Corso di laurea in Informatica
Algoritmi 1
A.A. 2025/2026

Esercizi riepilogativi

Tiziana Calamoneri

SAPIENZA
UNIVERSITÀ DI ROMA

Slides realizzate sulla base di quelle preparate da T. Calamoneri e G. Bongiovanni per il corso di Informatica Generale tenuto a distanza nell'A.A. 2019/20

Esercizio svolto 1 (1)

Esercizio 1. Si dimostri, utilizzando la definizione di Θ, che

$$f(n) = \lg^2(2n) + \lg 4n = \Theta(\lg^2 n)$$

mettendo in evidenza e commentando con chiarezza i passi seguiti.

Soluzione.

- Per dimostrare che f(n) è in $\Theta(\lg^2 n)$ dobbiamo trovare tre costanti positive c,c' ed n' tali che

  $$c \lg^2 n \leq \mathbf{\lg^2(2n) + \lg 4n} \leq c' \lg^2 n \text{ per ogni } n \geq n'.$$

- Trattiamo le due disuguaglianze separatamente.

Esercizio svolto 1 (2)

1. $c \lg^2 n \leq \lg^2(2n) + \lg 4n$ è banalmente vero ad es. per $c = 1$, perché il logaritmo è una funzione crescente per ogni $n \geq 1$.

2. $\lg^2(2n) + \lg 4n \leq c' \lg^2 n$ infatti:

   poiché $\lg^2(2n) + \lg 4n = (\lg 2 + \lg n)^2 + (\lg 4 + \lg n) =$

   $= 1 + \lg^2 n + 2 \lg n + 2 + \lg n =$

   $= \lg^2 n + 3 \lg n + 3$

   la disuguaglianza diventa:

   $\lg^2 n + 3 \lg n + 3 \leq c' \lg^2 n$

che è vera ad esempio per $c' = 7$ ed $n \geq 2$.

Esercizio svolto 2 (1)

Esercizio 2. Si imposti la relazione di ricorrenza che definisce il tempo di esecuzione della seguente funzione e si trovi una limitazione superiore per la sua soluzione usando il metodo della sostituzione.

```
Strano(A,i,j))
        n = j-i+1
        if (n ≤ 1):
                return 1
        m = n/2
        while n > 0:
                n = n/2
        return Strano(A,i,i+m) +  Strano(A,i+m,j)
```

Esercizio svolto 2 (2)

```
Strano(A,i,j))
    n = j-i+1
    if (n ≤ 1):
        return 1
    m = n/2
    while n > 0:
        n = n/2
    return Strano(A,i,i+m) +
                Strano(A,i+m,j)
```

SOLUZIONE:

- La dimensione dell'input è n
- il caso base si ha quando n ≤ 1 e può essere eseguito in tempo costante senza ricorsione;
- in generale, la funzione Strano è chiamata 2 volte su n/2 elementi e il ciclo while è eseguito in ogni chiamata in tempo logaritmico, quindi la relazione di ricorrenza è:
  - $T(n) = 2T(n/2) + \Theta(\lg n)$
  - $T(1) = \Theta(1)$

Esercizio svolto 2 (3)

Per applicare il metodo di sostituzione dobbiamo eliminare la notazione asintotica:

- $T(n) = 2T(n/2) + c \lg n$
- $T(1) = d$

Hp (da verificare per induzione):

esiste k positiva k t.c. $T(n) \leq k\, n \lg n$, per ogni n.

Il passo base è verificato infatti:

$T(1) = d$ non può essere usato, ma $T(2) = 2d+c$ e

$2d+c=T(2) \leq 2k$ è vero prendendo ad es. $k \geq (d+c)/2$.

Esercizio svolto 2 (4)

Sia ora vera la tesi per ogni m < n (hp induttiva):

$T(m) \leq k\, m \lg m$ e dimostriamo per n:

$T(n) = 2T(n/2) + c \lg n \leq$

$2k\,(n/2) \lg n/2 + c \lg n =$

$2k\,(n/2)(\lg n - 1) + c \lg n =$

$k\, n \lg n - kn + c \lg n \leq k\, n \lg n$ se e solo se

$-kn + c \lg n \leq 0.$

Quest'ultima disug. è vera per $k \geq c$ e per ogni $n \geq 1$.

