Domanda A (6 punti)

1. Dare la definizione della notazione Ω, cioé, date due funzioni $f(n)$ e $g(n)$, definire il significato della notazione $f(n) = \Omega(g(n))$.
2. Ordinare le seguenti funzioni per ordine di grandezza decrescente, cioé scrivere le funzioni secondo un ordine $f_1, f_2, \ldots, f_8$ tale che risulti $f_1 = \Omega(f_2), f_2 = \Omega(f_3), \ldots, f_7 = \Omega(f_8)$.

$$n^{2/3} \qquad 10 \qquad \frac{n}{\sqrt{n}} \qquad 1.1^n \qquad \frac{n}{2^n} \qquad n^2 \qquad \sqrt{\log n} \qquad \log n$$

1. Definizione di Ω(g(n))

Scrittura formale standard:

Diciamo che **f(n) = Ω(g(n))** se e solo se:

$\exists\, c > 0, \exists\, n_0 \geq 0 : \forall\, n \geq n_0, 0 \leq c \cdot g(n) \leq f(n)$

Interpretazione: f(n) è limitata inferiormente da g(n) asintoticamente (a meno di costanti moltiplicative).

---

2. Ordinamento delle Funzioni - Metodo del Corso

Approccio sistematico:

1. **Classificare per famiglia di crescita**
2. **All'interno di ogni famiglia, confrontare gli esponenti**
3. **Verificare casi limite** (funzioni che tendono a 0 o costanti)

Passo 1: Classificazione

Funzioni esponenziali: 1.1^n

Funzioni polinomiali: n^2, n^(2/3), √n = n^(1/2)

Funzioni logaritmiche: log n, √(log n) = (log n)^(1/2)

Funzioni costanti: 10

Funzioni che tendono a 0: n/2^n

Passo 2: Gerarchia Standard

Per il corso, la gerarchia è:

Esponenziali > Polinomiali > Logaritmiche > Costanti > Infinitesimi

Passo 3: Ordinamento Interno

Polinomiali: confronta gli esponenti

- n^2 (esponente 2)
- n^(2/3) (esponente 2/3)
- n^(1/2) (esponente 1/2)

Quindi: n^2 > n^(2/3) > √n

Logaritmiche: confronta gli esponenti

- log n = $(\log n)^1$
- (log n)^(1/2)

Quindi: log n > √(log n)

Scrittura su Carta (Formato Esame)

```
f₁ = 1.1ⁿ
f₂ = n²
f₃ = n^(2/3)
f₄ = n/√n = √n
f₅ = log n
f₆ = √(log n)
f₇ = 10
f₈ = n/2ⁿ
```

Verifiche (se richieste):

- $f_1 = \Omega(f_2)$ poiché 1.1^n domina n^2 (esponenziale > polinomiale)
- $f_2 = \Omega(f_3)$ poiché n^2 domina n^(2/3) (2 > 2/3)
- $f_3 = \Omega(f_4)$ poiché n^(2/3) domina n^(1/2) (2/3 > 1/2)
- $f_4 = \Omega(f_5)$ poiché √n domina log n (polinomiale > logaritmico)
- $f_5 = \Omega(f_6)$ poiché log n domina (log n)^(1/2) (1 > 1/2)
- $f_6 = \Omega(f_7)$ poiché (log n)^(1/2) → ∞ mentre 10 è costante
- $f_7 = \Omega(f_8)$ poiché 10 > $n/2^n$ definitivamente ($n/2^n$ → 0)

Nota critica: $n/2^n$ è l'unica funzione che tende a 0, quindi va sempre ultima.

Esercizio 2 (11 punti) Una *longest common substring* di due stringhe X e Y è una sottostringa di X e di Y di lunghezza massima. Si vuole progettare un algoritmo efficiente per calcolare la lunghezza di una longest common substring. Per semplicità si assuma che entrambe le stringhe di input abbiano stessa lunghezza n.

(a) Qual è la complessità dell'algoritmo esaustivo che analizza tutte le possibili sottostringhe comuni?

(b) Assumendo di conoscere un algoritmo che determina se una stringa di m caratteri è sottostringa di un'altra stringa di n caratteri in tempo $O(m+n)$, come si può modificare l'algoritmo del punto precedente per renderlo più efficiente?

