

## Notazione asintotica

martedì 2 marzo 2021 09:47

2 MARZO 2021

Le notazioni utilizzate sono

O-grande

theta

Omega

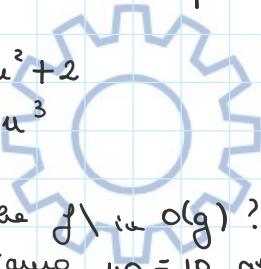
O-grande  $\rightarrow$  Date due funzioni  $f$  e  $g$ , se  $f \in O(g)$  infinitamente

si ha che al tendere dell'argomento all'infinito,  $g$  cresce più velocemente di  $f$

Prima definizione:

$$O(g) = \{ f \mid \exists u_0 \text{ tale che per tutti i valori } u \geq u_0 \text{ allora } f(u) \leq g(u) \}$$

es.  $f(u) = 3u^2 + 2$   
 $g(u) = 2u^3$



- Si ha che  $f \in O(g)$ ?  $\rightarrow$  Vero  
Se scegliamo  $u_0 = 10$  allora  $f(10) = 302$  e  $g(10) = 2000$

- APPUNTI DI INGEGNERIA  
INFORMATICA
- Si ha che  $g \in O(f)$ ?  $\rightarrow$  Falso  
Non esiste nessun valore con la proprietà desiderata

GAIA BERTOLINO

- Si ha che  $g \in O(f)$   $\rightarrow$  Vero
- Si ha che  $f \in O(g)$   $\rightarrow$  Vero
- Si ha che  $f \in O(f)$   $\rightarrow$  Falso

Theta - Omega  $\rightarrow$   $f \in \Theta(g) \Leftrightarrow$  è solo se  $f \in O(g)$  e  $f \in \Omega(g)$

**FIBONACCI 1** → rapporto ampio

**FIBONACCI 2** → costo esponenziale

```
public static int fib2 (int u) {
    if (u ≤ 2) return 1;
    return fib2 (u-1) + fib2 (u-2);
}
```

↳ Albero delle ricorrenze

Ogni nodo è una chiamata ricorsiva

e i modi figli suo le sue sottochiamate  
 $\rightarrow$  COSTO LINEARE

**FIBONACCI 3** → usa un vettore in cui salva le soluzioni

```
public static int fib3 (int u) {
```

int[] f = new int[u];

f[1] = 1;

f[2] = 1;

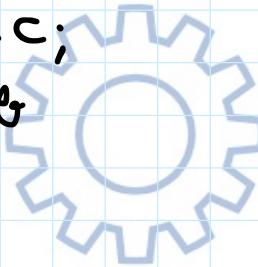
for (int i = 3; i < u; i++)

f[i] = f[i-1] + f[i-2];

return f[u];

FIBONACCI → riduce il costo spaziale

```
public static int fib4 (int n) {  
    int a = 1;  
    int b = 1;  
    int c = 0;  
    for (int i = 3; i < n; i++)  
        c = a + b;  
    a = b;  
    b = c;  
    return b;
```



FIBONACCI 516 → utilizza le matrici

```
public static int fib (int n) {  
    int H[2][2] = {{1, 1}, {1, 0}};  
    for (int i = 0; i < n - 1; i++)  
        H *= H;  
    return H[0][0];
```

$H \times = H_i$

$H[0][0]$

# Codici degli algoritmi

martedì 20 luglio 2021 19:19

## FIBONACCI 6 - Theta(log(n))

```
public static int fib6 (int n) {  
    int[][] a = {{1,1},{1,0}};  
    int[][] m = prodotto(a,n-1);  
    return m[1][1];  
}  
  
public static int[][] prodotto(int[][] a, int n) {  
    if (n<=1) m= {{1,0},{0,1}};  
    else  
        int[][] m = prodotto(A, n/2);  
        m *=m;  
    if (n%2 == 1) m = m*a;  
    return m;  
}
```



## RICERCA SEQUENZIALE (array NON ordinato) -> O(n)

```
public static boolean ricercaSeq(int[] array, int x) {  
    for(int i=0; i<array.length; i++)  
        if (array[i] == x)  
            return true;  
    return false;  
}
```

## RICERCA BINARIA ITERATIVA (array ordinato) -> O(log(n))

```
public static boolean ricercaBinaria(int[] array, int x) {  
    int inizio = 1;  
    int fine = array.length;  
    int medio = 0;  
    while (array[(inizio+fine)/2] != x) {  
        medio = (inizio + fine) /2;  
        if (array[medio] > x)  
            fine = medio - 1;  
        else  
            inizio = medio + 1;  
        if (inizio > fine)  
            return false;  
    }  
    return true;
```

}

RICERCA BINARIA RICORSIVA (array ordinato) -> O(log(n))

```
public static boolean ricercaBinaria(int[] array, int x) {  
    return ricercaBinaria(array, x, 1, array.length);  
}  
  
private static boolean ricercaBinaria(int[] array, int x, int inizio, int fine) {  
    if (fine < inizio) return false;  
    int medio = (inizio+fine)/2;  
    if (array[medio] == x)  
        return true;  
    if (array[medio] > x)  
        return ricercaBinaria(array, x, inizio, medio-1);  
    return ricercaBinaria(array, x, medio +1, fine);  
}
```

ALBERI

— Visita profonda

vista(albero a)

    if a== null

        return

    vista(a.sin())

    salva a.val()

    vista(a.des())

— Visita a livelli

vista(albero a)

    coda c

    c.enqueue(a)

    while coda non è vuota

        preleva il nodo

        salva nodo destro

        salva nodo sinistro

        c.enqueue(a.des())

        c.enqueue(a.sin())

GRAFI

— Visita profonda

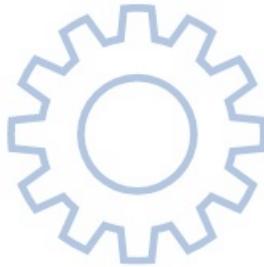
vista(vertice v)

    albero T

    vista(v, T)

    return T

vista(vertice v, albero T)



## APPUNTI DI INGEGNERIA INFORMATICA

GAIA BERTOLINO

```

marca v
for each arco(v, x)
    if x non marcato
        aggiungi arco(v,x) a T
        verifica(x,T)

```

— Visita a livelli

```

visita(vertice c)
albero T
coda c
aggiungi v a T
aggiungi v a c
marca v
while coda non è vuota
    estrai v
    for each arco(v,x)
        if x non marcato
            aggiungi x a c
            aggiungi x a T
            marca x

```

SELECTION SORT -> Theta( $n^2$ )

Seleziona sempre l'elemento minore e lo mette in coda alla parte di array ordinata



```

public static void selectionSort (int[] array) {
    for (int i=0; i<array.length-1; i++) {
        int index = i;
        for (int j=i+1; j<array.length; j++) {
            if(array[j] < array[index])
                index = j;
        }
        int a = array[i];
        array[i] = array[index];
        array[index] = a;
    }
}

```

INSERTION SORT -> Theta( $n^2$ )

Seleziona un elemento alla volta e lo pone nella sua posizione corretta nella metà ordinata dell'array

```

public static void insertionSort(int[] array) {
    for (int i=1; i<array.length-1; i++) {
        int x = array[i];
        int index = 0;
        for (int j=0; j<i; j++) {
            index = j;
            if (array[j] > x)

```

```

        break;
    }
    if (index < i) {
        ùàf or (int k=i; k>index; k - -)
            array[k] = array[k-1];
    }
    array[index] = x;
}
}

```

INSERTION SORT -> O( $n^2$ )  
(manca)

QUICKSORT -> O( $n^2$ )

Il perno in questo caso è sempre il primo elemento che alla fine posiziono nella sua collocazione adeguata scambiandolo con il valore minimo trovato.

Nel caso migliore scelgo sempre il mediano e dunque il costo è Theta( $n \log(n)$ )

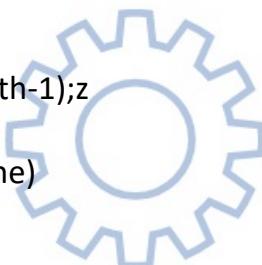
—Pseudocodice

```

quickSort(array a)
    return quickSort(a, 0, a.length-1);z

quickSort(array a, int inizio, int fine)
    if (inizio >= fine)
        return;
    m = partiona(a, inizio, fine)
    quickSort(a, inizio, m-1)
    quickSort(a, fine, m+1)

```



partiona(array a, int inizio, int fine) A BERTOLINO

```

x = array[inizio]
inf = inizio;
sup = fine;
while (true)
    while( inf <= fine && a[inf] < x)
        inf++;
    while (sup >= inizio && a[sup] > x)
        sup---;
    if (inf < sup) {
        temp = a[inf]
        a[inf] = a[sup]
        a[sup] = temp
    }
    else
        break;
return sup;

```

MERGESORT -> Theta( $n \log(n)$ )

Non ordina in loco ma usa un array ausiliario

Compie n confronti e spezza in  $\log(n)$  pezzi

— Java

```
public static void mergeSort(int[] array) {
    if (array != null)
        mergeSort(array, 0, v.length-1);
}

private static void mergeSort(int[] array, int inizio, int fine) {
    if (fine<=inizio)
        return;
    int medio = (inizio+fine)/2;
    mergeSort(v, inizio, medio);
    mergeSort(v, medio+1, fine);
    merge(v,inizio, medio, fine);
}

private static void merge(int[] array, int inizio, int medio, int fine) {
    int temp = new int[fine-inizio+1];
    int i = inizio, j = medio +1, k = 0;
    while( i<= medio && j<= fine) {
        if (array[i] < array[j]) {
            temp[k] = v[i];
            k++;
            i++;
        } else {
            temp[k] = v[j];
            k++;
            j++;
        }
    }
    for (; i<=medio; i++) {
        temp[k] = array[i];
        k++;
    }
    for (; j<=fine; j++) {
        temp[k] = array[j];
        k++;
    }
    for (int z=0; z<temp.length; z++)
        array[inizio+z] = temp[z];
}
```

HEAPSORT ->  $O(n \log(n))$

Ordina in loco

heapSort(array a)

```

heap b = heapify(a)
lista L
while ( b non è vuoto)
    massimo = b.val();
    rimuovi la radice
    L.addFirst(massimo)
return L;

```

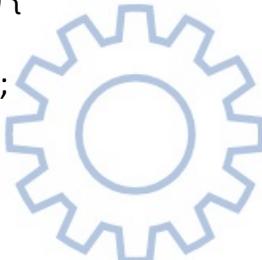
FIXHEAP ->  $O(\log(n))$

—Normale

```

fixHeap(nodo n, heap h)
    if (foglia(n))
        return;
    if (u.des().val() > u.val()) {
        scambia le chiavi
        fixHeap(u.des());
    }
    if (u.sin().val() > u.val()) {
        scambia le chiavi
        fixHeap(u.sin(), h);
    }
}

```



HEAPIFY -> Theta( $n$ )

—Normale

```

heapify(albero h)
    if (n == null)
        return;
    heapify(n.des());
    heapify(n.sin());
    fixHeap(n, h);
}

```

## APPUNTI DI INGEGNERIA INFORMATICA

GAIA BERTOLINO

—Posizionale

```

heapify(array)
    if (array.length == 1)
        return;
    else
        heapify(array[1: array.length/2]);
        heapify(array[array.length/2+1:array.length]);
        fixHeap(

```

### ESTRAZIONE DEL MASSIMO

Il massimo si trova nella radice. Si estraе e si scambia con la foglia situata più a destra. Si applica il fixheap all'albero

RADIXSORT -> Theta( $n$ )

—Java

```

radixSort(array a)
    int max = massimo(a);
    int i = contacifre(max);
    while (i>0) {
        countingSort(array, i)
        i--;
    }
}

```

```

contaCifre(int n)
    int i = 0;
    while (n>0)
        n%10;
        i++;
    return i;
}

```

COUNTINGSORT -> Theta( $n + k$ )

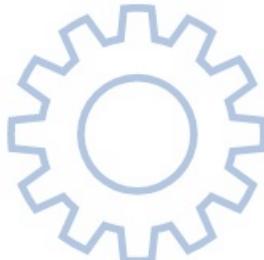
Scorre tutto l'array e in un array di supporto incrementa il rispettivo contatore. Per ordinare l'array poi legge l'array di supporto e mette il numero corrispondente alla cella nella posizione contenuta nella cella

— Java

```

countingSort(array a)
    int max = a[0];
    for (int i=1; i<a.length; i++) {
        if (a[i] > max)
            max = a[i];
    }
    return countingSort(a, max);
}

```



## APPUNTI DI INGEGNERIA INFORMATICA

```

countingSort(array a, int k)      GAIA BERTOLINO
    int[] nuovo = new int[k];
    for (int i=0; i<a.length; i++) {
        nuovo[a[i]] += 1;
    }
    int[] finale = new int[a.length];
    int index = 0;
    for (int i=0; i<nuovo.length; i++) {
        for (int j=i; j>0; j--) {
            finale[index] = i;
            index++;
        }
    }
}

```

DIJKSTRA

— Pseudocodice

```

dijkstra(grafo g, vertice s)
    for each vertice

```

```

distanza uguale a infinito
albero T
aggiungi s a T
coda c
aggiungi s a coda con priorità 0
metti distanza di s a 0
while coda non è vuota
    estrai il minimo in v
    for each arco(v,y)
        if distanza è infinita
            aggiungi y a coda con priorità Dsv + peso(v,y)
            metti Dsy = Dsv + peso(v,y)
            aggiungi y a T
        else if (Dsv + peso(v,y) < Dsy)
            decreaseKey di v in coda
            aggiungi y a coda
return T

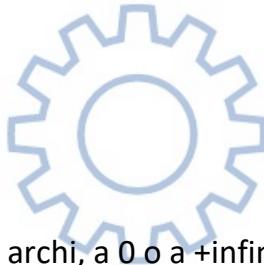
```

FLOYD  
— Spazio cubico

```

floyd(grafo g)
imposto le distanze pari agli archi, a 0 o a +infinito
for each vertice
    for each arco

```



**APPUNTI DI INGEGNERIA  
INFORMATICA**

GAIA BERTOLINO

## 1) METODO COSTO UNIFORME

```

public boolean uve (boolean[] vec) {
    for (int i = 0; i < vec.length; i++) {
        if (!vec[i]) return false;
    }
    return true;
}

caso peggiore → 1 + 1 + 1 + 1 = 4 operazioni. Totale = Theta(4)

caso peggiore → 1 + 1 + u(3) + 1 = 3 + 3u op. Totale =
= Theta (u)

```

Trovare caso peggiore  
e peggiore

In questo caso visto che non compio maggiocazioni o minorazioni devo usare la notazione di asymptotica Theta proprio perché compio un calcolo preciso.

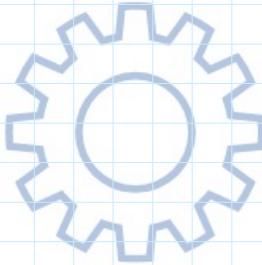
Se il calcolo cui costituisce un valore costante allora la funzione asintotica è 1 ovvero Theta(1)

## 2) METODO COSTO LOGARITMICO

```

public static int fatt (int u) {
    int fattoriale = 1;
    for (int i = 1; i <= u; i++)
        fattoriale *= i;
    return fattoriale;
}

```



In questo caso è come se moltiplicassi  $u$  volte una moltiplicazione che è data da  $\log u!$  +  $\log u$  in quanto la moltiplicazione fra due interi ha costo  $\log x + \log y$ .

