

## Esercizi

**Esercizio 1 (10 punti)** Sia dato un array  $V[1..n]$  i cui valori rappresentano la variazione giornaliera del valore di un titolo azionario. È noto che il titolo è stato prima perdita, con valori sempre negativi, poi ha iniziato a oscillare in giorni *consecutivi* tra valori positivi e negativi, e infine si è stabilizzato su valori positivi (dunque nella sequenza non ci possono essere due giorni positivi seguiti da un negativo). Realizzare un algoritmo *divide et impera*  $\text{Split}(V)$  che individua il giorno in cui il titolo ha iniziato a essere stabile su valori positivi, ovvero il minimo indice  $i \in [1, n]$  tale che per ogni  $j \geq i$  vale  $V[j] > 0$ . Se il titolo non si stabilizza su valori positivi, ritornare 0. Ad es., se l'array  $V = [-1, -2, 2, -1, 6, 3]$  l'indice da tornare sarà 5, mentre invece per  $V = [-1, -2, 2, -1, 6, -3]$  si ritornerà 0. Fornire lo pseudocodice di  $\text{Split}(V)$ , motivarne la correttezza e individuarne la complessità. Si assuma che non ci siano valori nulli.

$\text{Split}(V)$

return  $\text{Split\_rec}(V, 1, n)$

$\text{Split\_rec}(V, p, r)$

$q = \lfloor \frac{p+r}{2} \rfloor$   
 $\min = A[q]$

if( $\min < A[p]$ )  
     $\text{Split\_rec}(A, p, q)$   
else  
     $\text{Split\_rec}(A, q + 1, r)$

-1 -2  $\boxed{2}$  [-1 6 3]

Ecco l'intuizione per una soluzione divide et impera:

1. Dividiamo l'array in due metà
2. Se nella metà destra troviamo anche un solo valore negativo, il punto di stabilizzazione deve essere dopo
3. Se nella metà destra sono tutti positivi, il punto potrebbe essere nella metà sinistra o nel punto di divisione

NO cercare minimo locale (come sopra) e NO toccare ordine array (perdiamo stabilizzazione)

```
mid = ⌊(left + right)/2⌋

// Verifica stabilizzazione metà destra
stable_right = true
for i = mid + 1 to right do
    if V[i] ≤ 0 then
        stable_right = false
        break

    if not stable_right then
        // Se la metà destra non è stabile, la soluzione è lì
        return Split_Aux(V, mid + 1, right)
    else
        // Se la metà destra è stabile, verifichiamo la sinistra
        result = Split_Aux(V, left, mid)
        if result = 0 then
            return mid + 1 // Il punto di stabilizzazione è l'inizio della metà destra
        else
            return result
```

**Esercizio 2** (8 punti) Date due stringhe  $X = \langle x_1, x_2, \dots, x_m \rangle$  e  $Y = \langle y_1, y_2, \dots, y_n \rangle$ , si consideri la seguente quantità  $\ell(i, j)$ , definita per ogni coppia di valori  $i, j$  con  $0 \leq i \leq m$  e  $0 \leq j \leq n$ :

$$\ell(i, j) = \begin{cases} 1 & \text{se } i = 0 \text{ o } j = 0 \\ 3\ell(i, j - 1) & \text{se } i, j > 0 \text{ e } x_i = y_j \\ 2\ell(i - 1, j - 1) - \ell(i - 1, j) & \text{se } i, j > 0 \text{ e } x_i \neq y_j. \end{cases}$$

Si vuole calcolare la quantità  $q = \max\{\ell(i, j) : 0 \leq i \leq m, 0 \leq j \leq n\}$ .

(a) Scrivere un algoritmo bottom-up per il calcolo di  $q$ .

