
                eq2--;
                swap(v[eq1], v[eq2]);
            }
    }
}
```

L'idea è di tenere tre indici per memorizzare dove iniziano gli elementi con resto 0, resto 1 e resto 2. Per il calcolo della complessità temporale asintotica, andremo a considerare la complessità dell'unico ciclo while presente. Tale ciclo ha complessità $\Theta(n)$ in quanto ogni elemento dell'array verrà controllato una ed una sola volta. Da ciò possiamo dire che la complessità di `ordinaMod3` è $\Theta(n)$

L'algoritmo è in loco in quanto lo spazio utilizzato è costante quindi $\mathcal{O}(1)$. L'algoritmo non è stabile in quanto non mantiene l'ordinamento iniziale degli elementi il cui resto è uguale, ciò avviene per esempio con gli elementi con resto 2, in quanto il primo elemento il cui resto è 2 sarà l'ultimo elemento dell'array ordinato, mentre il secondo elemento con resto 2, sarà il penultimo elemento dell'array e così via.

Esercizio 4. Sia **H1** un vettore di **lunghezza 2n** contenente un **max-heap** di interi, di dimensione **n**. Sia **H2** un vettore di lunghezza **n** contenente un **max-heap** di interi di lunghezza **n**. Si consideri il problema di **trasformare** il vettore **H1** in un vettore **ordinato** contenente tutti gli elementi degli heap **H1** ed **H2**, **senza allocare altri vettori ausiliari**.

Fornire una procedura **EFFICENTE** per risolvere il problema proposto utilizzando, tra le procedure viste a lezione, **SOLAMENTE max-heapify**. Il prototipo della funzione è:

```
void ordinaMaxHeap(PMaxHeap h1, PMaxHeap h2)
```

Dove il tipo PMaxHeap è definito nel seguente modo:

```
struct MaxHeap {
       vector<int> elements;
       int heapsize;
       int dim;
};

typedef MaxHeap *PMaxHeap;
```

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

Soluzione:

```
/*struttura per definire il max-heap*/
struct MaxHeap{
  vector<int> elements;
  int heapsize;
  int dim;
};

typedef MaxHeap *PMaxHeap;

/*funzione che restituisce l'indice del padre del nodo i */
int parent(int i){
  if (i == 0)
    return -1;

  return (i-1)/2;
}

/*funzione che restituisce l'indice del figlio sinistro del nodo i*/
int left(int i){
  return i*2 + 1;
}

/*funzione che restituisce l'indice del figlio destro del nodo i*/
int right(int i){
  return i*2 + 2;
}

/*max-heapify*/
void maxHeapify(PMaxHeap h, int i){

  int massimo, r, l;
  int temp;

  l = left(i);
  r = right(i);

  if (l < h->heapsize && h->elements[l] > h->elements[i])
    massimo = l;
  else
    massimo = i;

  if (r < h->heapsize && h->elements[r] > h->elements[massimo])
    massimo = r;

  if (i != massimo){
    temp = h->elements[i];
    h->elements[i] = h->elements[massimo];
    h->elements[massimo] = temp;
    maxHeapify(h, massimo);
  }
}
```

