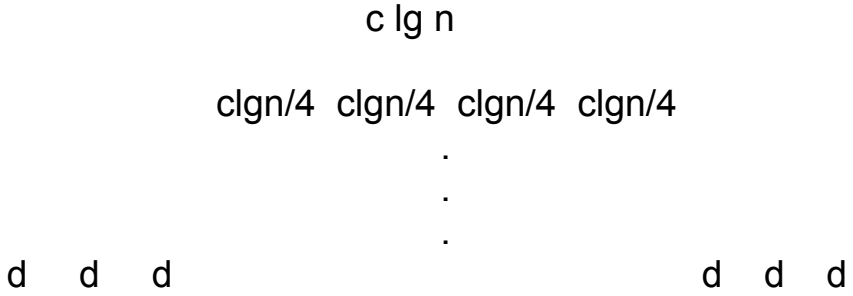
parte II - A

2. Si consideri la seguente funzione:

```
analizzami(int n)
        c = 1
        k = n*n
        while k > 1 do k = k/2
        for i = 0 to 3 do
                if n >1 then analizzami(n/4)
```

Si imposti la relazione di ricorrenza che ne descrive la complessità e la si risolva utilizzando il metodo della sostituzione.

Il ciclo while viene eseguito lg k volte, quindi lg n^2 volte, dunque tutto il codice al di fuori delle chiamate è eseguito in $\Theta(\lg n)$, perchè lg n^2 = 2 lg n.
Le chiamate sono 4 e su un quarto degli elementi, quindi
$T(n) = 4T(n/4) + \Theta(\lg n)$ è la relazione di ricorrenza che esprime il tempo di esecuzione della funzione ricorsiva analizzami.
Per risolverla esplicitiamo le costanti e aggiungiamo il caso base:
$T(n) = d$ se $n \leq 1$
$T(n) = 4T(n/4) + c \lg n$ altrimenti
L'albero della ricorsione, avendo posto $n = 4^h$ è



$T(n) = 4^h T(1) + \Sigma_{i=0,\ldots,h-1} 4^i \lg n/4^i = 4^h T(1) + \Sigma_{i=0,\ldots,h-1} 4^i (\lg n - \lg 4^i) \leq$
$4^h T(1) + \lg n \, \Sigma_{i=0,\ldots,h-1} 4^i = 4^h T(1) + \lg n \, O(4^h) = O(n \lg n)$

Per induzione verifichiamo il limite superiore, facendo vedere che possiamo trovare due costanti k ed n_0 tali che $T(n) \leq kn \lg n$, per ogni $n \geq n_0$.
Per ipotesi induttiva supponiamo che sia vero per tutti i valori m<n, allora
$T(n) = 4T(n/4) + c \lg n \leq 4 \, kn/4 \lg n/4 + c \lg n = kn(\lg n - \lg 4) + c \lg n =$
$kn \lg n - 2kn + c \lg n$.
Facciamo vedere che $kn \lg n - 2kn + c \lg n \leq k n \lg n$ per ottenere la tesi.
Semplifichiamo:
$kn \lg n - 2kn + c \lg n \leq k n \lg n \Leftrightarrow - 2kn + c \lg n \leq 0 \Leftrightarrow c \lg n \leq 2kn$ e ora vediamo che basta prendere $k = c$ e $n_0 = 1$.

Poiché T(1) = d e lg 1 = 0, consideriamo T(2) = 4d + c e cerchiamo un valore di k per cui anche T(2) = 4d + c ≤ 2 k, vediamo che basta prendere k = 2c + 2d, e concludiamo che l'asserto è vero per k=2c+2d e $n_0 = 2$.
Questo è quello che avreste potuto fare tutti.
Le considerazioni seguenti sono messe per amor di precisione.

Qui abbiamo maggiorato eliminando il termine negativo per ottenere un limite superiore, ma osserviamo che se la sua considerazione ci portasse a eliminare il termine in n lg n, resterebbe un termine in O(n) e non abbiamo motivi di escludere questo andamento.