(c) Progettare un algoritmo di programmazione dinamica più efficiente di quello del punto precedente. Sono richiesti relazione di ricorrenza sulle lunghezze (senza dimostrazione) e algoritmo bottom-up. (Suggerimento: considerare la lunghezza della longest common substring dei prefissi $X_i = \langle x_1, \dots, x_i \rangle$ e $Y_j = \langle y_1, \dots, y_j \rangle$ *che termina con* x_i *e* y_j, rispettivamente.)

Contesto

Problema: trovare la lunghezza della **longest common substring** (sottostringa comune massima) tra due stringhe X e Y, entrambe di lunghezza n.

Nota: Substring = sottostringa **contigua** (diverso da subsequence).

(a) Complessità dell'Algoritmo Esaustivo

Approccio esaustivo:

1. Enumera tutte le possibili sottostringhe di X
2. Per ognuna, verifica se è sottostringa di Y
3. Tieni traccia della massima lunghezza trovata

Analisi:

Numero di sottostringhe di X:

- Sottostringhe di lunghezza 1: n
- Sottostringhe di lunghezza 2: n-1
- ...
- Sottostringhe di lunghezza n: 1

Totale: n + (n-1) + ... + 1 = **n(n+1)/2 = $\Theta(n^2)$** sottostringhe

Verifica se una sottostringa è in Y:

- Per ogni sottostringa di lunghezza k, scorrere Y richiede O(n·k) nel caso peggiore
- Confronto di stringhe: O(k)

Complessità totale:

- Per tutte le sottostringhe: Σ(k=1 to n) (n-k+1)·O(n·k)
- Semplificando: **O(n^4)**

Risposta formale su carta:

```
L'algoritmo esaustivo ha complessità O(n⁴):
- Genera Θ(n²) sottostringhe di X
- Per ogni sottostringa di lunghezza k, verifica se appare in Y in O(n·k)
- Nel caso peggiore: Σ(k=1 to n) O(n·k) = O(n⁴)
```

(b) Ottimizzazione con Algoritmo O(m+n)

Problema: Dato un algoritmo che verifica se una stringa di m caratteri è sottostringa di una stringa di n caratteri in O(m+n) (es. KMP, Rabin-Karp).

Ottimizzazione:

Invece di verificare ogni sottostringa singolarmente, usiamo **binary search sulla lunghezza**:

1. **Idea:** Se esiste una common substring di lunghezza k, allora esiste anche di lunghezza k-1
2. **Ricerca binaria:** sulla lunghezza della substring (da 1 a n)
3. **Verifica:** per una lunghezza k fissata, verifica tutte le n-k+1 sottostringhe di X di lunghezza k

Algoritmo:

```
LCS-BinarySearch(X, Y, n):
    low = 1, high = n, result = 0

    while low ≤ high:
        mid = ⌊(low + high)/2⌋

        if ExistsCommonSubstring(X, Y, mid):
            result = mid
            low = mid + 1
        else:
            high = mid - 1

    return result

ExistsCommonSubstring(X, Y, k):
    for i = 1 to n-k+1:
        S = X[i..i+k-1]  // sottostringa di X di lunghezza k
        if IsSubstring(S, Y):  // O(k + n)
```

```
        return true
    return false
```

Complessità:

- Ricerca binaria: O(log n) iterazioni
- Per ogni iterazione con lunghezza k:
  - Verifica O(n) sottostringhe di X
  - Ogni verifica: O(k + n)
  - Totale per iterazione: $O(n \cdot (k + n)) = O(n^2)$ nel caso peggiore ($k \leq n$)

Complessità totale: $O(n^2 \log n)$

Risposta formale su carta:

```
Utilizzo binary search sulla lunghezza della substring:
1. Cerco binariamente la massima lunghezza k ∈ [1, n]
2. Per ogni k, verifico se esiste common substring di lunghezza k:
   - Genero n-k+1 sottostringhe di X di lunghezza k
   - Per ognuna verifico se è in Y usando l'algoritmo O(k+n)
3. Complessità: O(log n) iterazioni × O(n·(k+n)) = O(n² log n)
```

(c) Programmazione Dinamica

Definizione della ricorrenza:

Sia **L[i][j]** = lunghezza della longest common substring che **termina** in X[i] e Y[j].