(b) Determinare la complessità esatta dell'algoritmo, supponendo che le uniche operazioni di costo unitario e non nullo siano i confronti tra caratteri.

```

COMPUTE(X, Y)
    for i = 1 to m - 1
        for j = n - 1 downto i
            if x_i == y_j
                l[i, j] = 3 * l[i, j-1]
                q = max(q, l[i, j])
            else
                l[i, j] = 2 * l[i - 1, j - 1]
                - l[i - 1, j]
                q = max(q, l[i, j])
    return q;

```

$$\sum_{i=1}^{m-1} \sum_{j=i}^{n-1} 1 = \sum_{i=1}^{m-1} (n-1-i) = \sum_{k=1}^{m-1} k = \frac{(m-1)(n-1)}{2}$$

**Domanda A** (8 punti) Si consideri la seguente funzione ricorsiva con argomento un intero  $n \geq 0$

```

val(n)
    if n <= 2
        return 1
    else
        [ return val(n-1) + val(n-2) + val(n-2) ]

```

$$T(n) = T(n-1) + T(n-2) + T(n-2)$$

Determinare la ricorrenza che esprime la complessità della funzione e mostrare che la soluzione è  $\Omega(2^n)$ . La complessità è anche  $O(2^n)$ ? Motivare le risposte.

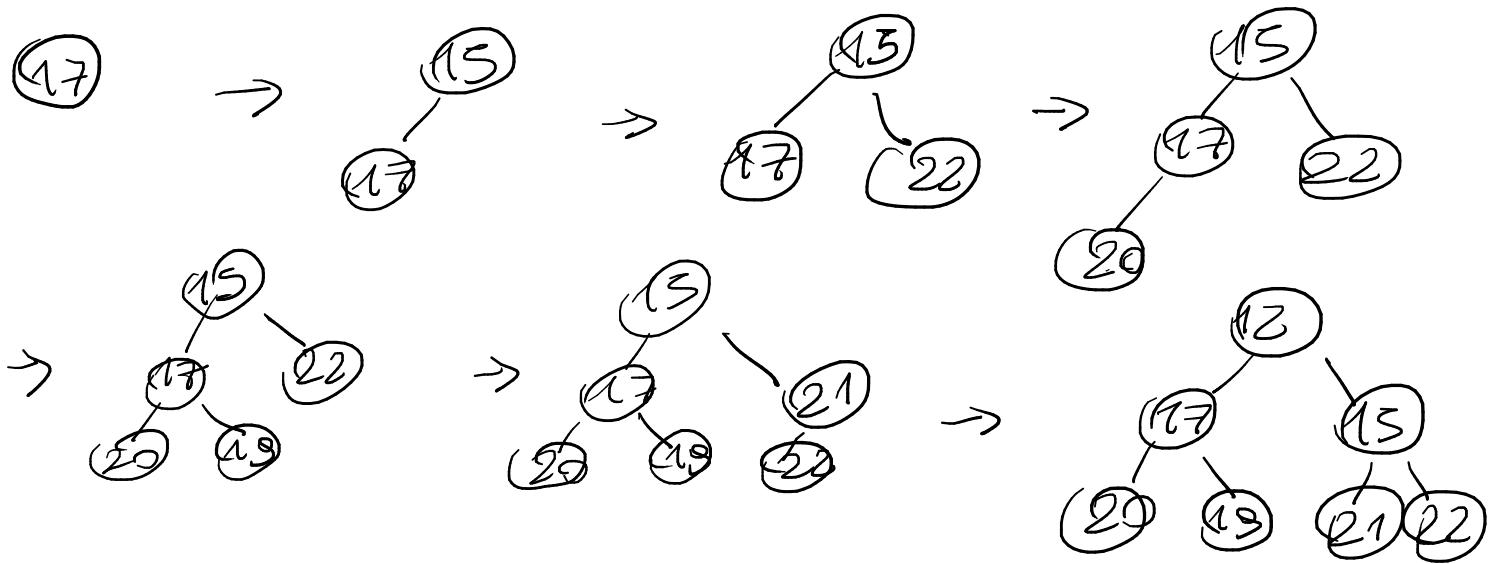
$$T(n) = T(n-1) + 2T(n-2) \quad \dots \quad \left\{ \begin{array}{l} T(n) \geq \Theta(2^n) \\ T(n) \leq C_2^n \end{array} \right.$$