Possiamo vedere cosa succede facendo i calcoli precisi. Ci serviranno le seguenti formule:
la formula per il calcolo della somma finita di una serie di potenze e quella per una somma di potenze moltiplicate per l'esponente:
$\Sigma_{i=0,\ldots,h-1}\ 4^i = (4^h - 1)/3$ e
$\Sigma_{i=0,\ldots,h-1}\ 4^i\ i = 1/9\ (4 - h4^h + (h - 1)4^{h+1}) = 1/9(4 - h4^h + 4h4^h - 4^{h+1}) = 1/9(4 + 3h4^h - 4^{h+1}) = 4/9 + 1/3h4^h - 4^{h+1}/9.$
Quindi

$\Sigma_{i=0,\ldots,h-1}\ 4^i\ (\lg n - \lg 4^i) = \lg n\ \Sigma_{i=0,\ldots,h-1}\ 4^i - \Sigma_{i=0,\ldots,h-1}\ 4^i\ 2i = \lg n\ (4^h - 1)/3 - 2\Sigma_{i=0,\ldots,h-1}\ 4^i\ i = \lg n\ (4^h - 1)/3 - 2(4/9 + 1/3h4^h - 4^{h+1}/9)$
= (ricordando che $n = 4^h$) $2/3h4^h - 2h/3 - 8/9 - 2/3h4^h + 2/94^{h+1} = - 2h/3 - 8/9 + 2/9*4^{h+1} = O(n)$

Dunque possiamo prevedere che la relazione di ricorrenza abbia un andamento lineare.
Più in dettaglio abbiamo ottenuto T(n) = O(n) - O(lg n), questo giustifica quanto segue. Per dimostrare che T(n) = O(n) non possiamo usare la più semplice funzione, kn, in O(n), infatti se usiamo l'ipotesi induttiva si ottiene $T(n) \le 4kn/4 + c \lg n$ per opportune costanti k e n_0. Imponendo che $kn + c \lg n \le kn$ non si riesce a determinare un valore di k che renda vera la relazione, perché resta $c \lg n \le 0$ che è falso.
Prendiamo allora la funzione k(n - lg n) in O(n) e dimostriamo che T(n) ≤ k(n - lg n), per un opportuno k per ogni $n \ge n_0$.
Sfruttando l'ipotesi induttiva si ha
$T(n) \le 4T(n/4) + c \lg n \le 4\ k(n/4 - \lg n/4) + c \lg n = kn - 4k \lg n + 8k + c\lg n.$
Imponendo $kn - 4k \lg n + 8k + c \lg n \le kn - k \lg n$ si ottiene
$c \lg n \le 3k\lg n - 8k$ che è vera per k = 3c e per ogni $n \ge n_0 = 8$, infatti
$c \lg n \le c\lg n + 8\ c\lg n - 24c \Leftrightarrow 8c\lg n \ge 24c \Leftrightarrow c\lg n \ge 3c \Leftrightarrow \lg n \ge 3 \Leftrightarrow n \ge 8$

Andando al caso base si ha T(8) = 4T(2) + 3 c = 16T(2/4) + c + 3c = 16d + 4c, imponiamo che T(8) ≤ 8k - 3k = 5k, si ha 16d + 4c ≤ 5k , questo può essere reso vero prendendo k = 16d + 4c.
Possiamo concludere che T(n) = O(n), avendo trovato k = 16d +4c e $n_0 = 8$ tali che $T(n) \le kn - k \lg n$ per ogni $n \ge n_0$.

Volendo possiamo dimostrare anche che $T(n) = \Omega(n)$, facendo vedere che $T(n) \geq k\, n$ per opportune costanti k e n_0. e per ogni $n \geq n_0$.
Per ipotesi induttiva $T(n) \geq 4kn/4 + c \lg n$, imponendo che $kn + c \lg n \geq kn$, si ottiene che questo è vero per ogni scelta di k e $n > 1$ e quindi anche per $k = 16d + 4c$ e $n_0 = 8$. Dunque $T(n) = \Theta(n)$.

Es. 3
Si descriva un algoritmo che presi in input due alberi AVL T1 ed T2 fornisce in output tutte le chiavi comuni ai due alberi.
Si fornisca la valutazione asintotica del tempo di esecuzione dell'algoritmo proposto. Si valuti il tempo di esecuzione nel caso in cui T1 e T2 siano alberi binari di ricerca qualunque invece di essere alberi AVL.

Soluzione.