Dunque in questo caso ho  $u(\log u! + \log u)$   
che posso scrivere come  $u \log u! = u \log u^u =$   
 $= u^2 \log u$

Dunque la complessità sarà  $O(u^2 \log u)$

## 3) TEORIA DELLE RICORRENZE

## Ricerca Binaria

```
public class RicercaBinaria {
    /**
     * Riceve un vettore ordinato di interi v e un intero x e restituisce
     * la posizione di x in v o -1 se non presente
     */
    public static int ricercaBinariaRic(int[] v, int x) {
        return ricercaBinariaRic(v, x, 0, v.length-1);
    }

    private static int ricercaBinariaRic(int[] v, int x, int inf, int sup) {
        if (x < inf || x > sup) return -1;
        int med = (inf+sup)/2;
        if (v[med]==x) return med;
        if (v[med]<x)
            return ricercaBinariaRic(v, x, inf, med-1);
        else
            return ricercaBinariaRic(v, x, med+1, sup);
    }
}
```

posso rappresentare  $\Theta(n^d)$ ?

Si per  $d=0$

COMPL SPATIALE  
 $\Theta(\log n)$

- $a = 1$ , poiché cerchiamo l'elemento solo su una delle due metà del vettore
- $c = 2$ , poiché dividiamo il vettore in due metà
- $d = 0$  poiché per suddividere il vettore in due metà e comprendere qual è la metà su cui spostarci utilizziamo  $\theta(1)$  operazioni

$$\frac{a}{c^d} = \frac{1}{2^0} = 1$$

quindi siamo nel caso (1) del teorema e quindi la complessità è

$$\theta(n^d \log_c n) \Rightarrow \theta(n^0 \log_2 n) \Rightarrow \theta(\log n)$$

## Merge Sort

```
public class Ordinamento {
    ...

    public static void mergeSort(int[] v) {
        if (v==null) {
            mergeSort(v, 0, v.length-1); // se questa linea scrivo di più poi alla dom dell'espri "per"
        }
    }

    private static void mergeSort(int[] v, int in, int fin) {
        if (fin-in<1) // base
            return;
        int med = (in+fin)/2;
        mergeSort(v, in, med);
        mergeSort(v, med, fin);
        merge(v, in, med, fin); // inserisco qui sotto dove sono altri
    }

    Merge
    private static void merge(int[] v, int in, int med, int fin) {
        ArrayList stage = new ArrayList(in-fin+1);
        int i = in, j = med+1;
        while(i<med & j<fin) {
            if (v[i]<=v[j]) {
                stage.add(v[i]);
                i++;
            } else {
                stage.add(v[j]);
                j++;
            }
        }
        for (int k=i; k<fin; k++)
            stage.add(v[k]);
        for (int k=0; k<stage.size(); k++)
            v[in+k] = stage.get(k);
    }
}
```

CASE DI SPATIALE  
 $\frac{a}{c^d} = \frac{2}{2^0} = 2 \Rightarrow \Theta(n^0 \log_2 n) = \Theta(n \log n)$

Saranno domande se si può ripetere  $f_{n-1} \rightarrow f_n(n)$

COMPL SPATIALE  
NOME

- $a = 2$ , poiché ordiniamo separatamente con lo stesso algoritmo le due metà del vettore
- $c = 2$ , poiché dividiamo il vettore in due metà
- $d = 1$  poiché la funzione merge ha complessità  $\theta(n)$

$$\frac{a}{c^d} = \frac{2}{2^1} = 1 \Rightarrow \text{Caso 1} \dots \Theta(n^0 \log_2 n) = \Theta(n \log n)$$

quindi siamo nel caso (1) del teorema e quindi la complessità è

$$\theta(n^d \log_c n) \Rightarrow \theta(n^1 \log_2 n) \Rightarrow \theta(n \log n)$$

## 4) EX 1 ESAME

**Esercizio 1**  
Si consideri una classe AlberoBinario che rappresenta alberi binari in cui la parte informativa di ogni nodo è un numero intero. Si assume che la classe sia implementata i seguenti metodi:

```
public interface AlberoBinario{
    /* restituisce il ricalcolo dell'albero corrente, la complessità temporale è O(1)*
    public AlberoBinario destro();
    /* restituisce il ricalcolo sinistro dell'albero corrente, la complessità temporale è O(1)*
    public AlberoBinario sinistro();
    /* restituisce il valore memorizzato nella radice dell'albero, la complessità temporale è O(1)*
    public int val();
}
```

Si deve realizzare un metodo ricorsivo

```
public static boolean verifica(AlberoBinario a) {
    if (a == null) return false;
    if (a.destro() == null && a.sinistro() == null && a.val() >= 0)
        return true;
    return verifica(a.destro()) || verifica(a.sinistro());
}
```

La prima cosa da fare è definire quali sono le casistiche, avere caso base e caso generale (o più casi generali)

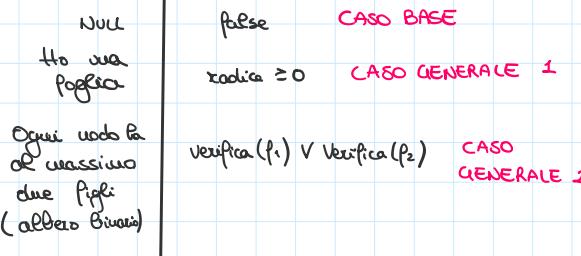
In questo caso il metodo funziona che se esiste una foglia con valore  $\geq 0$  allora restituisce True, senno False.

(albero)	INPUT	OUTPUT	
NULL		false	CASO BASE
Ho una foglia		radice $\geq 0$	CASO GENERALE 1

```
public static boolean verifica(AlberoBinario a) {
    if (a == null) return false;
    if (a.destro() == null && a.sinistro() == null && a.val() >= 0)
        return true;
    return verifica(a.destro()) || verifica(a.sinistro());
}
```

In questo caso avrà nel caso migliore un costo dato dall'altezza dell'albero. Se l'albero è bilanciato allora tale costo è  $\log(n)$  mentre nel caso in cui fosse bilanciato nel caso peggiore sarebbe n

Per quanto riguarda la complessità temporale si può fare lo stesso ragionamento casistico



Procedo a scrivere il metodo!

```
public static boolean verifica(AlberoBinario a) {
  if (a==null)
    return false;
  if(a.destro() == null && a.sinistro() == null)
    return a.val()>0;
  return verifica(a.destro()) || verifica(a.sinistro());
} //GIUSTO perché mi basta che anche una sola foglia sia positiva
```

## 5) EX 1 ESAME

Si consideri una classe AlberoBinario che rappresenta alberi binari in cui la parte informativa di ogni nodo è un numero intero. Si assume che in tale classe siano implementati i seguenti metodi:

```
public interface AlberoBinario{
  /* restituisce il sottoalbero destro dell'albero corrente, la complessità temporale è O(1)*/
  public AlberoBinario dest();
  /* restituisce il sottoalbero sinistro dell'albero corrente, la complessità temporale è O(1)*/
  public AlberoBinario sin();
  /* restituisce il valore memorizzato nella radice dell'albero, la complessità temporale è O(1)*/
  public int val();
}
```

Si deve realizzare un metodo ricorsivo:

```
public static int altezzaAlberoBinario(AlberoBinario a){...}
```

che restituisce l'altezza di a.

Scrivo la cosistica ricordando che è un albero binario

INPUT	OUTPUT
NULL	0
Ho albero un nodo	$1 + \max(\text{altezza destro}, \text{altezza sinistro})$



18:40

```
public static int altezza(AlberoBinario a) {
  if (a==null) return 0;
  return 1 + max(altezza(a.des()), altezza(a.sin()));
}

private static void max(int a, int b) {
  if (a>b) return a;
  return b;
}
```

In questo caso devo analizzare tutto albero quindi avrò una complessità lineare sia nel caso migliore che peggiore.  
Nel caso della complessità spaziale avrò che nel caso migliore ovvero di un AVL la ricerca si riduce a log(n) mentre nel caso peggiore a n.

Scrivo il codice :

```
public static int altezza(AlberoBinario a) {
  if (a==null) return 0;
  return 1+ Math.max(altezza(a.sin()), altezza(a.des()));
}
```

# APPUNTI DI INGEGNERIA INFORMATICA

GAIA BERTOLINO

In questo caso, coincidono nel caso della complessità temporale il caso migliore e quello peggiore.

La complessità spaziale invece sarà nel caso migliore pari a Theta(log n) mentre nel caso peggiore sarà Theta(n) perché faccio almeno una chiamata per ogni nodo

## COMPLESSITÀ SPAZIALE

Caso migliore  $\rightarrow \Theta(\log n)$   
Caso peggiore  $\rightarrow \Theta(n)$

## 6) EX 1 ESAME

Si consideri una classe AlberoBinario che rappresenta alberi binari in cui la parte informativa di ogni nodo è un numero intero. Si assume che in tale classe siano implementati i seguenti metodi:

```
public interface AlberoBinario{
  /* restituisce il sottoalbero destro dell'albero corrente, la complessità temporale è O(1)*/
  public AlberoBinario dest();
  /* restituisce il sottoalbero sinistro dell'albero corrente, la complessità temporale è O(1)*/
  public AlberoBinario sin();
  /* restituisce il valore memorizzato nella radice dell'albero, la complessità temporale è O(1)*/
  public int val();
}
```

Il metodo verifica se esiste un nodo interno di valore v tale che esso è minore o uguale delle somma dei sottoalberi destro e sinistro

```
public static boolean ver(AlberoBinario a) {
  if (a==null) return false;
  if (a.des()== null || a.sin() == null)
    return a.val()<=somma(a.des()) + somma(a.sin()) || ver(a.des()) || ver(a.sin());
}

public static int somma(AlberoBinario a) {
  if (a == null) return 0;
  return a.val() + somma(a.des()) + somma(a.sin());
```

```

    /* restituire il sottoalbero destro dell'albero corrente, la complessità temporale è O(1)*
    public AlberoBinario dest();
    /* restituire il sottoalbero sinistro dell'albero corrente, la complessità temporale è O(1)*
    public AlberoBinario sin();
    /* restituire il valore memorizzato nella radice dell'albero, la complessità temporale è O(1)*
    public int val();
}

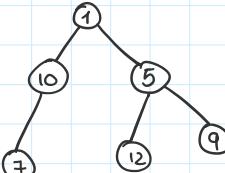
```

Si deve realizzare un metodo risorsa  
pubblica sostituendo questo:  
che restituisce vero se e solo se esiste un nodo non foglia di a avente valore è v o la somma dei valori contenuti nei suoi sottoalberi destro e  
sinistro è minore o uguale a v;

Costo caso :

Ancò due funzioni {   
 ver verifica che  
 $a.sin() \neq null \& a.des() \neq null$   
 $a.des() + a.sin() \leq v$

INPUT	OUTPUT
NULL	0
if $\exists$ figlio	$padre == v \& somma < v$



REGOLA GENERALE:

Se devo restituire una condizione che deve essere almeno una volta vera o almeno una volta falsa posso scrivere una return a parte.  
Se invece devo restituire un confronto (anche se deve valere una sola volta) conviene metterlo nella return principale

In questo caso per ogni nodo devo visitare il sottoalbero destro e sinistro e sommarli. Ciò vorrà dire che visito n volte e per ciascuna volta vado a rivistare l'albero n-1, poi n-2, poi n-4 volte e così via ovvero n-2k volte. Di conseguenza posso dire che la funzione è  $O(n^2)$ .

Nel caso migliore alla prima iterazione mi fermo ovvero quando ho visitato tutti i nodi una volta e cioè Theta(n).

```

public static boolean ver(AlberoBinario a) {
    if (a==null)
        return 0;
    if (a.val() >= somma(a.des())+somma(a.sin()))
        return true;
    return ver(a.des()) || ver(a.sin());
}

public static void somma(AlberoBinario a) {
    if (a==0)
        return 0;
    return a.val() + somma(a.des()) + somma(a.sin());
}

//ancora più compatto diventa

public static boolean ver(AlberoBinario a) {
    if (a==null)
        return 0;
    return a.val() >= somma(a.des()) + somma(a.sin()) || ver(a.des()) || ver(a.sin());
}

public static void somma(AlberoBinario a) {
    if (a==0)
        return 0;
    return a.val() + somma(a.des()) + somma(a.sin());
}

```

dovò fare i controlli (cioè cliccare tutti i nodi) e per ogni nodo dovo fare una somma di sottoalberi destri e sinistri dunque  $O(n^2)$

## COMPLESSITÀ TEMPORALE

Caso migliore  $\rightarrow$  Omega ( $n-1$ )  $\Rightarrow$  Omega( $n$ )  
 Caso peggiore  $\rightarrow$   $O(n^2)$

## 7) EX I ESAME

### Esercizio 1 (per testare)

Si considera una classe AlberoBinario che rappresenta albero binari in cui la parte informativa di ogni nodo è un numero intero. Si assume che in tale classe siano implementati i seguenti metodi:

```

public AlberoBinario alberoDestino();
/* restituire il sottoalbero destro dell'albero corrente, la complessità temporale è O(1)*
public AlberoBinario alberoSinistro();
/* restituire il sottoalbero sinistro dell'albero corrente, la complessità temporale è O(1)*
public AlberoBinario alberoVedi();
/* restituire il valore memorizzato nella radice dell'albero, la complessità temporale è O(1)*
public int val();
}

```

Si deve realizzare un metodo risorsa  
pubblica sostituendo questo:  
che restituisce vero se e solo se inserendo il sottoalbero sinistro e quello destro di ogni nodo in a l'albero non cambia

```

public static boolean simmetrico (AlberoBinario a) {
    if (a==null) return true;
    return simmetrico(a.dest(), a.sin());
}

```

```

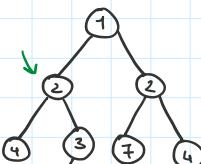
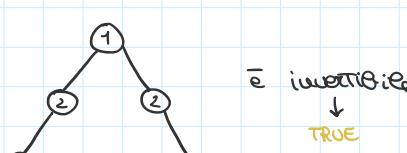
private static boolean simmetrico(AlberoBinario a, AlberoBinario b) {
    if (a==null && b==null) return true;
    if (a==null || b==null) return false;
    return a.val().equals(b.val()) && simmetrico(a.des(), b.sin()) && simmetrico(a.sin(), b.des());
}

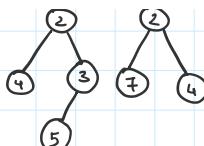
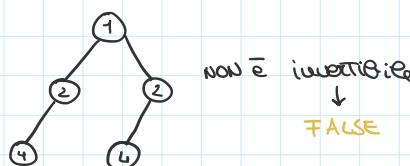
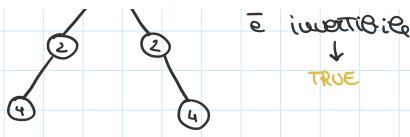
```

In questo caso la complessità temporale migliore è costante nel caso in cui termino subito il controllo mentre quella peggiore visto che devo verificare tutti i nodi, una volta ciascuno sarà Theta(n).

Nel caso della spaziale, nel caso precedente ovvero se si interrompe subito ho costo lineare mentre in quello peggio Theta(n).