$$T(n) \geq \Theta(2^n)$$

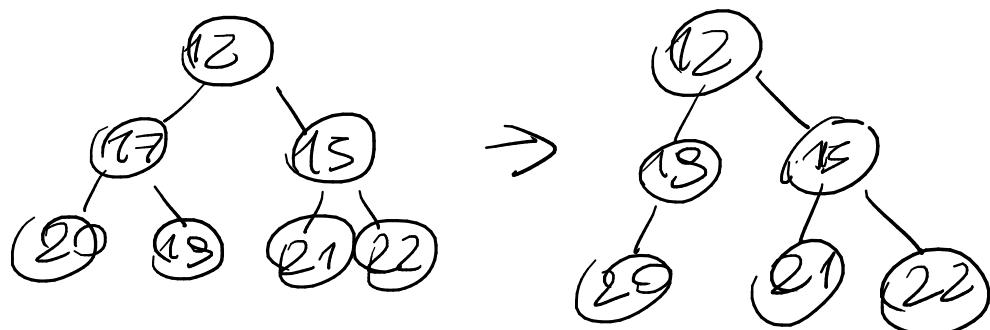
↓ IND. FORNE

$$T(n-1) \geq \Theta(2^{n-1})$$

**Domanda B** (7 punti) Dare la definizione di min-heap. Data la sequenza di elementi 17, 15, 22, 20, 19, 21, 12, si specifichi il min-heap ottenuto inserendo, a partire da uno heap vuoto, uno alla volta questi elementi nell'ordine indicato e infine rimuovendo 17. Si descriva sinteticamente come si procede per arrivare al risultato.