Si può adattare l'idea della funzione di fusione di due array ordinati, eseguendo una visita inorder per ogni albero, in modo da visitarli in ordine crescente. Si utilizza la visita iterativa basata sull'individuazione del minimo e il calcolo del successivo. Ogni volta che si considerano due nodi x in T1, di chiave a, e y in T2 , di chiave b, si confrontano le chiavi e

1. se a = b il loro valore deve essere stampato e si passa al confronto tra le successive delle due
2. se a > b si cerca in T2 il nodo di chiave successiva a b in modo da poterla confrontare con a, infatti a potrebbe essere presente in T2
3. se a < b si cerca in T1 il nodo di chiave successiva ad a in modo da poterla confrontare con b, in questo caso infatti a non è presente in T2.

Il processo termina quando si è visitato almeno uno dei due alberi.
Il tempo asintotico è in O(n+m), se n è il numero dei nodi di T1 e m quello di T2, perché può capitare di dover completare la visita su ciascuno dei due alberi, per esempio se i valori delle chiavi fossero perfettamente alternati nell'ordine.
Nel caso gli alberi siano semplici ABR la complessità non cambia, infatti con questa soluzione l'altezza dell'albero non pesa sul tempo di esecuzione.

Pseudocodice:

```
Comuni(T1,T2)
input: T1 e T2 sono i puntatori alla radice di alberi binari
precondizione: T1 e T2 sono AVL
output: stampa le chiavi comuni ai due alberi
if T1 == NULL  or T2 ==  NULL return "uno dei due alberi è vuoto"
x = MINIMUM(T1)
```

```
y = MINIMUM(T2)
while x ≠ NULL and y ≠ NULL do
    if x.key == y.key  then print x.key; x = successor(T1,x); y = successor(T2,y)
    if x.key < y.key then x = successor(T1,x);
    if x.key > y.key then y = successor(T2,y);
```

Breve descrizione delle soluzioni alternative “ragionevoli”.
1. Si poteva immaginare di scaricare il risultato delle due visite inorder in due array, per poi fonderli, visto che sono ordinati, e cercare i doppioni nell’array ottenuto dalla fusione, dato che nei nostri alberi i duplicati sono esclusi. Si può anche evitare di fare la fusione e semplicemente modificare la funzione di fusione (merge) in modo analogo alla soluzione su presentata che opera direttamente sugli alberi.
Questa soluzione ha lo stesso tempo di esecuzione asintotico della prima, ma impegna anche $\Theta(n+m)$ memoria aggiuntiva (n è il numero dei nodi di T1 e m quello di T2)
2. Si poteva impostare una visita del primo albero con ricerca della chiave di ogni nodo visitato nel secondo, utilizzando quindi una qualunque delle visite sugli alberi binari. Questa soluzione ha tempo di esecuzione asintotico $O(n \lg m)$. Questa è l’unica soluzione il cui tempo di esecuzione cambia passando agli ABR perché l’altezza non è più garantito che sia logaritmica. In questo caso il tempo di esecuzione è in $O(nh)$, dove h è l’altezza di T2.

parte II - B

2.
```
analizzami(int n)
    c = 1
    k = n*n
    while k > 1 do k = k - 2
    for i = 0 to 1 do
    if n >1 then analizzami(n/2)
```

Il ciclo while viene eseguito k/2 volte, quindi $n^2/2$ volte, dunque tutto il codice al di fuori delle chiamate è eseguito in $\Theta(n^2/2)$.
Le chiamate sono 2 e quindi
$T(n) = 2T(n/2) + \Theta(n^2/2)$ è la relazione di ricorrenza che esprime il tempo di esecuzione della funzione ricorsiva analizzami.
Per risolverla esplicitiamo le costanti e aggiungiamo il caso base:
$T(n) = d$ se $n \leq 1$
$T(n) = 2T(n/2) + cn^2$ altrimenti

L’albero della ricorsione, avendo posto $n = 2^h$ è

cn^2

$c(n/2)^2$ $c(n/2)^2$

.
.
.