Io metto un richiede di scambiare figlio destro e sinistro. L'obiettivo vuol cambiare se i due valori sono uguali una ricordando che deve essere simmetrico. Esempio





```

public static boolean simmetrico(AlberoBinario a) {
    if (a==null) return true;
    return simmetrici(a.sin(), a.des());
}

public static boolean simmetrici(AlberoBinario sin, AlberoBinario des) {
    if (sin == null & des == null)
        return true;
    if (sin == null || des == null)
        return false;
    return sin.val() == des.val() && simmetrici(sin.sin(), des.des()) && simmetrici(sin.des(), des.sin());
}

// i casi di uscita devono essere sempre le situazioni in cui i nodi hanno valore null.
// infatti, se non gestiti bene i casi null sollevano eccezione se si prova ad accedervi.
// DUNQUE -> è importante gestire come casi di uscita i null delle variabili

```

Controlla ogni nodo vera o no

### COSTO TEMPORALE

Caso migliore → Theta(1) es. figlio destro e sinistro diversi

Caso peggiore → Theta(n) perché se l'albero è simmetrico allora la complessità per il metodo dell'operazione dominante sarà n

### COSTO SPAZIALE

Caso migliore → Theta(1)

Caso peggiore → Theta(n)

## 8) ESERCIZI

# APPUNTI DI INGEGNERIA

## INFORMATICA GAIA BERTOLINI

### Esercizio 1

Si consideri una classe `AlberoBinario` che rappresenta alberi binari in cui la parte informativa di ogni nodo è un numero intero. Si assuma che in tale classe siano implementati i seguenti metodi:

```

public interface AlberoBinario{
    /* restituisce il sottoalbero destro dell'albero corrente, la complessità temporale è O(1)*/
    public AlberoBinario destro();
    /* restituisce il sottoalbero sinistro dell'albero corrente, la complessità temporale è O(1)*/
    public AlberoBinario sinistro();
    /* restituisce il valore memorizzato nella radice dell'albero, la complessità temporale è O(1)*/
    public int val();
}

```

Si realizza un metodo ricorsivo

```

public static boolean verificaNodiInterni(AlberoBinario a) {...}

```

che restituisce `true` se per ogni nodo interno `n` (e per la radice qualora questa non sia una foglia) è verificata la condizione che il valore contenuto in `n` è presente anche in almeno un nodo appartenente al sottoalbero sinistro o destro di `n`, `false` altrimenti. Si noti che la condizione non va verificata per i nodi foglia.

Si caratterizza la complessità temporale e spaziale del metodo nel caso migliore e peggiore, specificando anche quali siano il caso migliore ed il caso peggiore per la complessità temporale e spaziale.

Se radice != foglia o se nodo interno allora  
true se nodo.val() è presente in un altro nodo  
Se è foglia ecco (cioè des() e sin() == null)

```
public static boolean verificaNodiInterni(AlberoBinario a) {
```

```
    if (a==null) return true;
    if (a.des()== null && a.sin()== null)
        return true;
```

```
    return (verificaNodiInterni(a.val(), a.des()) || verificaNodiInterni(a.val(), a.sin())) &&
```

```
    verificaNodiInterni(a.des()) && verificaNodiInterni(a.sin()));
```

```
}
```

```
public static boolean verificaNodiInterni(int val, AlberoBinario a) {
```

```
    if ( a == null) return false;
```

```
    return val == a.val() || verificaNodiInterni(val, a.des()) || verificaNodiInterni(val, a.sin());
```

```
}
```

Visto che bisogna analizzare tutti i nodi, il caso migliore si ha se verifico subito per il primo nodo che tale condizione è falsa ovvero Theta(1).

Il caso peggiore si ha quando per ogni nodo devo attraversare i sottoalberi destro e sinistro il che implica una complessità del tipo  $O(n^2)$

La complessità spaziale nel caso migliore sarà Theta(1) perché non devo analizzare tutti nodi mentre Theta(n) nel caso peggiore

INPUT	OUTPUT
NULL	false
Solo radice	false
if contiene(des) e contiene(sin)	true
else	false

Per ogni nodo controllo sottoalbero destro

```
public static boolean verificaNodiInterni(AlberoBinario a) {
```

```

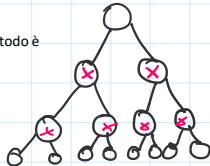
public static boolean verificaNodiInterni(AlberoBinario a) {
    if (a == null) return false;
    if (a.des() == null && a.sin() == null) return false;
    if (a.sin() == null)
        return contiene( a.des(), a.val() );
    else if (a.des() == null)
        return contiene( a.sin(), a.val() );
    return (contiene( a.des(), a.val() ) || contiene( a.sin(), a.val() )) && verificaNodiInterni( a.des() ) && verificaNodiInterni( a.sin() );
}

private static void contiene(AlberoBinario a, int val) {
    if (a==null) return false; //non posso escludere questa riga perché anche questo metodo è ricorsivo e rischio di chiamare val() su un oggetto null
    return a.val() == val || contiene(a.des(), val) || contiene(a.sin(), val);
}

//devo stare molto attenta alle seguenti cose
// 1) mettere le condizioni di uscita (che di solito sono le verifiche dei null)
// 2) richiamare la ricorsione nel metodo principale

```

Per ogni nodo controllo sottoalbero destro e sottoalbero sinistro quindi  $n \cdot n$  e di conseguenza applico poi la verifica quindi  $n^2$



### COSTI DI COMPLESSITÀ TEMPORALE

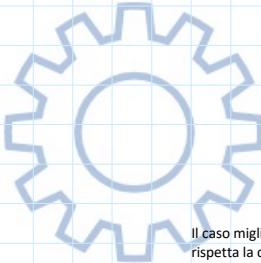
Caso migliore  $\rightarrow \Theta(1)$  poiché esce subito  
Caso peggiore  $\rightarrow \Theta(n^2)$

### COSTI DI COMPLESSITÀ SPAZIALE

Caso migliore  $\rightarrow \Theta(1)$   
Caso peggiore  $\rightarrow \Theta(n)$  poiché proporzionale all'altezza

## 9) ESEMPI

Si realizzi un metodo  
public static boolean verificaNodoLivello(AlberoBinario a, int l) { ... }  
che restituisce true se esiste un nodo n che appare in a ad un livello l  $\Rightarrow$  l'avente valore maggiore di l. Si caratterizzi la complessità temporale e spaziale del metodo nel caso migliore e peggiore, specificando anche quando si ha il caso migliore ed il caso peggiore per la complessità temporale e spaziale.



True se un nodo in a è nel livello  
maggiore del parametro I passato

Il caso migliore si ha quando il nodo radice o uno dei figli rispetta la condizione e dunque Theta(1).

Il caso peggiore si ha quando devo analizzare tutti i nodi; visto che leggo una volta sola tutti i nodi essa sarà Theta(n).

La complessità spaziale sarà Theta(1) nel primo caso e Theta(n) nel secondo

INPUT	OUTPUT
NUL	false
Se ha almeno un figlio	liv++

APPUNTI DI INGEGNERIA  
INFORMATICA  
GAIA BERTOLINO

```

public static boolean verificaNodoLivello(AlberoBinario a, int l) {
    if (a == null) return false;
    if (l == 0) return true;
    return verificaNodoMod(a, l, 0);
}

private static boolean verificaNodoMod(AlberoBinario a, int l, int liv) {
    if (a.des() == null && a.sin() == null) return false;
    liv++;
    if (a.des() == null) return liv>l || verificaNodoMod( a.sin(), l, liv );
    else if (a.sin() == null) return liv>l || verificaNodoMod( a.des(), l, liv );
    return liv>l || verificaNodoMod( a.des(), l, liv ) || verificaNodoMod( a.sin(), l, liv );
}

```

### COSTI DI COMPLESSITÀ TEMPORALE

Caso migliore  $\rightarrow \Theta(1)$   
Caso peggiore  $\rightarrow \Theta(n)$  poiché chiama il metodo n volte

### COSTI DI COMPLESSITÀ SPAZIALE

Caso migliore  $\rightarrow$   
Caso peggiore  $\rightarrow$

10)

**Esercizio 1**

Si consideri una classe *AlberoBinario* che rappresenta *alberi binari* in cui la parte informativa di ogni nodo è un numero intero. Si assume che in tale classe siano implementati i seguenti metodi:

```
public interface AlberoBinario{
    /* restituisce il sottoalbero destro dell'albero corrente, la complessità temporale è O(1)*;
    public AlberoBinario destro();
    /* restituisce il sottoalbero sinistro dell'albero corrente, la complessità temporale è O(1)*;
    public AlberoBinario sinistro();
    /* restituisce il valore memorizzato nella radice dell'albero, la complessità temporale è O(1)*;
    public int val();
}
```

Si deve realizzare un metodo ricorsivo

```
public static boolean verificaAlberoBinario(AlberoBinario a, AlberoBinario b) {
    if (a==null) return false;
    return verifica(a,b,0, 0);
}
```

che restituisce true se e solo se esiste almeno un valore che appare in un nodo di a che si trova ad un livello *liv* e tale valore appare anche in b su un nodo che si trova ad un livello maggiore di *liv*.

//21:45

Tale metodo analizza per ogni nodo l'albero b alla ricerca di un elemento che si trova ad un livello superiore di quello dell'albero a e che abbia lo stesso valore. Deve esistere almeno un valore quindi quando ho verificato che esiste anche un solo nodo posso restituire true.

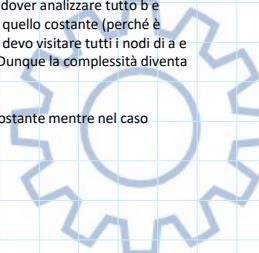
```
public static boolean verifica(AlberoBinario a, AlberoBinario b) {
    if (a==null) return false;
    return verifica(a,b,0, 0);
}

private static boolean verifica (AlberoBinario a, AlberoBinario b, int liva, int libv) {
    if (a==null) return false;
    return proprietà(a.val(), b, liva, libv) || verifica(a.des(), b, liva++, libv++) || verifica(a.sin(), b,
    liva++, libv++);
}

private static boolean proprietà(int val, AlberoBinario b, int liva, int libv) {
    if (b == null) return false;
    return (b.val()==val && libv>liva) || proprietà(val,b.des(), liva, libv++) || proprietà(val, b.sin(), liva,
    libv++);
}
```

La complessità di tale algoritmo deve tenere conto del fatto che per ogni nodo vado ad esaminare l'albero b. Nel caso migliore verifico la condizione dopo pochi passi senza dover analizzare tutto e è senza dover analizzare tutto ae dunque avrò un costo che è proporziale a quello costante (perché è come se considerassi tutte operazioni semplici). Nel caso peggiore invece devo visitare tutti i nodi di b e per ogni nodo di a devo visitare tutti i nodi di b (che dirò che ha m nodi). Dunque la complessità diventa Theta( $n*m$ ).

Per quanto riguarda la complessità spaziale nel primo caso sarà sempre costante mentre nel caso peggiore sarà Theta(n) nel caso della linkedlist



**TI DI INGEGNERIA  
FORMATICA  
GAIA BERTOLINO**

11)

**Esercizio 4**

Si consideri una classe *AlberoBinario* che rappresenta *alberi binari* in cui la parte informativa di ogni nodo è un numero intero. Si assume che in tale classe siano implementati i seguenti metodi:

```
public interface AlberoBinario{
    /* restituisce il sottoalbero destro dell'albero corrente, la complessità temporale è O(1)*;
    public AlberoBinario destro();
    /* restituisce il sottoalbero sinistro dell'albero corrente, la complessità temporale è O(1)*;
    public AlberoBinario sinistro();
    /* restituisce il valore memorizzato nella radice dell'albero, la complessità temporale è O(1)*;
    public int val();
}
```

Si deve realizzare un metodo ricorsivo

```
public static boolean verificaAlberoBinario(AlberoBinario a) {
    return verifica(a, 0, 0);
}
```

che restituisce true se e solo se esiste almeno un valore che appare in un nodo di a che si trova ad un livello *liv* e tale valore appare anche in b su un nodo che si trova ad un livello maggiore di *liv*.

In questo caso è dato un albero binario che potrebbe essere aallora anche sbilanciato e non ordinata (caso molto generale).

```
public static boolean verifica(AlberoBinario a, int liva, AlberoBinario b) {
    if (a == null) return false;
    return verifica(a, liva, b, liva) || verifica(a.des(), liva++, b, 0) || verifica(a.sin(), liva++, b, 0);
}

public static boolean verifica(AlberoBinario a, int liva, AlberoBinario b, int libv) {
    if (a == null || b == null) return false;
    return (libv>liva && a.val() == b.val()) || verifica(a, liva, b.des(), libv++) || verifica(a, liva, b.sin(), libv++);
}
```

Nel caso migliore ho un sottoalbero della radice che verifica subito la condizione in un tempo costante ovvero visitando solo i primi nodi di a e i primi nodi di b. Quindi tempo costante.

Nel caso peggiore devo visitare tutto l'albero di a e tutto l'albero di b per ogni nodo di a quindi chiamate n la lunghezza di a e m la lunghezza di b definiscono i costo tramite la notazione asintotica O( $n*m$ ) in quanto potrebbe richiedere al massimo di visitare tutti i nodi di b per i nodi di a.

Nel caso spaziale ho sempre nel caso migliore un valore costante poiché alloco poco spazio per le poche chiamate.

Nel caso peggiore ho invece un albero degenero che richiede un costo Theta( $n*m$ )

12)

Si consideri una classe *AlberoBinario* che rappresenta *alberi binari* in cui la parte informativa di ogni nodo è un numero intero. Si assume che in tale classe siano implementati i seguenti metodi:

```
public interface AlberoBinario{
    /* restituisce il sottoalbero destro dell'albero corrente, la complessità temporale è O(1)*;
    public AlberoBinario destro();
    /* restituisce il sottoalbero sinistro dell'albero corrente, la complessità temporale è O(1)*;
    public AlberoBinario sinistro();
    /* restituisce il valore memorizzato nella radice dell'albero, la complessità temporale è O(1)*;
    public int val();
}
```

Si deve realizzare un metodo ricorsivo

```
public static boolean verificaAlmenoUnNodo(AlberoBinario a, int l) {
    che restituisce true se e solo se esiste in a al livello l almeno un nodo foglia n che è figlio unico e tale che n.val() è contenuto in almeno un altro nodo che si trova a un qualsiasi livello ll tale che ll != l.
```

Si caratterizza la complessità temporale e spaziale del metodo nel caso migliore e peggiore, specificando anche quali siano il caso migliore ed il caso peggiore per la complessità temporale e spaziale.

Caso Migliore:

Caso Peggior:

Si consideri una classe *AlberoBinario* che rappresenta *albert binari* in cui la parte informativa di ogni nodo è un numero intero. Si assume che in tale classe siano implementati i seguenti metodi:

```
public interface AlberoBinario {
    /* restituire il sottoalbero destro dell'albero corrente, la complessità temporale è Θ(l) */
    public AlberoBinario destro();
    /* restituire il sottoalbero sinistro dell'albero corrente, la complessità temporale è Θ(l) */
    public AlberoBinario sinistro();
    /* restituire il valore memorizzato nella radice dell'albero, la complessità temporale è Θ(1) */
    public int val();
}
```

Si deve realizzare un metodo ricorsivo

```
public static boolean verificaAlmenoUnNodo(AlberoBinario a, int l) { ... }
```

che restituisce true se e solo se esiste in al livello  $l$  almeno un nodo foglia  $n$  che è figlio unico e tale che  $n.val()$  è contenuto in almeno un altro nodo che si trova a un qualsiasi livello  $l'$  tale che  $l' \neq l$ .

Si caratterizzi la complessità temporale e spaziale del metodo nel caso migliore e peggiore, specificando anche quali siano il caso migliore ed il caso peggiore per la complessità temporale e spaziale.

Caso Migliore:  
 1. Complessità temporale:  $\Theta(\underline{\hspace{2cm}})$   
 2. Complessità spaziale:  $\Theta(\underline{\hspace{2cm}})$

Caso Peggior:  
 1. Complessità temporale:  $\Theta(\underline{\hspace{2cm}})$   
 2. Complessità spaziale:  $\Theta(\underline{\hspace{2cm}})$

In questo caso la traccia parla di un albero binario che dunque assumo che non sia né bilanciato né binario (e dunque sarà disordinato).

```
public static boolean verificaAlmenoUnNodo(AlberoBinario a, int l) {
    if (a == null) return ??;
    return verificaAlmenoUnNodo(a, l, 0);
}

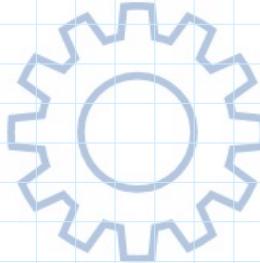
/*
 * Tale funzione verifica se un nodo è un figlio unico di una foglia. In tal caso
 * richiama la funzione che verifica la presenza o meno della proprietà.
 * Visto che basta un solo caso, allora restituisco direttamente il valore
 * della funzione
 */
public static boolean verificaAlmenoUnNodo(AlberoBinario a, int l, int lev) {
    if (a == null) return ??;
    AlberoBinario unico = fogliaUnica(a);
    if (unico != null)
        return ver(a, l, lev, unico.val());
    return verificaAlmenoUnNodo(a.des(), l, lev++) || verificaAlmenoUnNodo(a.sin(), l, lev++);
}

public static boolean ver(AlberoBinario a, int l, int lev, int val) {
    if (a == null) return false;
    if (a.val() == val && l != lev)
        return true;
    return ver(a.des(), l, lev++, val) || ver(a.sin(), l, lev++, val);
}

private static AlberoBinario fogliaUnica(AlberoBinario a) {
    if (a==null) return null;
    if (a.des() != null && a.sin() != null) return null;
    if (a.des() == null && a.des() == null) return null;
    if (a.des() != null) return a.des();
    return a.sin();
}
```

In questo problema, il caso migliore si ha quando con pochi passi costanti io verifico che l'albero a ha una foglia subito dopo la radice e ad esempio lo stesso valore di tale foglia si trova nella radice dell'albero b quindi avrà valore costante. Nel caso della complessità spaziale alloco poca memoria e dunque sarà sempre un costo costante.