Ricorrenza:

```
L[i][j] = {
    0                    se i = 0 oppure j = 0
    L[i-1][j-1] + 1     se X[i] = Y[j]
    0                    se X[i] ≠ Y[j]
}
```

Soluzione finale: $\max\{L[i][j] : 1 \leq i, j \leq n\}$

Intuizione: Se X[i] = Y[j], posso estendere la substring comune che terminava in X[i-1] e Y[j-1]. Altrimenti, la substring comune che termina esattamente qui ha lunghezza 0 (perché deve essere **contigua**).

Algoritmo Bottom-Up

```
LCS-DP(X, Y, n):
    Alloca matrice L[0..n][0..n]
    maxLen = 0

    // Inizializzazione
    for i = 0 to n:
        L[i][0] = 0
    for j = 0 to n:
        L[0][j] = 0

    // Riempimento
    for i = 1 to n:
        for j = 1 to n:
            if X[i] = Y[j]:
                L[i][j] = L[i-1][j-1] + 1
                maxLen = max(maxLen, L[i][j])
            else:
                L[i][j] = 0

    return maxLen
```

Complessità:

- Tempo: **$\Theta(n^2)$** (due cicli annidati)
- Spazio: **$\Theta(n^2)$** (matrice $n \times n$)

Hint: Prefissi

Interpretazione con prefissi:

Sia $X_i = \langle x_1, ..., x_i \rangle$ e $Y_j = \langle y_1, ..., y_j \rangle$

L[i][j] rappresenta la lunghezza della longest common substring tra Xi e Yj **che termina esattamente in xi e yj**.

Se $x_i \neq y_j$, qualsiasi substring comune non può includere contemporaneamente xi e yj come ultimi caratteri, quindi L[i][j] = 0.

Risposta formale su carta:

```
Programmazione Dinamica:

Definizione: L[i][j] = lunghezza LCS che termina in X[i] e Y[j]

Ricorrenza:
L[i][j] = {
    0                     se i = 0 ∨ j = 0
    L[i-1][j-1] + 1      se i,j > 0 ∧ X[i] = Y[j]
```

```
    0                    se i,j > 0 ∧ X[i] ≠ Y[j]
}

Soluzione: max{L[i][j] : 1 ≤ i,j ≤ n}

Algoritmo: riempi matrice L bottom-up, Θ(n²) tempo e spazio.
```

Domande

Domanda A (7 punti) Si dia la definizione di limite asintotico stretto. Data la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ 2T(n/5) + T(n/2) + n & \text{se } n > 1 \end{cases}$$

mostrare che $f(n) = n$ è limite asintotico stretto per la soluzione.

Definizione di Limite Asintotico Stretto

$f(n)$ è **limite asintotico stretto** per $T(n)$ se:

$T(n) = \Theta(f(n))$

cioè se esistono $c_1, c_2 > 0$ e $n_0 \geq 0$ tali che:

$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$ per ogni $n \geq n_0$

Equivalentemente: $T(n) = O(f(n)) \wedge T(n) = \Omega(f(n))$

Soluzione della Ricorrenza

Data:

```
        ⎧ 1                        se n = 1
T(n) = ⎨
        ⎩ 2T(n/5) + T(n/2) + n    se n > 1
```

Obiettivo: Mostrare che $T(n) = \Theta(n)$

Dimostrazione: $T(n) = \Theta(n)$

Metodo: Sostituzione (Induzione)

Passo 1: Dimostrare $T(n) = O(n)$

Ipotesi: $\exists\, c_2 > 0$ tale che $T(n) \leq c_2 \cdot n$ per $n \geq n_0$

Caso base: $n = 1$, $T(1) = 1 \le c_2 \cdot 1$ (verificato per $c_2 \ge 1$)

Passo induttivo: Assumiamo $T(k) \le c_2 \cdot k$ per ogni $k < n$

$$
\begin{aligned}
T(n) &= 2T(n/5) + T(n/2) + n \\
&\le 2 \cdot c_2 \cdot (n/5) + c_2 \cdot (n/2) + n \qquad \text{[per ipotesi induttiva]} \\
&= (2c_2/5 + c_2/2) \cdot n + n \\
&= (4c_2/10 + 5c_2/10) \cdot n + n \\
&= (9c_2/10) \cdot n + n \\
&= (9c_2/10 + 1) \cdot n
\end{aligned}
$$

Vogliamo: $(9c_2/10 + 1) \cdot n \le c_2 \cdot n$

$$
\begin{aligned}
&9c_2/10 + 1 \le c_2 \\
&1 \le c_2 - 9c_2/10 \\
&1 \le c_2/10 \\
&c_2 \ge 10
\end{aligned}
$$