d d d d d d

$T(n) = 2^h T(1) + \Sigma_{i=0,\ldots,h-1} 2^i(n/2^i)^2 = 2^h T(1) + \Sigma_{i=0,\ldots,h-1} 2^i (n^2/2^{2i}) =$
$2^h T(1) + n^2\Sigma_{i=0,\ldots,h-1} (1/2^i) \leq 2^h T(1) + n^2\Sigma_{i=0,\ldots,\infty} (1/2^i) = O(n^2)$, perché $\Sigma_{i=0,\ldots,\infty}$ $(1/2^i) = \Theta(1)$, cioè converge a una costante.
Per induzione verifichiamo il limite superiore, facendo vedere che possiamo trovare due costanti k e n_0 tali che $T(n) \leq kn^2$, per ogni $n \geq n_0$.
Per ipotesi induttiva supponiamo che sia vero per tutti i valori m<n.
$T(n) = 2T(n/2) + cn^2 \leq 2\ kn^2/4 + c\ n^2 = kn^2/2 + c\ n^2$.
Facciamo vedere che $kn^2/2 + c\ n^2 \leq kn^2$ per ottenere la tesi.
Semplifichiamo, supponendo $n \geq 1$, $k/2 + c \leq k \Leftrightarrow k + 2c \leq 2k \Leftrightarrow 2c \leq k$ e ora vediamo che basta prendere $k = 2c$ e $n_0 = 1$.
Poiché $T(1) = d$, cerchiamo un valore di k per cui anche $T(1) \leq k$, vediamo che basta prendere $k = 2c + d$, e concludiamo che l'asserto è vero per $k=2c+d$ e $n_0 = 1$.
Poiché il termine aggiuntivo nella relazione di ricorrenza $T(n) = 2T(n/2) + \Theta(n^2/2)$ è quadratico possiamo dir che $T(n) = \Omega(n^2)$ e di conseguenza che $T(n) = \Theta(n^2)$.

Es. 3
Si descriva un algoritmo che presi in input due alberi AVL T1 ed T2 fornisce in output le chiavi di T1 che non compaiono in T2.
Si fornisca la valutazione asintotica del tempo di esecuzione dell'algoritmo proposto.
Si valuti il tempo di esecuzione nel caso in cui T1 e T2 siano alberi binari di ricerca qualunque invece di essere alberi AVL.

Soluzione
Si può adattare l'idea della funzione di fusione di due array ordinati, eseguendo una visita inorder per ogni albero, in modo che le chiavi siano visitate in ordine crescente. Si utilizza la visita iterativa basata sull'individuazione del minimo e il calcolo del successivo. Ogni volta che si considerano due nodi x, di chiave a, e y, di chiave b, nei due alberi T1 e T2 e si confrontano le chiavi e
1. se a = b si passa al confronto tra i successivi dei due

2. se a > b si cerca in T2 il nodo di chiave successiva a b in modo da poterla confrontare con a, infatti a potrebbe essere in T2
3. se a < b si stampa a, perché si è sicuri che non compare in T2, e si cerca in T1 il nodo di chiave successiva ad a in modo da poterla confrontare con b.

Questo processo termina quando si è visitato almeno uno dei due alberi.
Se è T2 l'albero di cui è terminata la visita, si stampano le chiavi in T1 successive all'ultima in T1 confrontata con quella di valore massimo di T2.
Il tempo asintotico è in O(n+m), se n è il numero dei nodi di T1 e m quello di T2, perché può capitare di dover completare la visita su ciascuno dei due alberi, per esempio se i valori delle chiavi fossero perfettamente alternate nell'ordine.
Nel caso gli alberi siano semplici ABR la complessità non cambia, infatti l'altezza dell'albero non pesa sul tempo di esecuzione.

Pseudocodice:

```
Diff(T1,T2)
input: T1 e T2 sono i puntatori alla radice di alberi binari
precondizione: T1 e T2 sono AVL
output: stampa le chiavi di T1 che non sono in T2
if T1 == NULL  or T2 ==  NULL return "uno dei due alberi è vuoto"
x = MINIMUM(T1)
y = MINIMUM(T2)
while x ≠ NULL and y ≠ NULL do
        if x.key == y.key  then x = successor(T1,x); y = successor(T2,y)
        if x.key > y.key then y = successor(T2,y);
        if x.key < y.key then print x.key; x = successor(T1,x);
if y≠ NULL then
        while x ≠ NULL do
                print x.key; x = successor(T1,x);
```

La soluzione alternativa in cui gli elementi ottenuti dalla visita inorder sono scaricati in due array, analogamente al caso del testo A, comporta poi o un'adattamento della funzione di fusione, analogo a quello adottato direttamente per le chiavi o l'uso delle ricerca binaria degli elementi di T1 in T2, in questo caso si stampano solo le chiavi per cui la ricerca binaria fallisce.
Questa soluzione nel primo caso ha lo stesso tempo di esecuzione asintotico della prima, nel secondo invece il tempo è in O(n lg m), ma impegnano anche Θ(n+m) memoria aggiuntiva (n è il numero dei nodi di T1 e m quello di T2).
Se si visita T1 cercando ogni elemento di T1 in T2 e stampando quelli non trovati si ottiene una soluzione in O(n lg m).

parte I

Esercizio 2.

a. Si determini la funzione g con la quale esprimere il limite asintotico stretto della seguente funzione:

$f(n) = 18n^3 + \lg n^8$

Si dimostri che il limite trovato è corretto.