Nel caso peggiore, visto che devo trovare almeno uno che corrisponda alla mia ricerca, dovrò esaminare tutti i nodi di  $a$  fino ad arrivare ad una foglia ed esaminare anche tutti i nodi di  $b$  fino ad esaminare una foglia. Non avendo infatti un albero di ricerca binaria (che avrebbe facilitato le cose permettendo una diminuzione di visite ai nodi) devo necessariamente considerare una notazione asintotica del tipo  $\Theta(n^*m)$  dove  $n$  sono il numero di nodi di  $a$  e  $m$  quelli di  $b$ . Posso infatti ipotizzare che il caso peggiore si abbia quando tutte le foglie di  $a$  hanno dato esito negativo e, giunta all'ultima foglia di  $a$ , ripeto per l'ennesima volta la visita a tutti i nodi di  $b$  per verificare tale confronto.



## EX

Dato un grafo orientato in cui i nodi sono attività e gli archi rappresentano relazioni di propedeuticità fra le attività. Ogni attività è caratterizzata da un tempo di esecuzione (memorizzato in un array) e si possono eseguire più attività contemporaneamente.

Progettare ed implementare un algoritmo che calcoli il minimo tempo di completamenti di tutte le attività che rispettano i tempi di esecuzione di ogni attività e le propedeuticità.

Metodo che calcola la somma di tutti i valori più grandi di  $x$  che appaiono nelle foglie dell'albero a

```
public static int sumGreaterThan(AlberoBinario a, int x) {
    if (a==null) return 0;
    if (foglia(a) && a.val() > x)
        return 1;
    return sumGreaterThan(a.des(), x) + sumGreaterThan(a.sin(), x);
}

private static boolean foglia(AlberoBinario a) {
    if (a==null) return false;
    if (a.des() == null && a.sin() == null) return true;
    return false;
}
```

In questo problema devo attraversare tutto l'albero e visitare ciascun nodo almeno una volta prima di poter restituire una somma effettiva. Dunque il costo, sia nel caso migliore che peggiore sarà  $\Theta(n)$ .

Nel caso spaziale sarà, se è un AVL, pari a  $\log(n)$  altrimenti  $n$  nel caso degenere  $\Theta(n)$

Metodo che restituisce true solo se l'albero contiene almeno un nodo foglia con valore negativo e i cui predecessori hanno tutti valori positivi

```
public static boolean verifica(AlberoBinario a) {
    if (a == null) return false;
    if ((a.des() != null || a.sin() != null) && a.val() > 0)
        return verifica(a.des()) || verifica(a.sin());
    if (a.des() == null && a.sin() == null)
        return a.val() < 0;
    return false;
}
```

Tale algoritmo prevede di analizzare tutti quei percorsi fermanosi però quando lungo uno di essi si incrocia un nodo non positivo. In tal caso non ha senso proseguire oltre fino alla foglia perché si sa già che il metodo restituirà false.

Dunque nel caso migliore ho che la radice o i figli della radice sono negativi; ciò causa l'interruzione istantanea del metodo che quindi avrà un costo migliore che è costante ovvero Theta(1).

Nel caso peggiore invece dovrà analizzare tutta la lunghezza dell'albero che, nel caso di un albero degenere, può essere proprio pari ad n. Di conseguenza, se ho un albero di questo tipo con una sola foglia e un percorso tutto positivo allora otengo che il costo sarà Theta(n).

Nel caso spaziale avrò costo temporale migliore costante mentre quello peggiore sarà pari a Theta(n) nel caso dell'albero degenere.

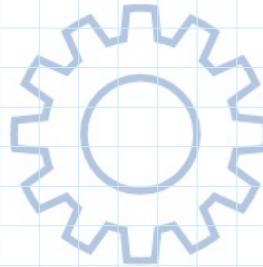
Metodo che restituisce true se tutti i nodi foglia di a contengono un valore che non appare in nessun altro nodo di a

Dunque tale metodo prevede di verificare tutti i nodi foglia (e quindi di attraversare tutto l'albero) per poi ricontrollare se esiste un altro nodo, diverso da quello in cui ci si trova, che non presenta lo stesso valore. Si può verificare ciò utilizzando un contatore per ciascun nodo che conti le occorrenze di ciascun valore (tecnica che implementerò)

```
public static boolean verificaUnicità(AlberoBin<Integer> a){
    if (a == null) return true;
    return verificaUnicità(a,a);
}

private static boolean verificaUnicità(AlberoBin<Integer> a, AlberoBin<Integer> b) {
    if (a == null) return true;
    if (a.des() == null && a.sin() == null)
        return !ocorre(b, a.val()) == 1;
    return verificaUnicità(a.des(), b) && verificaUnicità(a.sin(), b);
}

private static int occorre(AlberoBin<Integer> a, int val) {
    if (a == null) return 0;
    if (a.val().equals(val))
        return 1;
    return occorre(a.des(), val) + occorre(a.sin(), val);
}
```



Nel caso migliore ho un ramo che ha un solo nodo (dunque una foglia) che è presente pure nella radice dell'albero o nella radice del sottoalbero destro. Di conseguenza con un costo lineare ottengo il caso migliore. Stessa cosa vale per la complessità spaziale visto che alloco poco spazio in memoria.

Nel caso peggiore, se ho un albero degenere devo attraversare tutti i nodi dell'albero e, per verificare che non esistano copie di tale nodo, devo riattraversare l'albero per scoprire che non ci sono nodi.

Quindi n+n ovvero Theta(2n). Per la spaziale avrà una complessità sempre lineare

## DI INGEGNERIA INFORMATICA GAIA BERTOLINO

metodo che calcola quanti nodi foglia sono uguali a x

```
public static int contaFoglie(AlberoBinario a, int x) {
    if (a==null) return 0;
    if (a.des() == null && a.sin() == null)
        if (a.val() == x)
            return 1;
    return contaFoglie(a.des(), x) + contaFoglie(a.sin(), x);
}
```

Metodo che calcola il numero di nodi interni con valore maggiore di x e con entrambi i figli

```
public static int countGreaterThan(AlberoBin<Integer> a, int x) {
    if (a==null) return 0;
    if (a.des() != null && a.sin() != null && a.val() == x)
        return 1 + countGreaterThan(a.des(), x) + countGreaterThan(a.sin(), x);
    return countGreaterThan(a.des(), x) + countGreaterThan(a.sin(), x);
}
```

Tale algoritmo in qualunque caso deve analizzare tutti i nodi. Dunque sia nel caso migliore che peggiore avrà valore Theta(n). L'unico caso in cui tale condizione può migliorare è se si ha un AVL perché in quel caso si potrebbe scrivere un algoritmo ad hoc che va a ricercare proprio la metà in cui si trova il valore x. per quanto riguarda la complessità spaziale essa sarà appunto sempre Theta(n), tranne nel caso di un AVL per cui sarebbe Theta(logn)

Dato un albero a, scrivere un metodo countLeavesEqualTo(Albero Binario a, int x) che restituisce il numero di foglie di a che hanno valore x

Metodo che dato un albero binario restituisce true se tutti i nodi foglia sono maggiori di un valore k

```

public static boolean verifica(AlberoBin<Integer> a, int x) {
    if (a == null) return true;
    if (a.des() == null && a.sin() == null)
        if (a.val()<x)
            return false;
        else
            return true;
    return verifica(a.des(), x) && verifica(a.sin(), x);
}

```

In questo caso il caso banale si ha quando una foglia è il figlio della radice e subito non verifica la proprietà. Quindi costo costante. Caso peggiore Theta(n). Stessa cosa per la spaziale

Metodo che calcola il cammino di costo massimo che si ha attraversando dalla radice alla foglia

```

public static int camminoMassimo(AlberoBinario a) {
    if (a == null) return 0;
    if (a.des() != null && a.sin() != null)
        if (a.des().val() > a.sin().val())
            return a.des().val() + camminoMassimo(a.des());
        else
            return a.sin().val() + camminoMassimo(a.sin());
    if (a.des() != null)
        return a.des().val() + camminoMassimo(a.des());
    return a.sin().val() + camminoMassimo(a.sin());
}

public static int camminoMassimo(AlberoBinario a) {
    if (a == null) return 0;
    return a.val() + Massimo(camminoMassimo(a.des()), camminoMassimo(a.sin()));
}

```

In questo caso ho che la complessità di tale algoritmo è data dalla necessità di attraversare per la sua lunghezza tutto l'albero. Visto che devo verificare tutti i nodi almeno una volta il costo migliore e peggiore sarà lineare. Stessa cosa nel caso della complessità spaziale tranne nel caso in cui ho un AVL perché si riduce a  $\log(n)$

Metodo che restituisce i valori dell'albero che sono ripetuti almeno due volte

```

public static int contaRipetuti (AlberoBinario a) {
    if (a == null) return 0;
    return contaRipetuti(a, a);
}

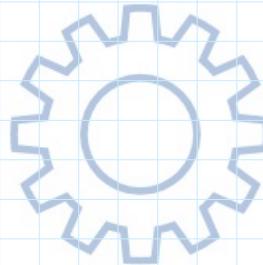
private static int contaRipetuti(AlberoBinario a, int val, AlberoBinario origin) {
    if (a==null) return null;
    if (conta(a.val()), origin) >= 2)
        return 1 + contaRipetuti(a.des(), origin) + contaRipetuti(a.sin(), origin);
    return contaRipetuti(a.des(), origin) + contaRipetuti(a.sin(), origin);
}

private static int conta(int val, AlberoBinario origin) {
    if (a==null) return 0;
    if (origin.val() == val)
        return 1 + conta(val, a.des()) + conta(val, a.sin());
    return conta(val, a.des()) + conta(val, a.sin());
}

```

Tale algoritmo attraversa una volta tutti i nodi e per ogni nodo riatraversa nuovamente tutta la struttura. Ciò vuol dire che il costo nel caso migliore e peggiore sarà sempre quadratico. Nel caso spaziale avremo nel caso migliore un AVL e dunque la ricerca sarà proporzionale ad un logaritmico mentre invece quella peggiore sarà proporzionale alla lineare.

RICORDA-> nel caso dell'attraversamento di un albero, nel caso migliore lo si può considerare come un AVL mentre nel caso peggiore come un albero degenero.  
Nel caso di più visite, la complessità spaziale si somma e non si moltiplica come la temporale in quanto sta ad indicare il numero delle chiamate di metodi contemporaneamente attive sullo stack



## APPUNTI DI INGEGNERIA FORMATICA

GAIA BERTOLINO

Metodo che dati due alberi binari, restituisce true se almeno un nodo di a ad un certo livello liva compare in b in un livello minore di b

```

public static boolean livelloMaggiore(AlberoBinario a, AlberoBinario b) {
    if (a==null) return false;
    return livelloMaggiore(a, 0, b, 0);
}

private static boolean livelloMaggiore(AlberoBinario a, int liva, AlberoBinario b, int livb) {
    if (a==null || b== null) return false;
    if (a.val() == b.val() && liva<livb)
        return true;
    return livelloMaggiore(a, liva, b.des(), livb++) || livelloMaggiore(a, liva, b.sin(), livb++) ||
    livelloMaggiore(a.des(), liva++, b, livb) || livelloMaggiore(a.sin(), liva++, b, livb);
}

```

Tale metodo ha come caso migliore quanto dopo pochi nodi posso verificare che ho ottenuto un nodo che rispetta la proprietà quindi Theta(1). Nel caso peggiore devo attraversare tutti i nodi di a e tutti i nodi di b per ciascuna visita in a e b e dunque sarà pari a  $\Theta(n*m)$  dove chiamo m il numero di nodi di b

Per quanto riguarda la complessità spaziale vale lo stesso ragionamento: Theta(1) nel caso migliore e  $\Theta(n+m)$  nel caso peggiore

Metodo che restituisce true se esiste almeno un cammino dalla radice alla foglia la cui media dei valori sia maggiore di un valore x passato come argomento

```

public static boolean camminoMedio(AlberoBin<Integer> a, int x) {
    if (a==null) return false;
    return camminoMedio(a, 0, 0)>x;
}

```

```

}

private static int camminoMedio(AlberoBin<Integer> a, int nodi, int somma){
    if (a==null) return (somma/nodi);
    somma+= a.val();
    nodi++;
    return Math.max(camminoMedio(a.des(), nodi, somma), camminoMedio(a.sin(), nodi, somma));
}

```

In questo caso otengo il caso migliore se verifico subito la condizione col primo nodo a sinistra (che è una foglia) e dunque costo migliore temporale e spaziale Theta(1). Nel caso peggiore è Theta(n).

si deve realizzare un metodo

```

public static boolean dueNodiLivello(AlberoBinario a, int l) {..}
che riceve un albero binario di interi e restituisce true se
esistono due nodi diversi n ed m al livello l tali che n.val()==m.val()

public static boolean dueNodiLivello(AlberoBin<Integer> a, int l) {
    if (a==null) return false;
    return dueNodiAlbero(a, l, 0, a);
}

private static boolean dueNodiAlbero(AlberoBin<Integer> a, int l, int lev,
AlberoBin<Integer> origin) {
    if (a==null) return false;
    if (lev==l) return occorrenze(origin, 0, a.val(), lev)>1;
    return dueNodiAlbero(a.des(), l, lev++, origin) ||
    dueNodiAlbero(a.sin(), l, lev++, origin);
}

private static int occorrenze (AlberoBin<Integer> a, int s, int val, int lev) {
    if (a==null) return 0;
    System.out.println("numero " + a.val() + " livello " + s);
    if (s==lev && a.val().equals(val))
        return 1;
    return occorrenze(a.des(), s+1, val, lev) + occorrenze(a.sin(), s+1, val,
    lev);
}

```

Metodo che restituisce true se la foglia che si trova a minore profondità nell'albero si trova ad un livello minore di l passato come argomento

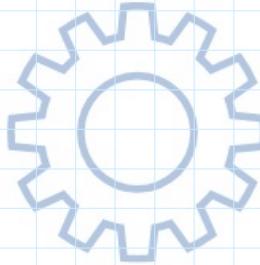
```

public static boolean verifica(AlberoBin<Integer> a, int l) {
    if (a==null) return false;
    return livello(a, 0)<l;
}

private static int livello(AlberoBin<Integer> a, int liv) {
    if (a==null) return 0;
    if (a.des() == null && a.sin() == null) return liv;
    return Math.min(livello(a.des(), liv+1), livello(a.sin(), liv+1));
}

```

In questo caso, dovrò per forza visitare tutti i nodi foglia. Dunque Theta(n).  
Nel caso spaziale migliore sarà Theta(log(n)) nel caso migliore e Theta(n) in quello peggiore



## APPUNTI DI INGEGNERIA

## INFORMATICA

GAIA BERTOLINO

# Appello 14 Luglio 2014

mercoledì 14 luglio 2021 16:35



AppelloASD  
14Luglio2...