Conclusione: Scegliendo $c_2 = 10$, si ha $T(n) \le 10n$, quindi **$T(n) = O(n)$**

Passo 2: Dimostrare $T(n) = \Omega(n)$

Ipotesi: $\exists\, c_1 > 0$ tale che $T(n) \ge c_1 \cdot n$ per $n \ge n_0$

Caso base: $n = 1$, $T(1) = 1 \ge c_1 \cdot 1$ (verificato per $c_1 \le 1$)

Passo induttivo: Assumiamo $T(k) \ge c_1 \cdot k$ per ogni $k < n$

$$
\begin{aligned}
T(n) &= 2T(n/5) + T(n/2) + n \\
&\ge 2 \cdot c_1 \cdot (n/5) + c_1 \cdot (n/2) + n \qquad \text{[per ipotesi induttiva]} \\
&= (2c_1/5 + c_1/2) \cdot n + n \\
&= (9c_1/10) \cdot n + n \\
&= (9c_1/10 + 1) \cdot n
\end{aligned}
$$

Vogliamo: $(9c_1/10 + 1) \cdot n \ge c_1 \cdot n$

$$
\begin{aligned}
&9c_1/10 + 1 \ge c_1 \\
&1 \ge c_1 - 9c_1/10 \\
&1 \ge c_1/10 \\
&c_1 \le 10
\end{aligned}
$$

Conclusione: Scegliendo $c_1 = 1$, si ha $T(n) \ge 1 \cdot n$, quindi **$T(n) = \Omega(n)$**

Conclusione

Avendo dimostrato che:

- T(n) = O(n) con c_2 = 10
- T(n) = Ω(n) con c_1 = 1

Concludiamo che:

T(n) = Θ(n)

Quindi **f(n) = n** è il limite asintotico stretto per la soluzione.

Esercizio 1 (7 punti) Realizzare una procedura `BST(A)` che dato un array `A[1..n]` di interi, ordinato in modo crescente, costruisce un albero binario di ricerca di altezza minima che contiene gli elementi di `A` e ne restituisce la radice. Fornire un'argomentazione che supporti il fatto che l'albero ha altezza minima. Per allocare un nuovo nodo dell'albero si utilizzi una funzione `mknode(k)` che dato un intero `k` ritorna un nuovo nodo con `x.key=k` e figlio destro e sinistro `x.left = x.right = nil`. Valutarne la complessità.

Soluzione:

i. L'implementazione è la seguente:

```
BST(A)
   return BST-rec(A,1,n)

BST-rec(T,A,p,q)
   if p <= q
      m = floor(p+q/2)
      x=mknode(A[m])
      x.l = BST-rec(A,p,m-1)
      x.r = BST-rec(A,m+1,q)
    else
      x = nil

    return x
```

Si può dimostrare, per induzione su n, che l'altezza dell'albero generato è $\lceil \log_2(n+1) \rceil$, quindi è la minima possibile.

ii. Si ottiene la ricorrenza $T(n) = 2T(n/2) + \Theta(1)$. Applicando il master theorem si ottiene quindi come costo $T(n) = O(n)$.

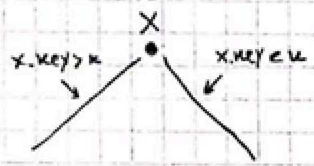
Esercizi

Esercizio 1 (7 punti)

- Scrivere una funzione `Rep(T,k)` che dato un albero di ricerca T e una chiave k ritorna il numero di occorrenze di k in T. Valutare la complessità dell'algoritmo definito.
- Se si assume che l'albero binario di ricerca non contenga chiavi ripetute cosa cambia? (Ovviamente in questo caso il numero di occorrenze è al più uno). Adattare la soluzione e discutere la relativa complessità.