Sol. a. $18n^3 + \lg n^8 = 18n^3 + 8\lg n = \Theta(n^3)$.

Per dimostrarlo bisogna trovare tre costanti c_1, c_2 e n_0 tali che
$c_1 n^3 \leq 18n^3 + 8\lg n \leq c_2 n^3$, per ogni $n \geq n_0$.
Consideriamo la prima disuguaglianza, per dimostrare che $18n^3 + 8\lg n = \Omega(n^3)$:
$c_1 n^3 \leq 18n^3 + 8\lg n$, basta prendere $c_1 = 18$ e $n_0 = 1$ e si ha $18n^3 + 8\lg n \geq 18n^3$, per ogni $n \geq n_0$.
Consideriamo la seconda disuguaglianza, per dimostrare che $18n^3 + 8\lg n = O(n^3)$:
$18n^3 + 8\log n \leq c_2 n^3$, poiché $n \geq \lg n$ per ogni $n \geq 1$, anche $n^3 \geq \lg n$, quindi basta prendere $c_2 = 18+8 = 26$ e $n_0 = 1$ e si ha $18n^3 + 8\lg n \leq 18n^3 + 8n^3 = 26n^3$, per ogni $n \geq n_0$.

b. Si consideri un algoritmo in cui ogni elemento dispari in un array di interi viene moltiplicato per due.
Quale fra le notazioni O e θ è più appropriata per misurare il numero di moltiplicazioni? E quale è più appropriata per misurare il tempo di esecuzione asintotico totale?

Sol. b. Il numero delle moltiplicazioni è $\Theta(n)$ nel caso peggiore in cui tutti gli n elementi dell'array sono dispari, quindi in generale possiamo dire che è $O(n)$. Il tempo di esecuzione asintotico totale è $\Theta(n)$ in tutti i casi perché bisogna sempre esaminare tutti gli elementi dell'array.

Es. 3 Siano date n monete d'oro tutte dello stesso peso, tranne una che è più leggera delle altre, ed una bilancia con due piatti, su ciascuno dei quali è possibile mettere un numero qualunque di monete e sapere se i piatti hanno lo stesso peso, o quale dei due è più leggero. Progettare un algoritmo per trovare la moneta "leggera" che richieda $O(\lg n)$ pesate nel caso peggiore.

Soluzione.
Si può risolvere con un algoritmo basato sulla ricerca binaria, in modo che il numero delle pesate sia in $O(\lg n)$.
Se le monete sono in numero pari poniamo la metà delle monete sui due piatti e il piatto il cui peso totale è minore è quello su cui ripetere le pesate per cercare la moneta falsa.

Se le monete sono in numero dispari, si esclude una moneta e si pesano le rimanenti due metà nei due piatti. Se le due metà monete hanno lo stesso peso allora la moneta più leggera è quella esclusa e terminiamo la ricerca. Altrimenti uno dei due piatti è più leggero ed è quello sul quale c'è anche la moneta più leggera, quindi si procede con le pesate su questa metà. Nel caso migliore l'algoritmo termina in Θ(1), quando per esempio le monete sono in numero dispari e la moneta falsa è l'ultima, mentre nel caso peggiore si fanno tante pesate quante volte è possibile dividere n per due prima di arrivare a un valore minore o uguale a uno, cioè O(lg n).

```
PiùLeggera(A)
input: un array di numeri contenenti i pesi delle monete.
precondizione: i pesi sono tutti uguali tranne uno che è minore.
output: l'indice della moneta più leggera
n = A.length
i=0 j = n-1
while i < j do
    k = (j - i + 1)/2 # in k la metà del numero degli elementi
      if (j-i+1) è dispari then
            if A[i] + …+A[i+k-1] = A[i+k] + …+A[j - 1] then return j
           else j = j-1
      if A[i] + …+A[i+k-1] < A[i+k] + …+A[j] then j = i+k-1 else i = i+k
return j
```