Cognome: \_\_\_\_\_ Nome: \_\_\_\_\_ Matricola: \_\_\_\_\_

UNIVERSITÀ DEGLI STUDI DELLA CALABRIA  
Corso di Laurea in Ingegneria Informatica

Prova scritta di *Algoritmi e Strutture Dati*  
(durata della prova: 60 minuti)

Traccia A

## Esercizio 1

Si consideri una classe *AlberoBinario* che rappresenta *alberi binari* in cui la parte informativa di ogni nodo è un numero intero. Si assuma che in tale classe siano implementati i seguenti metodi:

```
public interface AlberoBinario {  
    /* restituisce il sottoalbero destro dell'albero corrente, la complessità temporale è O(1) */  
    public AlberoBinario destro();  
    /* restituisce il sottoalbero sinistro dell'albero corrente, la complessità temporale è O(1) */  
    public AlberoBinario sinistro();  
    /* restituisce il valore memorizzato nella radice dell'albero, la complessità temporale è O(1) */  
    public int val();  
}
```

Si realizza un metodo ricorsivo

```
public static boolean verificaDueNodi(AlberoBinario a) {...}  
che restituisce true se e solo se esistono almeno due nodi dell'albero che soddisfano la proprietà A  
specificata di seguito. Un nodo soddisfa la proprietà A se la somma tra il valore contenuto in esso  
ed il livello del livello in cui si trova è minore di zero. Si caratterizza la complessità temporale e  
spaziale del metodo nel caso migliore e peggiore, specificando anche quali siano il caso migliore ed  
il caso peggiore per la complessità temporale e spaziale.
```

### Caso Migliore:

1. Compl. temporale:  $O(1)$
2. Compl. spaziale:  $O(1)$

### Caso Peggio:

1. Compl. temporale:  $O(n)$
2. Compl. spaziale:  $O(n)$

Commenti: \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

Il metodo restituisce true se due nodi rispettano tale proprietà:  
la somma fra il livello del nodo e il suo contenuto è minore di zero

Si hanno i seguenti casi banali:

- 1) il nodo è null -> restituisce false
- 2) il nodo non è null -> se  $nodo.val() + livello < 0$

```
public static boolean verificaDueNodi(AlberoBinario a) {  
    if (a == null) return false;  
    return verificaDueNodi(a, 0, 1);  
}
```

```
private static boolean verificaDueNodi(AlberoBinario a, int cont, int liv) {  
    if (a == null) return false;  
    if (a.val() + liv < 0) cont++;  
    if (cont >= 2) return true;  
    return verificaDueNodi(a.des(), cont, liv++) || verificaDueNodi(a.sin(), cont, liv++);  
}
```

**INGEGNERIA  
SEMPLIFICANDO OTTERRO'**

```
public static boolean verificaDueNodi(AlberoBinario a) {  
    if (a == null) return false;  
    return verificaDueNodi(a, 0, 0);  
}
```

```
private static boolean verificaDueNodi(AlberoBinario a, int cont, int liv) {  
    if (a == null) return false;  
    if (a.val() + liv < 0) cont++;  
    return cont >= 2 || verificaDueNodi(a.des(), cont, liv++) || verificaDueNodi(a.sin(), cont, liv++);  
}
```

- 1) VERO solo nel caso in cui si intende minore o uguale di 1 in valore assoluto altrimenti si potrebbe avere un grado di sbilanciamento -2 che rispetta i canoni ma è appunto sbilanciato
- 2) FALSO Nel caso di visita anticipata si visita prima la radice e poi il resto dell'albero. il costo della visita sarà dunque necessariamente pari a  $\Theta(n)$
- 3) VERO perché la notazione omega indica una funzione minorante
- 4) VERA se il grafo è connesso altrimenti no
- 5) FALSO nel caso migliore l'inserimento avviene subito e direttamente alla sua cella quindi è  $\Theta(1)$
- 6) FALSO perché è  $\Theta(n)$
- 7) FALSO perché si possono avere più alberi minimi ricoprenti
- 8) FALSO perché essendo binario si può applicare un'analisi per altezza
- 9) VERO se si parla di un albero binario crescente
- 10) VERO perché un grafo può essere ciclico o meno anche se il nodo alla radice ha grado di entrata zero

## Esercizio 2

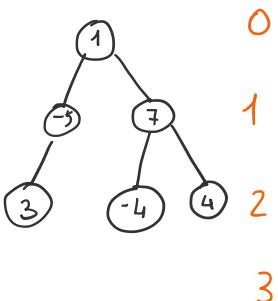
Per ognuna delle seguenti affermazioni, indicare se è vera o falsa.

V	F	Affermazione
1		Un albero binario è bilanciato se la differenza fra l'altezza del sottoalbero sinistro della radice e l'altezza del sottoalbero destro della radice è minore o uguale ad 1.
2		La complessità spaziale della visita anticipata di un albero binario con $n$ nodi è $O(lg n)$ nel caso peggiore.
3		La funzione $f(n) = 2n^2$ è $\Omega(n)$ .
4		Sia $G$ un grafo non orientato ed aciclico. $G$ è un albero.
5		L'inserimento di un elemento in una hash table ha complessità $O(n)$ nel caso migliore.
6		La complessità temporale della visita per livelli di un albero binario è $O(n^2)$ .
7		Un grafo non orientato connesso e pesato (sugli archi) ammette sempre un unico albero ricoprente di costo minimo.
8		L'inserimento di un elemento in un heap binario ha complessità temporale $O(n)$ nel caso peggiore (dove $n$ è il numero dei nodi).
9		Un albero binario è detto di ricerca se, per ognuno dei suoi nodi $u$ , la radice del figlio sinistro di $u$ contiene un valore minore di quello contenuto in $u$ e la radice del figlio destro di $u$ contiene un valore maggiore o uguale di quello contenuto in $u$ .
10		Un grafo connesso ed orientato in cui esiste almeno un nodo con grado di entrata uguale a 0 può contenere un ciclo.

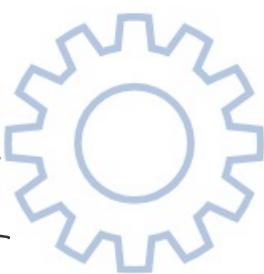
## Esercizio 3

Si descrivono le caratteristiche di un Heap binario e la procedura di estrazione della radice da esso.

— Un heap binario è un albero in cui vige la proprietà per cui ogni padre risulta avere una relazione gerarchica nei confronti dei figli. Si parla dunque di minHeap nel caso in cui esso sia minore mentre di max heap nel caso in cui sia maggiore.  
— Per estrarre la radice si intende l'estrazione del maggiore dall'heap.  
— Una volta rimossa la radice si può procedere ponendo come nuova radice uno dei due figli e procedendo in questo modo nei sottoalberi in modo da ribilanciare l'albero.



$a$	$l_{iv}$	$a+l_{iv}$	$cout$
1	0	1	0
7	1	8	0
-5	1	-4	1
4	2	6	1
-6	2	-2	2



APPUNTI DI INGEGNERIA  
INFORMATICA

GAIA BERTOLINO

# Appello 18 febbraio 2013

giovedì 15 luglio 2021 17:18



17:20

AppelloASD  
18febbrai...

Cognome: \_\_\_\_\_ Nome: \_\_\_\_\_ Matricola: \_\_\_\_\_

## UNIVERSITÀ DEGLI STUDI DELLA CALABRIA Corso di Laurea in Ingegneria Informatica

### Prova scritta di *Algoritmi e Strutture Dati* – TRACCIA A (durata della prova: 60 minuti)

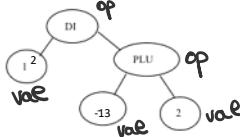
#### Esercizio 1

Si consideri una classe *AlberoBinario* che rappresenta *alberi binari* che rappresentano *espressioni*. Si assuma che in tale classe siano implementati i seguenti metodi:

```
public enum Operation {VALUE, PLUS, MINUS, MUL, DIV}  
  
public interface AlberoBinario {  
    /* restituisce il sottoalbero destro dell'albero corrente, la complessità temporale è Θ(1)*/  
    public AlberoBinario destro();  
  
    /* restituisce il sottoalbero sinistro dell'albero corrente, la complessità temporale è Θ(1)*/  
    public AlberoBinario sinistro();  
  
    /* Il metodo restituisce il valore memorizzato nella radice dell'albero. Se l'albero è una foglia il valore ha un  
    significato altrimenti va ignorato. La complessità temporale è Θ(1)*/  
    public int val();  
  
    /* Il metodo restituisce l'operazione caratteristica del nodo radice. Se l'albero non è una foglia il valore è  
    diverso da VALUE altrimenti è uguale a VALUE. La complessità temporale è Θ(1)*/  
    public Operation op();  
}
```

Si deve realizzare un metodo  
public static int eval(AlberoBinario a) {...}  
che calcola il valore dell'espressione descritta dall'albero. Si assume che l'albero rappresenta correttamente un'espressione.

Ad esempio l'espressione  $(12 / ((-13)+(21)))$  è rappresentata dall'albero



Si caratterizza la complessità temporale e spaziale del metodo nel caso migliore e peggiore, specificando anche quali siano il caso migliore ed il caso peggiore per la complessità temporale e spaziale.

Caso Migliore:

1. Complessità temporale:  $\Theta(\underline{\hspace{1cm}})$
2. Complessità spaziale:  $\Theta(\underline{\hspace{1cm}})$

Caso Peggio:

1. Complessità temporale:  $\Theta(\underline{\hspace{1cm}})$
2. Complessità spaziale:  $\Theta(\underline{\hspace{1cm}})$

Commenti: \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

- 1) FALSO -> ha costo n
- 2) FALSO ->  $n(\log n)$  è maggiore di  $n^2$
- 3) VERO
- 4) VERO -> Floyd si applica ai grafi orientati con pesi di tutti i tipi
- 5) FALSO -> posso sempre avere un nodo (quello di partenza ad esempio) che ha grado di entrata 0
- 6) VERO
- 7) FALSO -> ha complessità dettata sia dal range di numeri possibili che dal valore di input
- 8) VERO
- 9) FALSO -> un albero binario è completo non vi è differenza fra sottoalbero destro e sinistro
- 10) FALSO

#### Esercizio 2

Dire quali delle seguenti affermazioni sono vere e quali false.

N°	Vero	Falso	Affermazione
1			L'inserimento di un valore in una tabella hash contenente n valori ha complessità temporale pari a $\Theta(\lg n)$ nel caso peggiore
2			La funzione $f(n) = n^2$ è $\Omega(n \lg(n))$
3			L'algoritmo mergeSort ha complessità temporale $O(n \lg(n))$ nel caso peggiore.
4			L'algoritmo di Floyd calcola, dati un grafo orientato e pesato e un nodo x di tale grafo, le distanze minime fra x e i nodi da esso raggiungibili.
5			Sia G un grafo orientato. Se G contiene cicli allora non vi è nessun nodo in G con grado di entrata zero
6			In un grafo non orientato connesso con n nodi e n-1 archi il numero di componenti connesse (non vuote) è uguale a n.
7			L'algoritmo countingSort ha complessità $\Theta(n^2)$ nel caso peggiore
8			Nel caso peggiore, rimuovere un elemento da un albero AVL con n nodi ha complessità $O(\log n)$
9			Un albero binario a è completo, se e solo se per tutti i nodi x di a, la differenza fra l'altezza del sottoalbero sinistro di x e l'altezza del sottoalbero destro di x è minore o uguale ad 1.
10			La complessità intrinseca del problema della ricerca del secondo minimo in una sequenza disordinata è $\Theta(\lg(n))$ .

#### Esercizio 3

Si assume di avere un algoritmo Divide et Impera che divide l'istanza problema originario (di dimensione n) in 2

In questo algoritmo ad ogni nodo è associata una operazione o un valore. Le foglie hanno valori mentre i nodi interni hanno dei valori. La radice da sola, se non è null, presenta un val.ore

Il metodo deve valutare l'espressione descritta dall'albero ricordando che nel caso generale che

```
private static int calcolo(AlberoBinario a) {  
    if (a==null) return 0;  
    if (a.des()!= null && a.sin()!= null) {  
        if (a.des().des() == null && a.des().sin() == null)  
            int op1 = a.val();  
        else  
            op1 = calcolo(a, ris);  
        if (a.sin().des() == null && a.sin().sin() == null)  
            int op2 = a.val();  
        else  
            op2 = calcolo(a, ris);  
        return operazione(a.op(), op1, op2);  
    }  
    return a.val();  
}  
  
private static int operazione(Operazione p, int a, int b) {  
    if (p == PLUS)  
        return a+b;  
    if (p==MINUS)  
        return a-b;  
    if (p==MUL)  
        return a*b;  
    if(p==DIV)  
        return a/b;  
}
```

Sia nel caso migliore che peggiore il costo è lineare ovvero Theta(n) perché devo sempre analizzare tutto l'albero e non posso fermarmi prima. Nel caso della complessità spaziale, vale la stessa regola in quanto nel caso migliore devo comunque chiamare la funzione su tutti i nodi mentre nel caso peggiore degenera in una lista.

		dell sottoalbero sinistro di $x$ e l'altezza del sottoalbero destro di $x$ è minore o uguale ad 1.
10		La complessità intrinseca del problema della ricerca del secondo minimo in una sequenza disordinata è $\Omega(\lg(n))$

**Esercizio 3**

Si assume di avere un algoritmo Divide et Impera che divide l'istanza problema originario (di dimensione  $n$ ) in 2 istanze la cui dimensione è  $n - k$ , dove  $k$  è una costante. La fase di suddivisione del problema in sottoproblemi e di costruzione della soluzione dell'istanza originaria a partire dalle soluzioni delle istanze in cui essa è decomposta costa  $b*n$ . Calcolare la complessità dell'algoritmo?

Posso definire i seguenti parametri:

$a = 2$  ovvero sottoistanze del problema

$n-k$  = dimensione di ciascuna istanza

$d = 1$  che è il costo asintotico della suddivisione

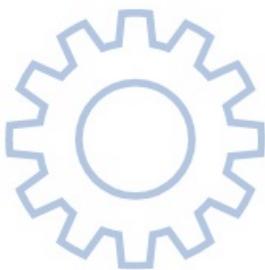
Posso allora utilizzare la seguente formula del teorema della ricorrenza per cui ho che se

$a=1$  allora la funzione sarà asintotica a  $O(n^{d+1})$

$a \geq 2$  allora la funzione sarà asintotica a  $O(a^n + n^d)$

In questo caso vale la seconda relazione per cui avrà che  
la complessità di tale algoritmo sarà

$O(2^n + n)$



# APPUNTI DI INGEGNERIA INFORMATICA

GAIA BERTOLINO

# PROVA 1

mercoledì 14 luglio 2021 22:40

Cognome: \_\_\_\_\_ Nome: \_\_\_\_\_ Matriqua: \_\_\_\_\_

UNIVERSITÀ DEGLI STUDI DELLA CALABRIA  
Corso di Laurea in Ingegneria Informatica

Prova scritta di *Algoritmi e Strutture Dati* – TRACCIA A  
(durata della prova: 60 minuti)

**Esercizio 1**

Si consideri una classe *AlberoBinario* che rappresenta alberi binari in cui la parte informativa di ogni nodo è un numero intero. Si assume che in tale classe siano implementati i seguenti metodi:

```
public interface AlberoBinario {
    /* restituisce il sottoalbero destro dell'albero corrente, la complessità temporale è O(1) */
    public AlberoBinario destro();
    /* restituisce il sottoalbero sinistro dell'albero corrente, la complessità temporale è O(1) */
    public AlberoBinario sinistro();
    /* restituisce il valore memorizzato nella radice dell'albero, la complessità temporale è O(1) */
    public int val();
}
```

Si deve realizzare un metodo

```
public static boolean verifica(AlberoBinario a) { ... }
```

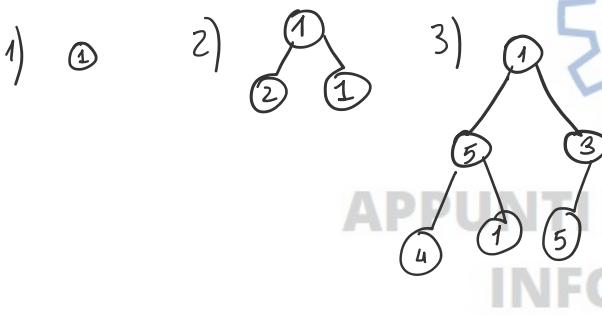
che restituisce true se e solo se tutti i nodi non foglia di *a* contengono un valore che appare in almeno un nodo foglia di *a*.