Segue che $T(n)=O(n \log n)$.

Esercizio svolto 3 (1)

Esercizio 3. Si risolva la seguente equazione di ricorrenza con due metodi diversi, per ciascuno dettagliando il procedimento usato:

$$T(n) = T(n/2) + \Theta(n^2) \text{ se } n > 1$$
$$T(1) = \Theta(1)$$

tenendo conto che n è una potenza di 2.

Soluzione

Utilizziamo dapprima il metodo principale, che è il più rapido, e poi verifichiamo la soluzione trovata con il metodo iterativo.

Esercizio svolto 3 (2)

Metodo principale:

Le costanti a e b del teor. princ. valgono qui 1 e 2, quindi $\log_b a = 0$ ed $n^{\log_b a} = 1$: ponendo $\varepsilon < 2$, siamo nel caso 3. del teorema, se vale che a $f(n/b) \leq c\, f(n)$.

Poiché f(n) è espressa tramite notazione asintotica, per verificare la disuguaglianza, dobbiamo eliminare tale notazione, e porre ad esempio $f(n) = \Theta(n^2) = hn^2$ e $h \geq 0$

Ci chiediamo se esista una costante $c<1$ tale che $h(n/2)^2 \leq c(hn^2)$.

Risolvendo otteniamo: $hn^2(1/4 - c) \leq 0$ che è verificata ad esempio per $c = 1/2$.

Ne possiamo dedurre che $T(n) = \Theta(n^2)$.

Esercizio svolto 3 (4)

Metodo iterativo:

Dall'equazione di ricorrenza deduciamo che:

- $T\left(\frac{n}{2}\right) = T\left(\frac{n}{2^2}\right) + \Theta\left(\left(\frac{n}{2}\right)^2\right),\ T\left(\frac{n}{2^2}\right) = T\left(\frac{n}{2^3}\right) + \Theta\left(\left(\frac{n}{2^2}\right)^2\right)$

e così via. Sostituendo:

- $T(n) = T\left(\frac{n}{2}\right) + \Theta(n^2) = T\left(\frac{n}{2^2}\right) + \Theta\left(\left(\frac{n}{2}\right)^2\right) + \Theta(n^2) =$

$$= T\left(\frac{n}{2^3}\right) + \Theta\left(\left(\frac{n}{2^2}\right)^2\right) + \Theta\left(\left(\frac{n}{2}\right)^2\right) + \Theta((n)^2) = \ldots$$

$$= T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} \Theta\left(\left(\frac{n}{2^i}\right)^2\right)$$

Esercizio svolto 3 (5)

Metodo iterativo (segue):

Continuiamo ad iterare fino al raggiungimento del caso base, cioè fino a quando $n/2^k = 1$, il che avviene se e solo se $k = \log n$.

L'equazione diventa così:

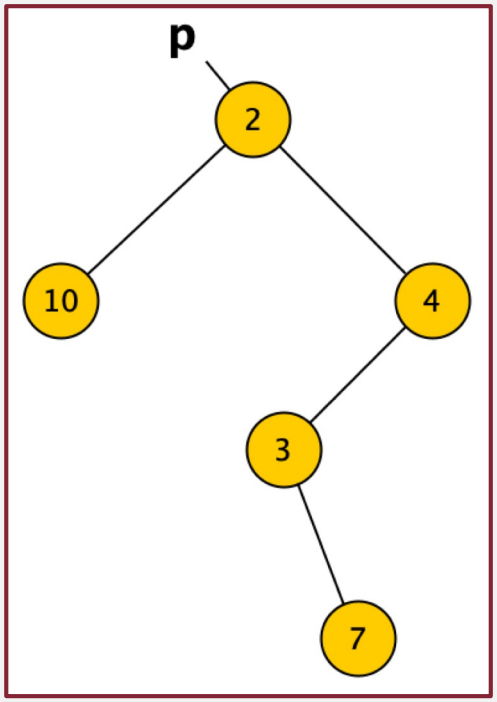
$$T(n) = T(1) + \sum_{i=0}^{\log n - 1} \Theta\left(\left(\frac{n}{2^i}\right)^2\right) = \Theta(1) + n^2 \sum_{i=0}^{\log n - 1} \Theta\left(\frac{1}{4^i}\right)$$