```c
/*funzione per ordinare il vettore usando solo la max-heapify*/
void ordinaMaxHeap(PMaxHeap h1, PMaxHeap h2){
    int k = h1->dim - 1;
    while (h1->heapsize > 0 && h2->heapsize > 0){
        if (h1->elements[0] > h2->elements[0]) {
            h1->elements[k] = h1->elements[0];
            h1->elements[0] = h1->elements[h1->heapsize - 1];
            h1->heapsize -= 1;
            maxHeapify(h1, 0);
        }
        else {
            h1->elements[k] = h2->elements[0];
            h2->elements[0] = h2->elements[h2->heapsize - 1];
            h2->heapsize -= 1;
            maxHeapify(h2, 0);
        }
        k--;
    }
    while(h1->heapsize > 0) {
        int tmp = h1->elements[h1->heapsize - 1];
        h1->elements[k] = h1->elements[0];
        h1->elements[0] = tmp;
        h1->heapsize -= 1;
        maxHeapify(h1, 0);
        k--;
    }
    while(h2->heapsize > 0) {
        h1->elements[k] = h2->elements[0];
        h2->elements[0] = h2->elements[h2->heapsize - 1];
        h2->heapsize -= 1;
        maxHeapify(h2, 0);
        k--;
    }
}
```

L'idea è di utilizzare la proprietà dei max-heap per estrarre l'elemento più grande dei due max-heap e inserirlo nell'ultima cella vuota dell'heap H1. Nella prima parte della funzione finché entrambi gli heap contengono elementi viene controllato qual'è il valore più grande tra il primo elemento di H1 ed il primo di H2 e verrà inserito nell'ultima cella libera dello spazio libero di H1. Dopo ogni estrazione viene eseguita una max-heapify per garantire le proprietà degli heap. Nella seconda parte della funzione vengono estratti i restanti elementi presenti in H1 o in H2.

Per calcolare la complessità temporale asintotica della funzione abbiamo che ad ogni iterazione di uno dei tre cicli while viene estratto uno ed un solo valore da uno dei due heap, quindi in quanto gli elementi in totale sono $2n$ (n elementi in H1 ed n elementi in H2) avremo $2n$ iterazioni in totale. A ogni iterazione viene chiamata la funzione `max-heapify` la cui complessità è $\mathcal{O}(\log n)$ come visto a lezione. Da ciò possiamo concludere che la complessità di `ordinaMaxHeap` è $\mathcal{O}(2n \log n)$ ovvero $\mathcal{O}(n \log n)$.

Esercizio 5. 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 $\langle 5, 6, 11, 9, 3\rangle$.

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.

Il prototipo della funzione è

```
int numinversioni(vector<int>& v)
```

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(vector<int>& v, int inf, int med, int sup){
    vector<int> aux;
    int primo = inf, secondo = med +1, count = 0;

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

```
            aux.push_back(v[primo]);
            primo ++;
        }

    if (primo <= med) {
        for (int i = med; i >= primo; i--){
            v[sup] = v[i];
            sup --;
        }
    }
    for (int i = 0; i < aux.size(); i++)
        v[i + inf] = aux[i];

    return count;
}

int inversioni(vector<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);
}

int numinversioni(vector<int>& v){
    return inversioni(v, 0, v.size()-1);
}
```

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 $T(n) = \Theta(n \log n)$.

Esercizio 6. In un parcheggio destinato ad automobili di lusso, **ogni auto** è assicurata per un **determinato valore**. Il parcheggiatore deve **riorganizzare** le auto di modo che la **differenza** tra i valori di auto parcheggiate **consecutivamente** sia la **più piccola possibile**. Più precisamente l'obiettivo è **minimizzare la somma delle differenze** in **valore assoluto** dei valori tra auto adiacenti.

Ovviamente il parcheggiatore vuole ottenere questo risultato con il **numero minimo di scambi** di automobili. Si richiede quindi la creazione di un algoritmo **EFFICENTE** che determini il **numero minimo** di scambi necessari.

Le **auto** sono rappresentate da un **array di numeri naturali**, e **l'output** desiderato è un **singolo numero** naturale che indica il numero **minimo** di **scambi** richiesti.

Il prototipo della funzione è:

int numscambi(vector<int>& v)

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

Soluzione:

```
void mymerge(vector<pair<int, int>>& arr, int p, int med, int r){
    vector<pair<int, int>> aux;
    int i = p, j = med +1;

    while (i <= med && j <= r)
        if (arr[i] <= arr[j]){
        aux.push_back(arr[i]);
        ++i;
    }
    else {
        aux.push_back(arr[j]);
        ++j;
    }
    if (i <= med){
        for (j = med; j>= i; --j){
            arr[r] = arr[j];
            --r;
        }
    }
    for (i = 0; i < aux.size(); ++i)
    arr[p + i] = aux[i];
}