ESERCIZIO 1

Rep(T,k)



Non posso supporre che se ho x.key==k allora posso cercare solo a destra perché sarebbe un problema se avessimo che l'albero deve essere bilanciato!

```
RepRec (x,k) {
    if (x=nil)
        return 0
    else if x.key<k
        return RepRec(x.right, k)
    else if x.key>k
        return RepRec(x.left, k)
    else
        return 1 + RepRec(x.left, k) + RepRec(x.right, k)
}
```

Sono nel caso in cui ho trovato il nodo con chiave uguale a k, devo ricordarmi quindi di averlo trovato (1+) e poi cerco a sinistra e a destra.

```
Rep(T,k) {
    return RepRec(T.root, k)
}
```

COMPLESSITÀ: $T(n) = O(n)$ nel caso pessimo (ovvero quando trovo la chiave k)

Se volessi scriverlo per bene, $T(n) = c + T(n-l-\underset{\text{radice}}{1}) + T(l)$

Soluzione lineare (equivalente ad una visita)

Domanda 3 Realizzare una funzione `Diff(A,k)` che, dato un array `A[1,n]` ordinato in senso crescente, verifica se esiste una coppia di indici `i, j` tali che `A[i]` − `A[j]` = `k`. Restituisce la coppia di indici se esiste e (0,0) altrimenti. La funzione non deve alterare l'input e deve operare in spazio costante. Scrivere lo pseudocodice e valutarne la complessità.

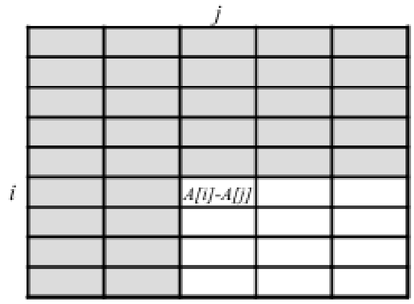
Soluzione:

Il codice può essere:

```
diff(A, n, k):
    i=1
    j=1
    while (i<=n) and (j<=n) and (A[i]- A[j] <> k)
        if (A[i]- A[j] < k)
            i++
        else
            j++

    if (i <= n) and (j<=n)
        return (i,j)
    else
        return (0,0)
```

È facile vedere che si mantiene l'invariante $\forall (i', j') \in [1, n].(i' < i) \vee (j' < j) \Rightarrow A[i] - A[j] \neq k$, ovvero una coppia (i', j') tale che $A[i] - A[j] = k$ può esistere solo tra le coppie ancora esplorabili ($i' \geqslant i$ e $j' \geqslant j$), ovvero, graficamente nella parte non grigia:



Infatti, inizialmente, con $i = j = 1$, l'invariante è vacuamente vero.
Ad ogni iterazione, se entro nel ciclo, ci sono due possibilità:

- Se $A[i] - A[j] < k$, allora incremento i. In questo modo, escludo dall'esplorazione le coppie (i, j') con $j' \geqslant j$ per le quali, dato che l'array è crescente e quindi $A[j] \leqslant A[j']$, vale

  $$A[i] - A[j'] \leqslant A[i] - A[j] < k.$$

  Dunque non escludo coppie utili, l'invariante continua a valere.

- Dualmente, $A[i] - A[j] > k$, allora incremento j. In questo modo, escludo dall'esplorazione le coppie (i', j) con $i' \geqslant i$ per le quali, dato che l'array è crescente e quindi $A[i] \leqslant A[i']$, vale

  $$A[i'] - A[j] \geqslant A[i] - A[j] > k.$$

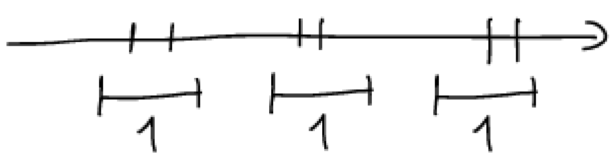
  Dunque, anche in questo caso, non escludo coppie utili, l'invariante continua a valere.

Quando esco dal ciclo, se $A[i] - A[j] = k$, ho concluso con successo. Altrimenti deve essere $i > n$ o $j > n$, che unitamente all'invariante, mi permettono di concludere che per ogni $i, j \in [1, n]$, $A[i] \neq A[j]$, come desiderato.

Da questo la correttezza segue immediatamente. La complessità è lineare. Il numero di iterazioni è pari al più a $2n - 1$, dato che i e j partono da 1, sono limitate da n ed ogni iterazione aumenta una delle due. Dato che ciascuna iterazione ha costo costante, ottengo $T(n) = O(2n - 1) = O(n)$.