Ricordando che $\sum_{i=0}^{b} a^i = \frac{a^{b+1}-1}{a-1}$ otteniamo infine:

$$T(n) = \Theta(1) + n^2\, \Theta\left(\frac{1-\left(\frac{1}{4}\right)^{\log n}}{1-\frac{1}{4}}\right) = \Theta(n^2)$$

Esercizio svolto 4 (1)

Esercizio 4. Progettare una funzione `Crea` che, dato il puntatore alla radice di un albero binario memorizzato tramite puntatori, restituisca l'albero in notazione posizionale.



0	1	2	3	4	5	6	7	8	9	10	11	12
2	10	4	--	--	3	--	--	--	--	--	--	7

Esercizio svolto 4 (2)

Soluzione.

Se l'albero puntato da p contiene nodi, allora si quantifica lo spazio necessario per inserire nell'array A i nodi dell'albero (già sappiamo che questo valore può essere esponenziale in n…) e poi si inseriscono in A i nodi dell'albero nelle posizioni opportune:

```
def crea(p):
    if p == None:
        return []
    n = spazio(p)
    A = [None]*(n+1)
    inserisci(p, A)
    return A
```

Esercizio svolto 4 (3)

```
def spazio(p, x = 0):
    # restituisce la locazione massima
    # necessaria a sistemare i nodi nella
    # versione posizionale dell'albero non vuoto
    a = b = 0
    if p.left:
        a=spazio(p.left, 2*x+1)
    if p.right:
        b=spazio(p.right, 2*x+2)
    return max(a,b,x)
```

Esercizio svolto 4 (4)

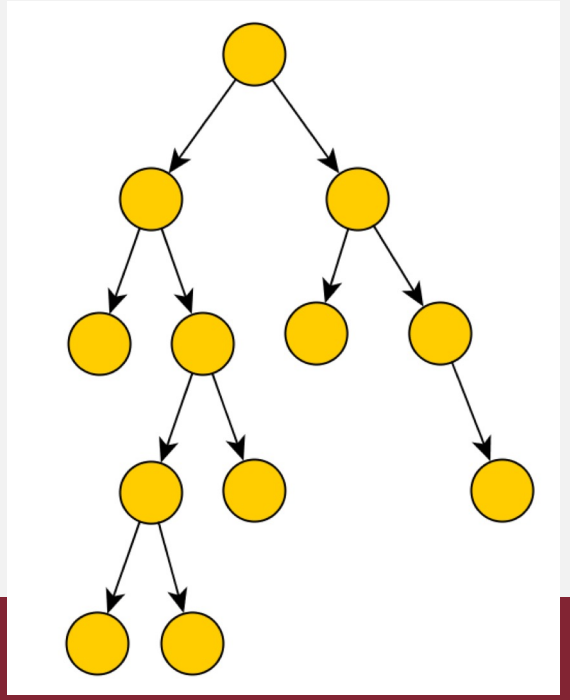
```
def inserisci(p, A, x=0):
    # per ogni nodo dell'albero a puntatori
    # crea un nodo  nell'array A
    if p!= None:
        A[x]=p.key
        if p.left:
            inserisci(p.left, A, 2*x+1)
        if p.right:
            inserisci(p.right, A, 2*x+2)
```

Esercizio svolto 5 (1)

Esercizio 5. Dato il puntatore alla radice di un albero binario non vuoto di n nodi, memorizzato tramite record e puntatori, progettare un algoritmo che restituisca il livello minimo in cui compaiono le sue foglie.

L'algoritmo deve avere costo computazionale O(n).

Esempio: per questo albero binario la risposta deve essere 2



Esercizio svolto 5 (2)

Soluzione: Eseguiamo una visita in post-ordine, perché ogni nodo restituisce al padre informazioni sulle foglie del suo sottoalbero:

```
def es(p):
    if p.left == p.right == None: return 0
    if p.left == None: return es(p.right) + 1
    if p.right == None: return es(p.left) + 1
    return min(es(p.right), es(p.right)) +1
```

Il costo è quello di una visita, e quindi Θ(n).