Si caratterizzi la complessità temporale e spaziale del metodo nel caso migliore e peggiore, specificando anche quali siano il caso migliore ed il caso peggiore per la complessità temporale e spaziale.

Caso Migliore: 1. Complessità temporale:  $O(\underline{\hspace{1cm}})$  2. Complessità spaziale:  $O(\underline{\hspace{1cm}})$

Caso Peggio: 1. Complessità temporale:  $O(\underline{\hspace{1cm}})$  2. Complessità spaziale:  $O(\underline{\hspace{1cm}})$

Commenti: *[scrivere qui]*



La funzione deve verificare se tutti i nodi non foglia contengono un valore che è presente in un nodo foglia.

Di conseguenza devo verificare per tutti i nodi interni ciascun nodo foglia ma per raggiungere tali nodi necessito di riesaminare tutti i nodi

Il primo caso banale è la radice nulla.

Per operare tale verifica devo avere un metodo che, richiamato su un nodo

```
ArrayList<Integer> foglie = new ArrayList<>();
private static void calcolaFoglie(AlberoBin<Integer> a) {
    if (a == null) return;
    if (a.des() == null && a.sin() == null)
        foglie.add(a.val());
    calcolaFoglie(a.des());
    calcolaFoglie(a.sin());
}

public static boolean verifica(AlberoBin<Integer> a) {
    calcolaFoglie(a);
    return verifica(a);
}

public static boolean verifica(AlberoBin<Integer> a) {
    if (a == null || (a.des() == null && a.sin() == null)) return true;
    //verifico se tale valore non è una foglia
    if (a.des() != null || a.sin() != null)
        if (foglie.contains(a.val()))
            return true && verifica(a.des()) && verifica(a.sin());
        else return false;
    return verifica(a.des()) && verifica(a.sin());
}
```

## VERSIONE RIDOTTA

```
private static void calcolaFoglie(AlberoBin<Integer> a) {
    if (a == null) return;
    if (a.des() == null && a.sin() == null)
        foglie.add(a.val());
    calcolaFoglie(a.des());
    calcolaFoglie(a.sin());
}

public static boolean verificaNEW(AlberoBin<Integer> a) {
    if (a == null) return true;
    if (a.des() != null || a.sin() != null)
        return foglie.contains(a.val()) && verificaNEW(a.des()) && verificaNEW(a.sin());
    return verificaNEW(a.des()) && verificaNEW(a.sin());
}
```

GAIA BERTOLINO

## Esercizio 2

Dire quali delle seguenti affermazioni sono vere e quali false.

N°	Vero	Falso	Affermazione
1			L'inserimento di un valore in un heap contenente n valori ha complessità temporale pari a $\Theta(n \lg n^2)$
2			La funzione $f(n) = n^2$ è $\Omega(n \lg n^2)$
3			L'algoritmo quickSort ha complessità temporale $O(n \lg n)$ nel caso peggiore.
4			L'algoritmo di Prim calcola, dati un grafo orientato e pesato e un nodo <i>x</i> di tale grafo, le distanze minime fra <i>x</i> e i nodi da esso raggiungibili.
5			Sia <i>G</i> un grafo non orientato che contiene cicli. Il numero di archi di <i>G</i> è maggiore o uguale al numero di nodi di <i>G</i> .
6			In un grafo orientato non contenente cicli, esiste almeno un nodo con grado di entrata uguale a 0.
7			L'inserimento di un valore in un insieme rappresentato tramite array di boolean ha complessità $\Theta(1)$ .
8			Nei casi peggiori, inserire un elemento in un albero AVL con <i>n</i> nodi ha complessità $O(n \log n)$ .
9			Un albero binario è completo, se e solo se per tutti i nodi <i>x</i> di <i>a</i> , la differenza fra l'altezza del sottoalbero sinistro di <i>x</i> e l'altezza del sottoalbero destro di <i>x</i> è uguale ad 0.
10			Un grafo non orientato连通 e pesato (sugli archi) ammette sempre almeno due alberi ricoprenti.

## Esercizio 3

Dando per noto il concetto di albero binario, si definisca formalmente il concetto di *albero binario di ricerca bilanciato*.

Un albero binario di ricerca è un albero ordinato in cui alla sinistra della radice si trovano tutti i valori minori di essa (prendendo come esempio un albero crescente) e alla sua destra tutti i valori maggiori di essa.

Nel caso di un albero binario di ricerca (detto anche AVL) si ha che la differenza in altezza fra il sottoalbero destro e quello sinistro non è maggiore in valore assoluto di 1 ovvero è al massimo -1, 0 o 1. Per valori diversi l'albero è detto sbilanciato. Seguono esempi. Gli alberi binari sono molto utili nel caso della ricerca binaria che, nel caso di un AVL vale  $\Theta(\lg n)$  in entrambi i casi (ricordando che l'altezza di un albero binario è proprio  $\lg n$ ), mentre per un albero non bilanciato nel caso peggiore diventa lineare ovvero  $\Theta(n)$ .

## VERSIONE ULTERIORE

```
public static boolean verifica(AlberoBin<Integer> a) {
    if (a==null) return false;
    return verifica(a,a);
}

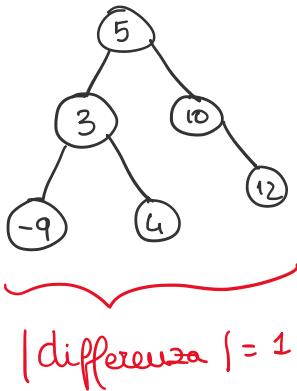
public static boolean verifica(AlberoBin<Integer> a, AlberoBin<Integer> origin) {
    if (a==null) return true;
    if (!foglia(a)) return verifica(origin, a.val()) && verifica(a.des(),origin) && verifica(a.sin(),origin);
    return true;
}

public static boolean verifica(AlberoBin<Integer> a, int val) {
    if (a==null) return false;
    if (foglia(a)) return a.val()==val;
    return verifica(a.des(), val) || verifica(a.sin(), val);
}

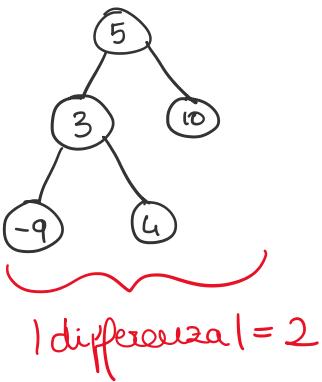
private static boolean foglia(AlberoBin<Integer> a) {
    if (a==null) return false;
    if (a.des() == null && a.sin() == null) return true;
    return false;
}

1) FALSO -> L'inserimento in un heap richiede tempo n se non è binario, altrimenti log(n) se esso è binario in quanto si riprende il concetto di altezza dell'albero
2) FALSO -> la funzione ha costo  $\Theta(n^2)$ 
3) falso -> L'algoritmo quicksort ha costo medio pari a  $\Theta(n \lg n)$  mentre ha costo peggiore quadratico quando l'elemento pivotale scelto è il maggiore o il minore
4) FALSO -> l'algoritmo di Prim si applica solo a grafi non orientati e con pesi positivi come l'algoritmo di Kruskal. Poi ho Floyd che posso applicare a grafi orientati e con pesi qualsiasi mentre Dijkstra a grafi orientati e non con pesi positivi.
```

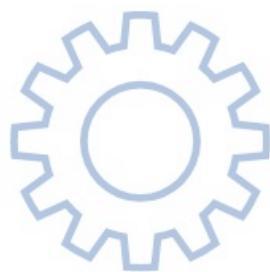
BILANCIATO



SBIANCIATO



- 5) VERO
- 6) VERO -> ad esempio si può pensare ad un nodo "radice" che non archi entranti
- 7) ???
- 8) FALSO -> ha costo sempre n se non è bilanciato
- 9) VERO -> un albero si dice completo se tutti i suoi livelli sono pieni e tutti nodi tranne le foglie hanno due figli. Si dice quasi completo se è completo fatta eccezione per l'ultimo livello
- 10) FALSO



## APPUNTI DI INGEGNERIA INFORMATICA

GAIA BERTOLINO

# PROVA 2

giovedì 15 luglio 2021 14:44

## Esercizio 2

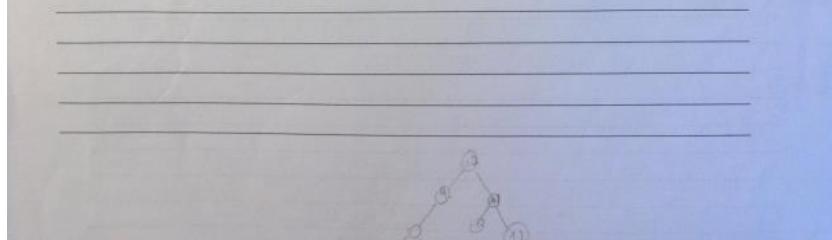
Per ognuna delle seguenti affermazioni, indicare se è vera o falsa.

	V	F	Affermazione
1	X		Un albero binario è bilanciato se la differenza fra l'altezza del sottoalbero sinistro della radice e l'altezza del sottoalbero destro della radice è minore o uguale ad 1. $\rightarrow \Delta \leq 1$ ?
2	X		La complessità spaziale della visita anticipata di un albero binario con $n$ nodi è $O(\lg n)$ nel caso peggiore. $O(n)$
3	X		La funzione $f(n) = 2n^2$ è $\Omega(n)$ . $\Omega(n^2)$
4	X		Sia $G$ un grafo non orientato ed aciclico. $G$ è un albero. <del>esse essere connesso</del>
5	X		L'inserimento di un elemento in una hash table ha complessità $O(n)$ nel caso migliore. $O(1)$
6	<		La complessità temporale della visita per livelli di un albero binario è $O(n^2)$ . $O(n)$
7	X		Un grafo non orientato connesso e pesato (sugli archi) ammette sempre un unico albero ricoprente di costo minimo.
8	X		L'inserimento di un elemento in un heap binario ha complessità temporale $\Theta(n)$ nel caso peggiore (dove $n$ è il numero dei nodi). $O(\lg n)$
9	X		Un albero binario è detto di ricerca se, per ognuno dei suoi nodi $u$ , la radice del figlio sinistro di $u$ contiene un valore minore di quello contenuto in $u$ e la radice del figlio destro di $u$ contiene un valore maggiore o uguale di quello contenuto in $u$ .
10	X		Un grafo connesso ed orientato in cui esiste almeno un nodo con grado di entrata uguale a 0 può contenere un ciclo.

## Esercizio 3

Si descrivano le caratteristiche di un Heap binario e la procedura di estrazione della radice da esso.

Un heap binario ad un livello  $l$  di albero binario è un albero binario radicato con le seguenti proprietà: 1) siamo alla radice  $=$  con solo albero fino all'albero binario; 2) ciascunno indennamente gli elementi di  $l$  sono memorizzati nei nodi dell'albero; ogni nodo si memorizza uno ed un solo elemento, che dividiamo con  $\Theta(n)$ , e quel elemento è memorizzato in un solo nodo. 3) ordinamento degli elementi: il valore dell'elemento in un nodo è sempre maggiore o uguale al valore degli elementi nei figli della radice



# PROVA 3

giovedì 15 luglio 2021 22:16

```
public static boolean verificaUnicità(AlberoBinario a) {}
```

è un metodo che restituisce true se e solo se tutti i nodi foglia contengono un valore che non appare in altri nodi di a

Dunque dovrò analizzare tutti i nodi foglia ed invocare poi un controllo di nuovo sull'albero

```
public static boolean verificaUnicità(AlberoBin<Integer> a) {  
    if (a == null) return true;  
    return verificaUnicità(a, a) && verificaUnicità(a, a);  
}
```

```
public static boolean verificaUnicità(AlberoBin<Integer> a, AlberoBin<Integer> b) {  
    if (a == null) return true;  
    if (a.des() == null && a.sin() == null)  
        return (presente(a.val(), b) == false) && verificaUnicità(a.des(), b) &&  
               verificaUnicità(a.sin(), b);  
    return verificaUnicità(a.des(), b) && verificaUnicità(a.sin(), b);  
}
```

```
private static boolean presente(int val, AlberoBin<Integer> a) {  
    if (a==null) return false;  
    if (a.des() != null || a.sin() != null)  
        return a.val().equals(val) || presente(val, a.des()) || presente(val,  
                           a.sin());  
    return presente(val, a.des()) || presente(val, a.sin());  
}
```

Nel caso migliore tale algoritmo va ad analizzare alcuni nodi di a ed alcuni di b per cui potremmo definire il costo come lineare. Nel caso peggiore deve verificare tutti i nodi di a per ciascuna verifica di ciascun nodo dunque il costo sarà Theta( $n^2$ ).

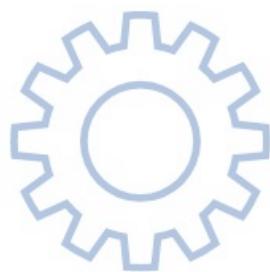
Il costo spaziale sarà costante nel primo caso e ed esponenziale nel secondo

RISPOSTE:

- 1) VERO -> perché per sommare n elementi bisogna per forza visitarli tutti
- 2) VERO -> nel caso peggiore perché nel caso migliore è costante
- 3) VERO -> quando chiede le funzioni chiede come diventano in notazione asintotica, non qual è la notazione corretta
- 4) FALSO -> nel caso peggiore/complessità intrinseca non è 1 (se non specifica è migliore/non ordinato)
- 5) FALSO -> l'algoritmo di floyd si applica a grafi orientati -> floyd non funziona per

cicli negativi!!!

- 6) FALSO -> l'inserimento è a costo costante
- 7) FALSO -> è theta(n)
- 8) VERO -> esistono implementazioni con  $O(n^2)$
- 9) FALSO -> se è debolmente connesso vuol dire che togliendo l'orientamento rimane connesso. Tuttavia non è detto che tutti i nodi siano connessi e quindi potrebbero mancare dei cammini vista la presenza dell'orientamento
- 10) FALSO -> se è ciclico non ammette un unico albero ricoprente di costo minimi. Se comunque è aciclico potrebbe sempre presentare più alberi ricoprenti con lo stesso costo minimo



## APPUNTI DI INGEGNERIA INFORMATICA

GAIA BERTOLINO

# PROVA 4

venerdì 16 luglio 2021 10:36

input ABR (albero binario di ricerca) non necessariamente bilanciato (se non viene detto non va fatta questa assunzione).

Si assume che non ci siano numeri ripetuti.

```
public static int conta(AlberoBinario a, int a, int b) {}
```

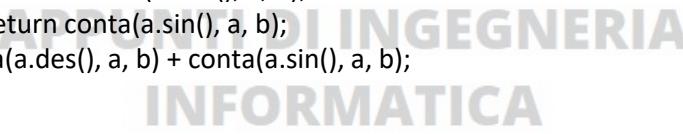
Calcolare il numero di nodi che hanno valori fra a e b

```
public static int conta(AlberoBinario a, int a, int b) {  
    if (a==null) return 0;  
    if ( a.val() >a && a.val()<b) return 1 + conta(a.des(), a, b) + conta(a.sin(), a, b);  
    return conta(a.des(), a, b) + conta(a.sin(), a, b)  
}
```

Tale algoritmo processa ciascun nodo una volta. Tuttavia, visto che bisogna sempre analizzare tutti i nodi (sia nel caso migliore che peggiore) il costo temporale sarà Theta(n).