Esercizio: Sia $X = \{x_1, x_2, \dots, x_n\}$ un insieme di punti ordinati sulla retta reale. Fornire un algoritmo greedy che determini un insieme I di cardinalità minima di intervalli chiusi di ampiezza unitaria ($[a,b] \in I \Rightarrow b-a=1$) tale che $\forall x_i \in X \ \exists j \in I$ tale che $x_i \in j$.

esempio:



√ 1) da sx a dx, inizia un nuovo intervallo nel primo punto non coperto

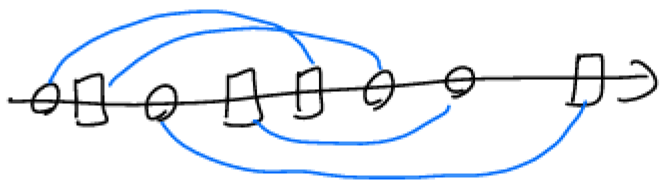
```
MIN_COVER(X)
    n = length(X)
    C = { [x_1, x_1 + 1] }
    last = 1

for i = 2 to n do
    if x_i > x_last + 1 then
        C = C ∪ { [x_i, x_i + 1] }
        last = i
return C
```

Esercizio: matching sulla linea

Sia $S = \{s_1, s_2, \dots, s_n\}$ un insieme di punti ordinati sulla retta reale, rappresentanti dei server. Sia $C = \{c_1, c_2, \dots, c_n\}$ un insieme di punti ordinati sulla retta reale, rappresentanti dei client. Il costo di assegnare un client c_i ad un server s_j è $|c_i - s_j|$. Fornire un algoritmo greedy che assegna ogni client ad un server distinto e che minimizzi il costo totale (equiv., medio) dell'assegnamento.

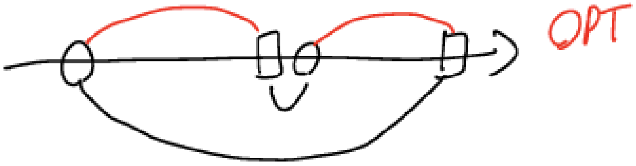
esempio:



$n=4$

~~1)~~ client al server + vicino, partendo dalla coppia client-server con distanza minore

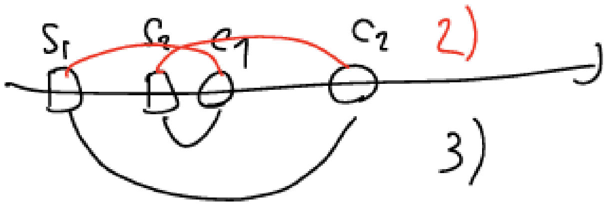
NON funziona:



√ 2) $c_1 - s_1$, $c_2 - s_2$, ...

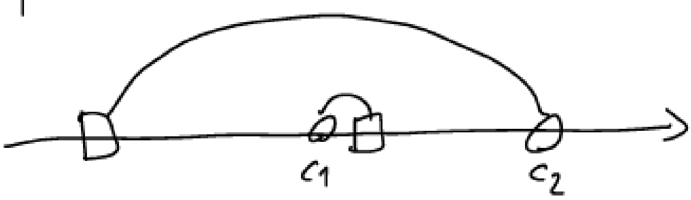
(equiv. $c_n - s_n$, $c_{n-1} - s_{n-1}$, ...)

~~3)~~ C_1 al server + vicino; C_2 al server + vicino



⇒ sono algoritmi ≠

NON funziona:



Esercizio 11 Sia dato un albero i cui nodi contengono una chiave intera $x.key$, oltre ai campi $x.l$, $x.r$ e $x.p$ che rappresentano rispettivamente il figlio sinistro, il figlio destro e il padre. Si definisce *grado di squilibrio* di un nodo il valore assoluto della differenza tra la somma delle chiavi nei nodi foglia del sottoalbero sinistro e la somma delle chiavi dei nodi foglia del sottoalbero destro. Il grado di squilibrio di un albero è il massimo grado di squilibrio dei suoi nodi.

Fornire lo pseudocodice di una funzione `sdegree(T)` che calcola il grado di squilibrio dell'albero T (si possono utilizzare funzioni ricorsive di supporto). Valutare la complessità della funzione.

Soluzione:

```
// computes the sum of the leaf nodes of the subtree and the sdegree
// for the node x (returns two values)

sdegree(x)

  if (x == nil)
     sum = 0
     degree = 0
  elif (x.left == nil) and (x.right = nil)      # leaf
     sum = x.key
     degree = 0
  else
     suml, degreel = sdegree(x.l)
     sumr, degreer = sdegree(x.r)
     sum = suml + sumr
     degree = max { degreel, degreel, abs(suml - sumr) }

  return sum, degree
```