RIMOZIONE DI 17

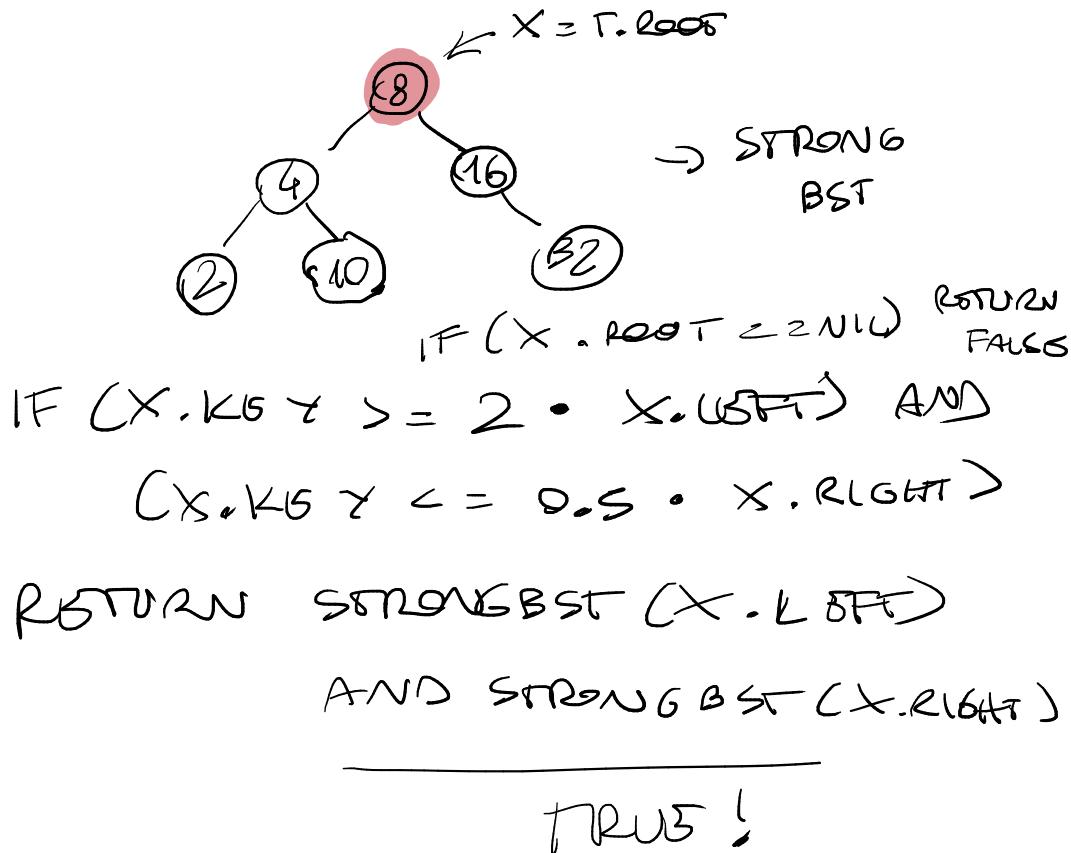


**Domanda B** (6 punti) Si calcoli la lunghezza della longest common subsequence (LCS) tra le stringhe *armo* e *oro*, calcolando tutta la tabella  $L[i, j]$  delle lunghezze delle LCS sui prefissi usando l'algoritmo visto in classe.

	O	R	O
O	0 0	0 0	0 0
A	0 0	0 0	0 0
R	0 0	0 0	1
M	0 0	0 0	1
O	0 0	1	2

UICUP  
MATOR  
NON C'È STANDO

Esercizio 1 (9 punti) Realizzare una funzione `strongBST(T)` che dato un albero binario  $T$  con chiavi numeriche non negative, verifica se, per ogni nodo, la chiave è maggiore uguale del doppio di ogni chiave nel sottoalbero sinistro e minore o uguale della metà di ogni chiave nel sottoalbero destro, e ritorna conseguentemente un valore booleano (la radice dell'albero è  $T.root$  e ogni nodo  $x$  ha i campi  $x.left$  e  $x.right$  e  $x.key$ ). Valutarne la complessità.



```

# strongBSTRec(x):
# verifica se il sottoalbero radicato in x e' uno strongBST e ritorna tre valori:
# - un booleano (che indica l'esito della verifica)
# - il massimo delle chiavi nel sottoalbero sx
# - il minimo delle chiavi nel sottoalbero dx

strongBSTRec(x)

if x = nil
    return true, 0, +infinity
else
    # ispeziona i sottoalberi sinistro e destro
    sl, ml, Ml = strongBSTRec(x.left)
    sr, mr, Mr = strongBSTRec(x.right)

    # se la chiave del nodo in esame e' maggiore o uguale del doppio
    # del massimo delle chiavi nel sottoalbero sinistro (quindi di
    # tutte le chiavi nel sottoalbero sinistro) e minore o uguale della
    # meta' del minimo delle chiavi nel sottoalbero destro (quindi di
    # tutte le chiavi nel sottoalbero destro) delle chiavi dei
    # discendenti, ritorna true
    s = (x.key >= 2*Ml) and (x.key <= 1/2 mr)

    # calcola il minimo e il massimo delle chiavi nel sottoalbero radicato in x
    m = min { ml, mr, x.key }
    M = max { Ml, Mr, x.key }

return s, m, M

```

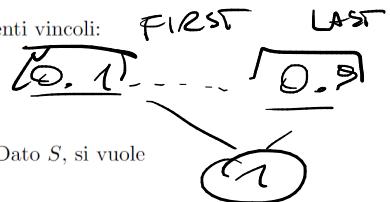
Si tratta di una visita e quindi la complessità è  $\Theta(n)$ .

**Esercizio 2** (10 punti) Dato un insieme di  $n$  numeri reali positivi e distinti  $S = \{a_1, a_2, \dots, a_n\}$ , con  $0 < a_i < a_j < 1$  per  $1 \leq i < j \leq n$ , un  $(2,1)$ -boxing di  $S$  è una partizione  $P = \{S_1, S_2, \dots, S_k\}$  di  $S$  in  $k$

- SORT

sottoinsiemi (cioè,  $\bigcup_{j=1}^k S_j = S$  e  $S_r \cap S_t = \emptyset, 1 \leq r \neq t \leq k$ ) che soddisfa inoltre i seguenti vincoli:

$$|S_j| \leq 2 \quad \text{e} \quad \sum_{a \in S_j} a \leq 1, \quad 1 \leq j \leq k.$$



In altre parole, ogni sottoinsieme contiene al più due valori la cui somma è al più uno. Dato  $S$ , si vuole determinare un  $(2,1)$ -boxing che minimizza il numero di sottoinsiemi della partizione.