Nel caso spaziale sarà, se l'albero è bilanciato ovvero un AVL, pari a Theta(log(n)) altrimenti nel caso peggiore Theta(n)

TUTTAVIA esiste una ottimizzazione

```
public static int conta(AlberoBinario a, int a, int b) {  
    if (a==null) return 0;  
    if ( a.val() < a ) return conta(a.des(), a, b);  
    if ( a.val() > b) return conta(a.sin(), a, b);  
    return 1 + conta(a.des(), a, b) + conta(a.sin(), a, b);  
}
```

con tale ottimizzazione si sfrutta la proprietà dell'albero binario di ricerca per cui il tempo si riduce a Theta(1) nell'ipotesi che tutti i nodi stiano nella metà che non andiamo ad analizzare mentre quella peggiore rimane sempre Theta(n) nel caso in cui tutti i nodi rientrino nel range

# PROVA 5

venerdì 16 luglio 2021 11:10

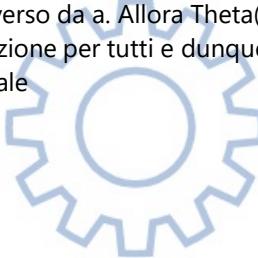
Dato un albero a binario semplice, scrivere un metodo che restituisce true se e solo se tutti i nodi foglia di a contengono un dato intero uguale a quello della radice di a

a è un albero. E' binario ma non di ricerca, non è bilanciato. In questo caso devo sempre analizzare tutte le lunghezze ma una sola volta.

```
public static boolean verifica(AlberoBinario a) {  
    if (a==null) return true;  
    return verifica(a, a.val());  
}  
  
private static boolean verifica(AlberoBinario a, int val) {  
    if (a==null) return true;  
    if (a.des() == null && a.sin() == null) return a.val().equals(val); //non ha senso andare oltre perché dopo non ho figli  
    return verifica(a.des(), val) && verifica(a.sin(), val);  
}
```

Nel caso migliore, visto che devono valere tutte le foglie, si ha un albero sbilanciato per cui vi è un sottoalbero composto da un solo nodo foglia che contiene valore diverso da a. Allora Theta(1) per spaziale e temporale.

Nel caso peggiore verifico sempre la condizione per tutti e dunque devo procedere ad analizzare almeno una volta tutti i nodi. Allora Theta(n) per spaziale e temporale



**APPUNTI DI INGEGNERIA  
INFORMATICA**

GAIA BERTOLINO

# PROVA 6

sabato 17 luglio 2021 17:30

Matricola:

---

---

---

---

---

---

---

---

---

---

*Esercizio 2*  
Dire quali delle seguenti affermazioni sono vere e quali false.

N°	Vero	Falso	Affermazione
1	✓		L'inserimento di un valore in una tabella hash contenente $n$ valori ha complessità temporale pari a $O(\lg n)$ nel caso peggiore.
2	✗		La funzione $f(n) = n^2$ è $\gg$ ( $\ln \lg(n^2)$ )
3	✗		L'algoritmo mergeSort ha complessità temporale $O(n \lg(n))$ nel caso peggiore.
4	✓		L'algoritmo di Floyd calcola, dati un grafo orientato e pesato e un nodo $x$ di tale grafo, la distanza minima fra $x$ e i nodi da esso raggiungibili.
5	✓		Sia $G$ un grafo orientato. Se $G$ contiene cicli allora non vi è nessun nodo in $G$ con grado di entrata zero.
6	✓		In un grafo non orientato connesso con $n$ nodi ci sono $n-1$ archi il numero di componenti connesse (non vuote) è uguale $n$ .
7	✗		L'algoritmo countingSort ha complessità $O(n)$ nel caso peggiore.
8	✓		Nel caso peggiore, rimuovere un elemento da un albero AVL con $n$ nodi ha complessità $O(\lg n)$ .
9	✗		Un albero binario a è completo, se e solo se per tutti i nodi $x$ di a, la differenza fra l'altezza del sottoalbero sinistro di $x$ e l'altezza del sottoalbero destro di $x$ è minore o uguale ad 1.
10	✗		La complessità intrinseca del problema della ricerca del secondo minimo in una sequenza disordinata è $\gg$ ( $\lg(n)$ )

*Esercizio 3*  
Si assume di avere un algoritmo Divide et Impera che divide l'istanza problema originario (di dimensione  $n$ ) in 2 istanze la cui dimensione è  $n-k$ , dove  $k$  è una costante. La fase di suddivisione del problema in sottoproblemi e di costruzione della soluzione dell'istanza originaria a partire dalle soluzioni delle istanze in cui essa è decomposta costa  $b^n$ . Calcolare la complessità dell'algoritmo.

Posso porre  $a=2$ ,  $d = 1$  e dunque ottengo che in questo caso la formula darà  $2^{n-n} + n^1 = 2^n + n$

- 1) FALSO -> è Theta( $n$ )
- 2) FALSO ->  $n^2$  è minore di  $n \log(n^n)$
- 3) VERO
- 4) VERO
- 5) FALSO -> può avere anche più di un nodo con grado di entrata maggiore di zero
- 6) VERO
- 7) FALSO -> è  $n+k$
- 8) VERO
- 9) FALSO -> deve essere 0
- 10) FALSO -> è  $n+n-1$  ovvero Theta( $n$ )



## APPUNTI DI INGEGNERIA INFORMATICA

GAIA BERTOLINO

# Appello 1

sabato 17 luglio 2021 17:34

## Esercizio 2

Dire quali delle seguenti affermazioni sono vere e quali false.

	V	F	Affermazione
1			La complessità spaziale delle visite anticipate di un albero binario ha complessità $O(\lg n)$ nel caso peggiore.
2			La funzione $f(n) = n^2$ è $O(n^3)$ .
3	X		L'algoritmo MergeSort ha complessità temporale $\Theta(n * \lg n)$ nel caso peggiore.
4			La ricerca di un elemento in una Hash table ha complessità temporale $O(n)$ nel caso peggiore.
5			Il problema della ricerca del minimo elemento in un array ordinato ha complessità intrinseca $\Omega(n)$ , dove $n$ è il numero di elementi nell'array.
6			Un grafo orientato è <i>debolmente连通</i> se sostituendo tutti gli archi diretti con archi indiretti si ottiene un grafo连通.
7			Preso un grafo non orientato, connesso e pesato (sugli archi) esiste un unico albero ricoprente di costo minimo.
8			Un grafo non orientato è connesso se e solo se ogni nodo è raggiungibile da ogni altro tramite un arco.
9			Dato un grafo $G$ la chiusura transitiva di $G$ contiene un arco fra due nodi $i$ e $j$ se e solo se in $G$ vi è un arco fra i nodi $i$ e $j$ .
10			Un grafo connesso ed orientato in cui esiste almeno un nodo con grado di entrata uguale a 0 può contenere un ciclo.

## Esercizio 3

Fornire le definizioni formali di albero binario e di albero binario bilanciato.

Un albero è binario se tutti i nodi tranne lo radice hanno grado  $\leq 2$  ovvero se ogni nodo ha al più 2 figli. È un albero formato da una radice e da 2 alberi sinistro e destro.

Un albero binario è bilanciato se per ogni nodo lo profondità del sottosalbero di destra e la profondità del sottosalbero di sinistra differiscono al più di 1.

Un albero binario ha due figli  
Può essere bilanciato o no, ordinato o no  
Inoltre può essere di ricerca o no

ERIA

# Appello 2

sabato 17 luglio 2021 17:35

Cognome: \_\_\_\_\_ Nome: \_\_\_\_\_ Matricola: \_\_\_\_\_

UNIVERSITÀ DEGLI STUDI DELLA CALABRIA  
Corso di Laurea in Ingegneria Informatica

Prova scritta di *Algoritmi e Strutture Dati* – TRACCIA A  
(durata della prova: 60 minuti)

**Esercizio 1**

Si consideri una classe *AlberoBinario* che rappresenta *alberi binari* in cui la parte informativa di ogni nodo è un numero intero. Si assuma che in tale classe siano implementati i seguenti metodi:

```
public interface AlberoBinario {
    /* restituisce il sottoalbero destro dell'albero corrente, la complessità temporale è O(1)*;
    public AlberoBinario destro();
    /* restituisce il sottoalbero sinistro dell'albero corrente, la complessità temporale è O(1)*;
    public AlberoBinario sinistro();
    /* restituisce il valore memorizzato nella radice dell'albero, la complessità temporale è O(1)*;
    public int val();
}
```

Si deve realizzare un metodo

```
public static boolean verificaCammini(AlberoBinario a, int l) { ... }
```

che restituisce true se e solo se esiste un nodo che si trova al livello l nell'albero a che ha valore 0 e i cui discendenti (compresi i figli) hanno tutti valore 0.

Si caratterizzi la complessità temporale e spaziale del metodo nel caso migliore e peggiore, specificando anche quali siano il caso migliore ed il caso peggiore per la complessità temporale e spaziale.

Caso Migliore: Caso Peggio:

1. Complessità temporale: O(\_\_\_\_)
2. Complessità spaziale: O(\_\_\_\_)

  1. Complessità temporale: O(\_\_\_\_)
  2. Complessità spaziale: O(\_\_\_\_)

Commenti:

*caso migliore: se l'albero è bilanciato, si ragiona solo per il figlio sinistro della radice o per il figlio destro. Nel caso peggiore: si analizzano tutti i nodi, quindi O(n).*

Il metodo in esame deve restituire un boolean.

La proprietà da verificare è che ci sia un nodo al livello l con valore 0 e discendenti sempre pari a 0. Dunque dovrà esserci un metodo di verifica di tale proprietà da richiamare sui figli

```
public static boolean verificaCammini(AlberoBinario a, int l) {
    if (a==null) return false;
    return verificaCammini(a, l, 0);
}

private static boolean verificaCammini(AlberoBinario a, int l, int liv) {
    if (a==null) return false;
    if (liv==l && a.val() == 0)
        return verifica(a.des()) && verifica(a.sin());
    return verificaCammini(a.des(), l, liv+1) || verificaCammini(a.sin(), l, liv+1);
}

private static boolean verifica(AlberoBinario a) {
    if (a==null) return true;
    if (a.val() != 0)
        return false;
    return verifica(a.des()) && verifica(a.sin());
}
```

Tale metodo nel caso migliore restituisce true subito nel caso in cui, avendo un albero sbilanciato, incontra subito il livello passato come input ad esempio nel figlio sinistro della radice o in uno dei nodi prossimi (vale lo stesso ragionamento in maniera speculare per il figlio destro) e valuta che i figli (si assume siano pochi e che la maggior parte siano appunto nel ramo destro) sono tutti zeri. Allora restituisce il risultato con costo temporale costante ovvero Theta(1).

Nel caso peggiore deve analizzare invece tutti i nodi, con conseguente costo pari a Theta(n). Nel caso spaziale valgono le considerazioni appena fatte per cui avremo nel caso migliore l'allocazione di poche chiamate delle funzioni sopra scritte e dunque un costo costante Theta(1) mentre nel caso peggiore sarà Theta(n).

## APPUNTI DI INGEGNERIA INFORMATICA

GAIA BERTOLINO

# Appello 3

sabato 17 luglio 2021 17:35

**Esercizio 2**  
Dire quali delle seguenti affermazioni sono vere e quali false.

	V	F	Affermazione
1	V	F	La complessità intrinseca del problema di calcolare la somma di un array di interi è $O(n)$ , dove $n$ è il numero di interi presenti nell'array.
2	F	V	L'inserimento di un valore in un heap binario può essere realizzato con un algoritmo di complessità $O(n)$ dove $n$ è il numero di elementi memorizzati nell'heap.
3	V	F	La funzione $f(n) = 2n^2$ è $O(n \lg n)$ .
4	V	F	Per ogni coppia di nodi $u, v$ appartenenti ad un grafo orientato debolmente connesso esiste sempre un cammino dal nodo $u$ al nodo $v$ e dal nodo $v$ al nodo $u$ .
5	V	F	La complessità spaziale della visita per livelli di un albero è $\Omega(n^2)$ , dove $n$ è il numero di nodi presenti nell'albero.
6	V	F	La complessità dell'inserimento di un elemento in una tabella hash in cui sono presenti $n$ elementi nel caso peggiore è $\Theta(n)$ .
7	V	F	Nel caso peggiore, ricercare un elemento in un albero AVL con $n$ nodi ha complessità $O(\log n)$ .
8	V	F	La complessità temporale dell'algoritmo di Dijkstra è $O(n^2)$ , dove $n$ è il numero di nodi presenti nel grafo, nel caso di rappresentazione con liste di adiacenza.
9	V	F	Per ogni coppia di nodi $u, v$ appartenenti ad un grafo orientato debolmente connesso esiste sempre un cammino dal nodo $u$ al nodo $v$ o dal nodo $v$ al nodo $u$ .
10	V	F	Un grafo non orientato connesso e pesato (sugli archi) ammette sempre un unico albero ricoprente di costo minimo.

**Esercizio 3**  
Dando per noto il concetto di albero binario, si definisca formalmente il concetto di albero binario di ricerca bilanciato.

*(Ricorda AVL)*

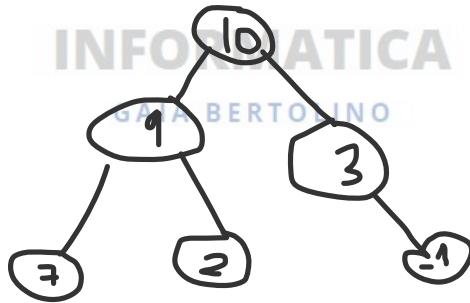
*Un'albero binario di ricerca è un albero binario che soddisfa le seguenti proprietà:* 1) ogni nodo in totale ha almeno due figli 2) è associato uno stesso criterio di ricerca da un determinato criterio binario 3) le due sottosearche risultanti sono bilanciate 4) le chiavi dei sottosearche destra di un nodo > chiavi del sottosearche sinistra di un nodo & viceversa 5) è bilanciato in altezza e in altezza dei sottosearchi sinistro e destro di ogni nodo differisce di più di un unità.