Esercizio svolto 6 (1)

Esercizio 6. Sia dato un ABR con n nodi memorizzato tramite record e puntatori left e right (non c'è il puntatore al padre).

Progettare una procedura ricorsiva che, dato il puntatore alla radice ed un valore k ≥ 1, restituisca il valore del nodo che avrebbe posizione k nell'elenco delle chiavi ordinate.

Si deve restituire `None` nel caso l'albero abbia meno di k nodi.

L'algoritmo deve avere costo computazionale O(n).

Esercizio svolto 6 (2)

Idea 1.

- Ricordando che la visita in-oreder su un ABR produce la sequenza ordinata, si effettua tale visita
- si inseriscono le chiavi via via incontrate in un array A
- se A ha almeno k elementi, si restituisce il valore A[k − 1].

Questo algoritmo ha costo Θ(n) ed utilizza un array d'appoggio, quindi memoria aggiuntiva dell'ordine di Θ(n).

Possiamo fare meglio di così?

Esercizio svolto 6 (3)

Idea 2.

- Per risparmiare la memoria aggiuntiva, possiamo contare gli elementi anziché salvare in un array.
- Possiamo fermarci dopo aver stampato il k-esimo elemento

Il costo computazionale è O(k).

Esercizio svolto 6 (4)

```
def es(p,k, i=0):
    if i == k: return k
    if p == None:
        return i
    i = es(p.left,k, i )
    if i == k: return k
    if i == k-1:
        print(p.key)
        return k
    return es(p.right,k,i+1)
```

Esercizio svolto 7 (1)

Esercizio 7. Siano dati due max-heap H_1 ed H_2, contenenti rispettivamente n_1 ed n_2 chiavi.

Si progetti un algoritmo che prenda in input i due array su cui sono memorizzati H_1 ed H_2 e restituisca in output un nuovo array con n_1 + n_2 elementi su cui sia memorizzato un nuovo max-heap che contiene le chiavi di H_1 ed H_2. L'algoritmo dovrebbe essere lineare nella somma del numero dei nodi dei due alberi.

Esercizio svolto 7 (2)

IDEA 1. Analogamente a quanto fatto per un esercizio simile con gli ABR, potremmo pensare di ricopiare nel nuovo array uno dei due heap e poi aggiungere, uno alla volta, gli elementi dell'altro.

Poiché questi elementi sono aggiunti come foglie, si può riaggiustare l'heap utilizzando la `heapify_bottom_up()`.

Questa soluzione costa troppo perché ogni inserimento può richiedere un riaggiustamento di costo logaritmico.

Esercizio svolto 7 (3)

IDEA 2. Sappiamo che `Buildheap()` ha un costo lineare e trasforma un generico array in uno heap…

Copiamo i due heap H_1 ed H_2 in un unico nuovo array, uno dopo l'altro così come sono: costo $\Theta(n_1)+ \Theta(n_2)$

Chiamiamo `Buildheap()` su questo nuovo array:

costo $\Theta(n_1+ n_2)$.

NOTA: questo è uno di quei casi in cui non abbiamo bisogno di usare le ipotesi dell'input!

PSEUDOCODICE PER ESERCIZIO

Esercizio svolto 8 (1)

Esercizio 8. Siano dati due ABR T_1 con n_1 nodi e T_2 con n_2 nodi.