/*N.B.: la funzione di ordinamento doveva essere implementata*/
void mymergesort(vector<pair<int, int>>& arr, int p, int r){
    if (p < r){
        int med = (p + r)/2;
        mymergesort(arr, p, med);
        mymergesort(arr, med + 1, r);
        mymerge(arr, p, med, r);
    }
}

void reverse(vector<pair<int, int>>& arr) {
    int first = 0;
    int last = arr.size()-1;
    while ((first < last)) {
        pair<int, int> var_for_swap;
        var_for_swap = arr[first];
        arr[first] = arr[last];
        arr[last] = var_for_swap;
        first++;
        last--;
    }
}
```

```
int numscambi(vector<int>& v) {
    int result = -1;
    int n = v.size();

    /*crea un array di coppie dove il primo elemento della coppia e' l'elemento
     dell'array v mentre il secondo elemento della coppia e' la posizione in v
    */
    vector<pair<int, int>> arrPos(n);
    for (int i = 0; i < n; i++) {
        arrPos[i].first = v[i];
        arrPos[i].second = i;
    }

    //itero due volte per controllare in quale verso di ordinamento il vettore
    necessita meno swap
    for (int rev = 0; rev < 2; rev++) {
        int curSwap = 0;

        //per tenere traccia degli elementi di v gia' visitati
        vector<bool> vis(n, false);

        //inverto l'array se rev = 1
        //NOTA: la funzione di ordinamento deve essere implementata a parte e
    la complessita' deve essere motivata tramite l'opportuna equazione di
    ricorrenza.
        if (rev) {
            reverse(arrPos);
        } else {
            mymergesort(arrPos, 0, arrPos.size()-1);
        }

        //faccio gli swap necessari per trasformare arr in sorted
        for (int i = 0; i < v.size(); i++) {

            //se ho visitato l'elemento oppure la sua posizione e' gia'
    corretta, salto l'iterazione corrente
            if (!vis[i] && arrPos[i].second != i) {

                /*Trova il numero di nodi da scambiare a partire dal nodo
    corrente. I nodi da scambiare permettono di individuare un ciclo*/
                int cycle_size = 0;
                int j = i;
                while (!vis[j]) {
                    vis[j] = 1;
                    j = arrPos[j].second;
                    cycle_size++;
                }

                /*Aggiorna il numero minimo di scambi*/
                if (cycle_size > 0) {
                    curSwap += (cycle_size - 1);
                }
```

```
            }
        }

        if (result == -1)
            result = curSwap;
        else
            //tengo il valore minore
            result = min(result, curSwap);
    }

    return result;
}
```

La complessità temporale asintotica di `numscambi` dipende dalla complessità della funzione di ordinamento e dei cicli `while` e `for` utilizzati. Il ciclo `while` utilizzato per inizializzare `arrPos` presenta complessità $\Theta(n)$, dove n è il numero di elementi del vettore `v` in input. Il corpo del primo ciclo `for` viene eseguito 2 volte, quindi la sua complessità coincide con quella del corpo. La funzione di ordinamento presenta complessità $\Theta(n \cdot logn)$, se la funzione di ordinamento implementata è una di quelle viste a lezione con complessità ottima. Ad esempio, nella soluzione proposta viene implementato l'algoritmo di ordinamento `Mergesort` tramite la procedura `mymergesort`. La complessità di `reverse`, invece, è $\Theta(n)$, dunque il caso peggiore per l'`if` è $\Theta(n \cdot logn)$. La complessità del ciclo `for` interno è $\Theta(n)$ poiché ciascun elemento del vettore `vis` viene visitato e settato a 1 una sola volta durante l'esecuzione delle n iterazioni. Quindi, la complessità di `numscambi` è $\Theta(n) + \Theta(n \cdot logn) + \Theta(n) = \Theta(n \cdot logn)$.