*In un albero AVL, molte altre volte si parla di alto e basso, questo parla un po' meno chiaro ma l'informazione se è altrettanto.*

- 1) VERO -> bisogna controllare esattamente tutti gli elementi dell'array (a meno che non si sappiano a priori delle proprietà come range di valori e presenza di ciascuno di essi)
- 2) FALSO -> Un heap binario è un albero in cui la relazione gerarchica esiste fra padre e figlio. In un maxheap il padre è maggiore dei figli e dunque il massimo si troverà nella radice; nel min heap invece il padre è più piccolo dei figli e dunque il minimo si troverà nella radice.  
Per inserire un valore potrebbe essere necessario
- 3) FALSO -> la funzione  $2n^2$  è maggiore di  $n \log(n)$  dunque sarebbe più corretto dire che  $f(n) = \Omega(n \log(n))$
- 4) FALSO -> debolmente connesso vuol dire che non tutti i nodi sono collegati seguendo l'orientamento del grafo stesso
- 5) FALSO -> il costo è  $\Theta(n)$
- 6) VERO -> nel caso di un hash imperfetto potrebbero esserci molte collisioni
- 7) VERO -> se si applica la ricerca binaria si dovrà al massimo attraversare tutto l'albero in altezza che
- 8) VERO -> nel caso dell'uso di matrici di adiacenza si può semplificare
- 9) FALSO
- 10) FALSO -> possono esisterne diversi

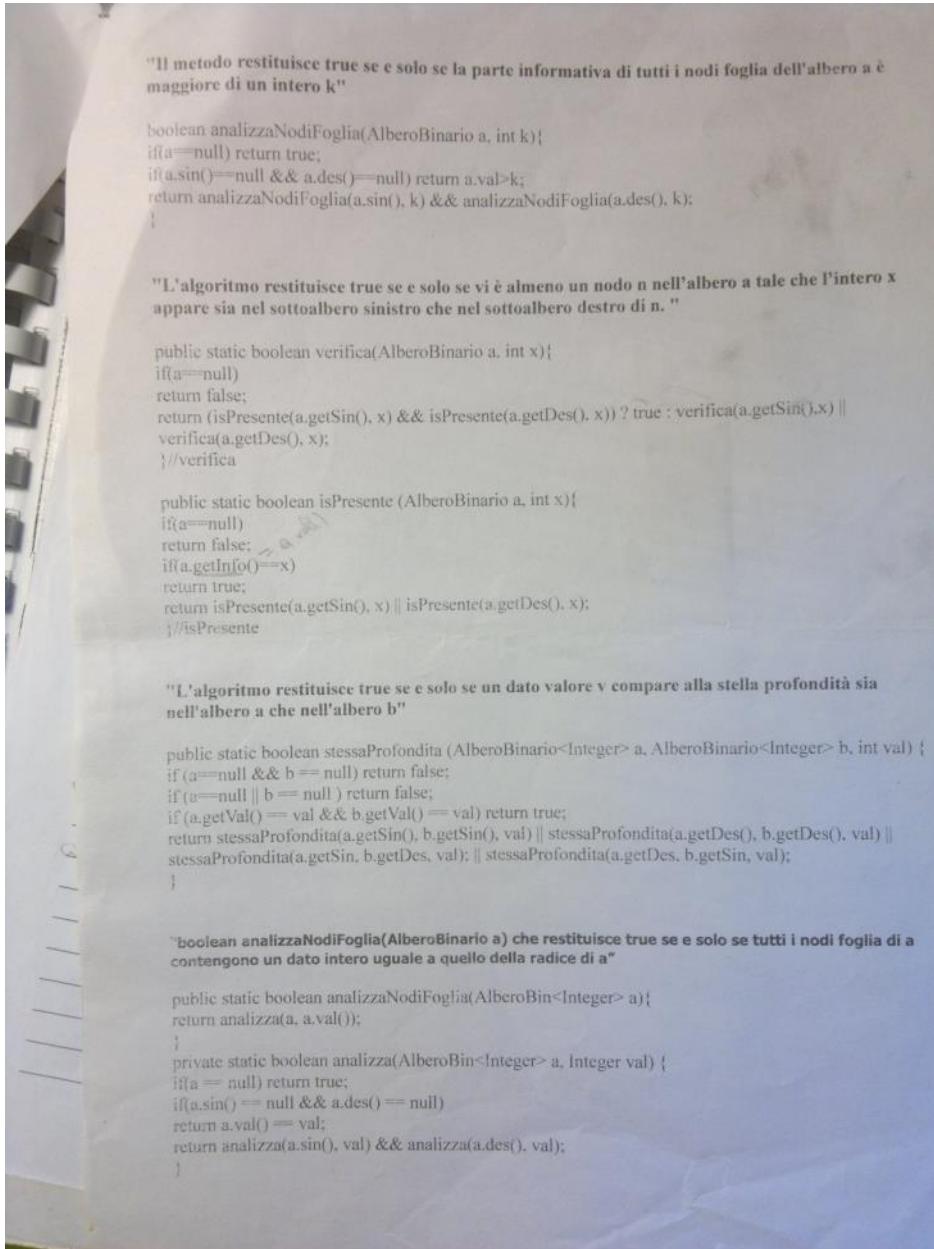


## APPUNTI DI INGEGNERIA INFORMATICA



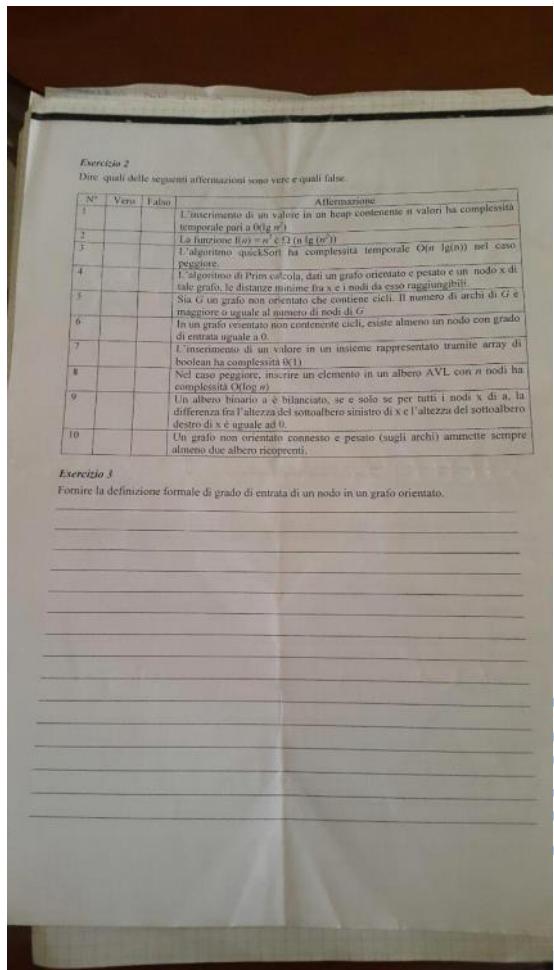
# Appello 4

sabato 17 luglio 2021 17:35



## Appello 5

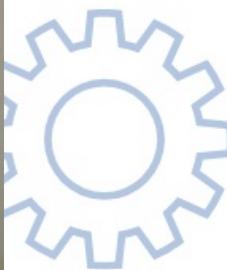
sabato 17 luglio 2021 17:35



N°	Vero	Falso	Affermazione
1			L'insertione di un valore in un heap contenente $n$ valori ha complessità temporale pari a $O(n^2)$ .
2			La funzione $f(n) = n^2$ è $\Omega(n \lg(n))$ .
3			L'algoritmo quickSort ha complessità temporale $O(n \lg(n))$ nel caso peggiore.
4			L'algoritmo di Prim calcola, da un grafo orientato e pesato e un nodo $v$ di tale grafo, le distanze minime fra $v$ e i nodi $u$ con grado raggiungibile.
5			Sia $G$ un grafo non orientato che contiene cicli. Il numero di archi di $G$ è maggiore o uguale al numero di nodi di $G$ .
6			In un grafo orientato non contenente cicli, esiste almeno un nodo con grado di entrata uguale a 0.
7			Per inserire un valore in un insieme rappresentato tramite array di booleani la complessità è $O(1)$ .
8			Nel caso peggiore, inserire un elemento in un albero AVL con $n$ nodi ha complessità $O(\log n)$ .
9			Un albero binario è bilanciato se e solo se per tutti i nodi $x$ di $u$ , la differenza fra l'altezza del sottosalvo sinistro di $x$ e l'altezza del sottosalvo destro di $x$ è uguale ad 0.
10			Un grafo non orientato connesso e pesato (sugli archi) ammette sempre almeno due alberi ricoprenti.

- 1) Vero
  - 2) Vero
  - 3) Falso  $\rightarrow$  Theta( $n^2$ )  $\rightarrow$  prende il pivot e lo sposta
  - 4) Falso  $\rightarrow$  Prim si applica a garfi non orientati
  - 5) Vero
  - 6) Vero
  - 7) Vero se disordinato
  - 8) Vero
  - 9) Falso minore o uguale a 1 in v.a.
  - 10) Falso

Grafo -> nodi legati fra di loro senza gerarchia. Tuttavia alcuni grafi sono anche alberi.  
Grafo orientato o meno  
Grado di entrata -> archi entranti



# ATTORNI DI INGEGNERIA INFORMATICA

## Appello 6

domenica 18 luglio 2021 12:51



4\_5985351  
06088403...

$$x = 1$$

### Esercizio 1

Si consideri una classe `AlberoBinario` che rappresenta alberi binari che rappresentano espressioni. Si assuma che in tale classe siano implementati i seguenti metodi:

```
public interface AlberoBinario{
    * restituisce il sottoalbero destro dell'albero corrente. la complessità temporale è Θ(1)*
    public AlberoBinario destro();
    * restituisce il sottoalbero sinistro dell'albero corrente. la complessità temporale è Θ(1)*
    public AlberoBinario sinistro();
    * Il metodo restituisce il valore memorizzato nella radice dell'albero. Se l'albero è una foglia il valore ha un significato altrimenti va ignorato. La complessità temporale è Θ(1)*
    public int val();
}
```

Si deve realizzare un metodo

```
public static int countEqualTo(AlberoBinario a, int x);
```

che calcola il numero dei nodi interni di  $a$  (compresa la radice se non è una foglia) aventi valore  $x$  che hanno entrambi i figli.

Si caratterizzi la complessità temporale e spaziale del metodo nel caso migliore e peggiore, specificando anche quali siano il caso migliore ed il caso peggiore per la complessità temporale e spaziale.

Caso Migliore	Caso Peggio
1. Complessità temporale $\Theta(1)$	1. Complessità temporale $\Theta(n^2)$
2. Complessità spaziale $\Theta(1)$	2. Complessità spaziale $\Theta(n)$

Commenta:

- 1) VERO -> se l'array è disordinato si devono almeno verificare  $n$  elementi per essere sicuri di trovare l'elemento cercato -> FALSO perché è al massimo  $n$  e non minimo  $n$
- 2) FALSO -> ha complessità  $\Theta(n^2)$
- 3) VERO -> la funzione  $\log(n^2) = 2 \log(n) < n \log(n)$  dove ho trasformato la funzione tramite le proprietà dei logaritmi
- 4) VERO -> posso opportunamente rappresentare tale grafo con le sembianze di un albero -> FALSO, deve essere anche connesso
- 5) FALSO -> la differenza deve essere assunta in valore assoluto altrimenti, per come è formulata la domanda, potrei avere una differenza pari a  $-2$  che rende l'albero sbilanciato ma bilanciato secondo la definizione fornita
- 6) FALSO -> nel caso migliore è  $\Theta(1)$  se ho una funzione hash perfetta e individuo dubbio l'elemento cercato
- 7) FALSO -> è  $\Theta(n)$
- 8) ---
- 9) FALSO -> potrebbero essere tutti nodi appartenenti al ciclo e quindi con grado di entrata almeno pari a uno
- 10) VERO

### Esercizio 2

Dire quali delle seguenti affermazioni sono vere e quali false.

V	F	Affermazione
1		La complessità intrinseca del problema di ricercare un elemento in un array di interi è $\Omega(n)$ , dove $n$ è il numero di interi presenti nell'array.
2		L'algoritmo QuickSort ha complessità temporale $\Theta(n * \lg n)$ nel caso peggiore.
3		La funzione $f(n) = \lg n^2$ è $\Theta(n \lg n)$ .
4		Sia $G$ un grafo non orientato ed aciclico. Il grafo $G$ è un albero.
5		Un albero binario è bilanciato se la differenza fra l'altezza del sottoalbero sinistro della radice e l'altezza del sottoalbero destro della radice è minore o uguale ad 1.
6		La complessità della ricerca di un elemento in una tabella hash in cui sono presenti $n$ elementi nel caso migliore è $\Theta(n)$ .
7		La complessità spaziale della visita infissa di un albero è $\Theta(n)$ , dove $n$ è il numero di nodi presenti nell'albero.
8		La complessità dell'algoritmo di Floyd è $\Theta(n^3)$ , dove $n$ è il numero di nodi nel grafo in input.
9		Dato un grafo orientato $G$ se $G$ contiene un ciclo allora c'è almeno un nodo con grado di entrata 0
10		In un grafo orientato un ciclo è un cammino che parte e finisce nello stesso nodo.

### Esercizio 3

Si assume di avere un algoritmo Divide et Impera che divide l'istanza problema originario (di dimensione  $n$ ) in 4 istanze la cui dimensione è  $n/2$ . La fase di suddivisione del problema in sottoproblemi e di costruzione della soluzione dell'istanza originaria a partire dalle soluzioni delle istanze in cui essa è decomposta costa  $b^n$ . Calcolare la complessità dell'algoritmo.

Tale problema presenta una equazione di ricorrenza che può essere risolta attavverso tre metodi: metodo dello strotolamento, metodo della sostituzione o attavverso il teorema master.

per il teorema master applicherò direttamente la soluzione attavverso dei parametri.

Assumo le seguenti grandezze:

$a = \text{numero istanze} = 4$

$n/c = \text{dimensione dati di ciascuna istanza} = n/2$  dunque  $c=2$

$d = \text{esponente della notazione asintotica del combina} = 3$

Verifico che il rapporto  $a/(c^d)$  è uguale a  $4/(2^3)$  che è minore di 1. Dunque in questo caso la soluzione sarà  $n^d$  ovvero  $n^3$

Verifica se un nodo interno ha entrambi i figli e valore x

Tale algoritmo ha come caso base un albero nullo per cui restituisce zero (infatti non può verificare la condizione necessaria all'incremento del contatore).

Per quanto riguarda il caso generale, dovrà verificare che ciascun nodo abbia dunque due figli e valore uguale o meno a quello passato come argomento.

Implemento dunque l'algoritmo utilizzando un if che verifica le due condizioni necessarie all'incremento del contatore.

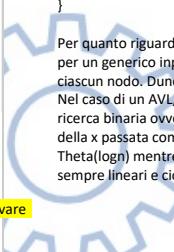
Se tale verifica non dà riscontro positivo, procedo a ripetere la verifica sui figli.

Tale algoritmo parte dalla radice e dunque ne tiene conto nel caso in cui non sia un nodo foglia.

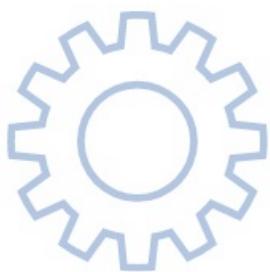
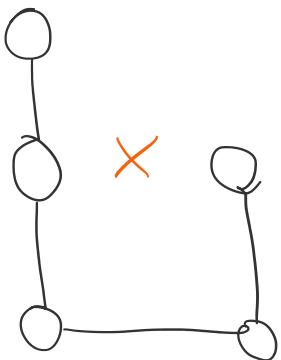
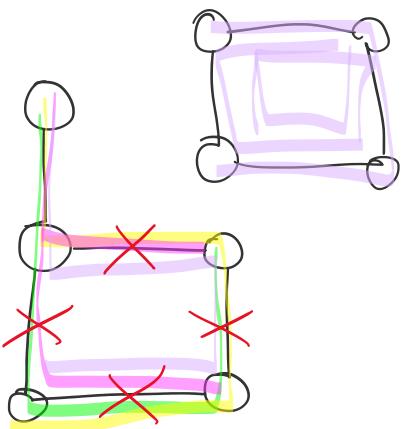
```
public static int countEqualTo(AlberoBinario a, int x) {
    if (a==null) return 0;
    if (a.des() != null && a.sin() != null && a.val() == x)
        return 1 + countEqualTo(a.des(), x) + countEqualTo(a.sin(), x);
    return countEqualTo(a.des(), x) + countEqualTo(a.sin(), x);
}
```

Per quanto riguarda il costo algoritmico di tale procedimento, non si hanno casi base per un generico input  $n$  in quanto il contatore necessita, sia nel caso migliore che peggiore, di verificare ciascun nodo. Dunque temporalmente esso sarà  $\Theta(n)$ .

Nel caso di un AVL, si potrebbe ottimizzare tale algoritmo riapplicando il metodo in una logica di ricerca binaria ovvero evitando quei rami in cui si sa già che i valori presenti sono minore (o maggiori) della  $x$  passata come argomento. Dunque in questo caso il costo temporale si abbasserebbe a  $\Theta(\lg n)$  mentre nel caso generale dell'algoritmo proposto, costo migliore e peggiore sarebbe sempre lineari e cioè  $\Theta(n)$ .



UNIVERSITÀ DI INGEGNERIA  
RISTORATORE  
BERTOLINO



## APPUNTI DI INGEGNERIA INFORMATICA

GAIA BERTOLINO