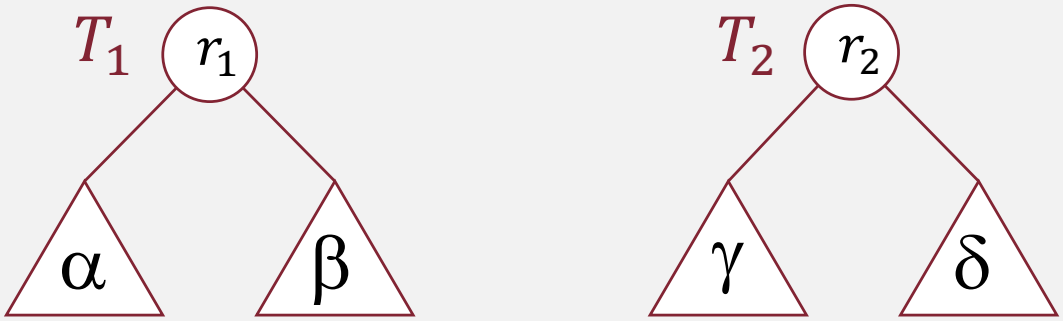
T_1 e T_2 sono memorizzati tramite record e puntatori; inoltre T_i , i = 1, 2, è costituito da una radice r_i , dal suo sottoalbero sinistro T_i^{sx} e dal suo sottoalbero destro T_i^{dx} con la proprietà che per ogni quaterna di chiavi:

$$k_1^{sx} \text{ in } T_1^{sx},\ k_1^{dx} \text{ in } T_1^{dx},\ k_2^{sx} \text{ in } T_2^{sx},\ k_2^{dx} \text{ in } T_2^{dx}$$

vale che $k_1^{sx} < k_2^{sx} < k_1^{dx} < k_2^{dx}$.

Si progetti un algoritmo con costo lineare nell'altezza dei due alberi che fonda T_1 e T_2 in un unico ABR con $n_1 + n_2$ nodi.

Esercizio svolto 8 (2)



$$\alpha < \gamma < \beta < \delta$$

Esercizio svolto 8 (3)

IDEA 1. Si potrebbe pensare di inserire in uno dei due alberi tutte le chiavi dell'altro, visitandone uno e chiamando per ogni nodo visitato la `ABR_insert()` sull'altro albero.

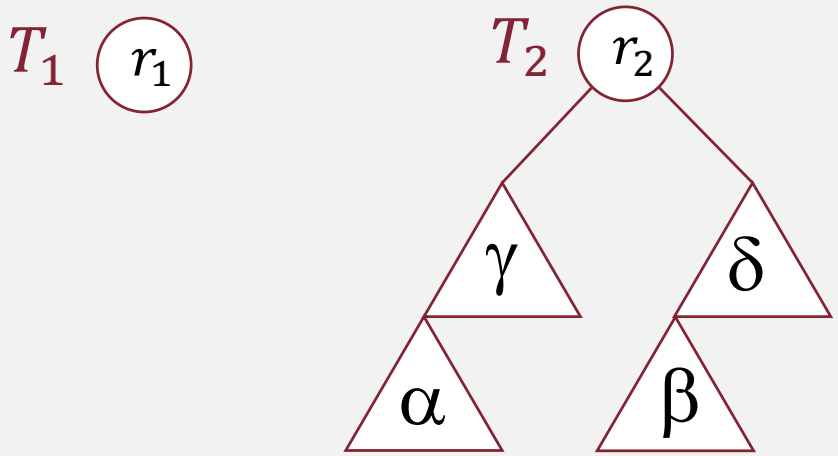
Questa soluzione costa troppo, perché va speso per ciascun inserimento un costo lineare nell'altezza.

IDEA 2. Dobbiamo sfruttare il fatto che conosciamo le relazioni di grandezza fra i sottoalberi.

Ma come???

Esercizio svolto 8 (4)

1. Trova il minimo di γ e appendigli α: costo O(h_2);
2. trova il minimo di δ e appendigli β: costo O(h_2);



3. Rimangono da fare due cose:
   - Inserire r_1 che è rimasto fuori.
   - sistemare eventualmente r_2 (che potrebbe non rispettare la proprietà di ordinamento orizzontale con i nodi in β).

Esercizio svolto 8 (5)

1. Inseriamo r_1 con `ABR_insert()`, costo: O(h_1+h_2)
2. Eliminiamo r_2 dalla posizione di radice:
   - salviamola in una variabile d'appoggio, costo Θ(1);
   - sostituiamola con il suo predecessore cioè il massimo di γ, costo: O(h_2);
   - (non serve riaggiustare perché $\alpha < \gamma < \beta < \delta$)
3. Inseriamo r_2 con `ABR_insert()`, costo: O(h_1+h_2)