1. Scrivere il codice di un algoritmo greedy che restituisce un  $(2,1)$ -boxing ottimo in tempo lineare.  
(Suggerimento: si creino i sottoinsiemi in modo opportuno basandosi sulla sequenza ordinata.)
2. Si enunci la proprietà di scelta greedy per l'algoritmo sviluppato al punto precedente e la si dimostri, cioè si dimostri che esiste sempre una soluzione ottima che contiene la scelta greedy.

OPT ≠ FIRST / LAST

GRISORY! → ROUNDS → 2-1 BOXING  
→ ATTIVITÀ → SERVER  
→ PARCOGGI → USCITE

0.4 10.1 10.9 (3 ITOR.)

0.1 10.4 10.9 (2 ITOR.)

ATTIVITÀ → OPT = {A<sub>i</sub>}  
LAST = 1  
FOR i = 2 TO n  
IF A[i].s ≥ A[1].s  
OPT = OPT ∪ A[i]

GREEDY-SEL( $S, f$ )  
1  $n = S.length$   
2  $A = \{a_1\}$   
3  $last = 1$  // indice dell'ultima attività selezionata  
4 **for**  $m = 2$  **to**  $n$   
5     **if**  $s_m \geq f_{last}$   
6          $A = A \cup \{a_m\}$   
7          $last = m$   
8 **return**  $A$

(2,1)-BOXING(S)  
n ← |S|  
P ← empty\_set  
first ← 1  
last ← n  
while (first ≤ last)  
    if (first < last) and  $a_{first} + a_{last} \leq 1$  then  
        P ← P ∪ {{a<sub>first</sub>, a<sub>last</sub>}}  
        first ← first + 1  
    else  
        P ← P ∪ {{a<sub>last</sub>}}  
        last ← last - 1  
return P

0.1 10.1 10.9

FIRST, LAST [OPT]  $\left[ \begin{matrix} S \\ S + A_i \end{matrix} \right] \rightarrow \text{NOT OPT}$

2. La scelta greedy è  $\left[ \begin{matrix} \{a_1, a_n\} & \text{se } n > 1 \text{ e } a_1 + a_n \leq 1, \\ \{a_n\} & \text{altrimenti} \end{matrix} \right]$ . Ora dimostriamo che esiste sempre una soluzione ottima che contiene la scelta greedy. I casi  $n = 1$  e  $a_1 + a_n > 1$  sono banali, visto che in questi casi ogni soluzione ammissibile deve contenere il sottoinsieme  $\{a_n\}$ . Quindi assumiamo che la scelta greedy sia  $\{a_1, a_n\}$ . Consideriamo una qualsiasi soluzione ottima dove  $a_1$  e  $a_n$  non sono accoppiati nello stesso sottoinsieme. Quindi, esistono due sottoinsiemi  $S_1$  e  $S_2$ , con  $a_1 \in S_1$  e  $a_n \in S_2$ . Sostituiamo questi due sottoinsiemi con  $S'_1 = \{a_1, a_n\}$  (cioè, la scelta greedy) e  $S'_2 = S_1 \cup S_2 \setminus \{a_1, a_n\}$ .  $|S'_2| \leq 2$  e, se  $|S'_2| = 2$ , allora  $S'_2 = \{a_s, a_t\}$  con  $a_s \in S_1$  e  $a_t \in S_2$ . Siccome  $a_t$  era precedentemente accoppiato con  $a_n$ , a maggior ragione può essere accoppiato con  $a_s < a_n$ , quindi la nuova soluzione così creata è ammissibile e ancora ottima.

$$F_i \leq F_{i+1} \quad \text{e} \quad OPT = OPT + f_{F_i - i + 1}$$

SOMMARE  
OPT?

GROSSE DAY → OUCHIO !

**Domanda B** (6 punti) Si consideri una tabella hash di dimensione  $m = 8$ , gestita mediante chaining (liste di trabocco) con funzione di hash  $h(k) = k \bmod m$ . Si descriva in dettaglio come avviene l'inserimento della sequenza di chiavi: 28, 19, 10, 35, 26.

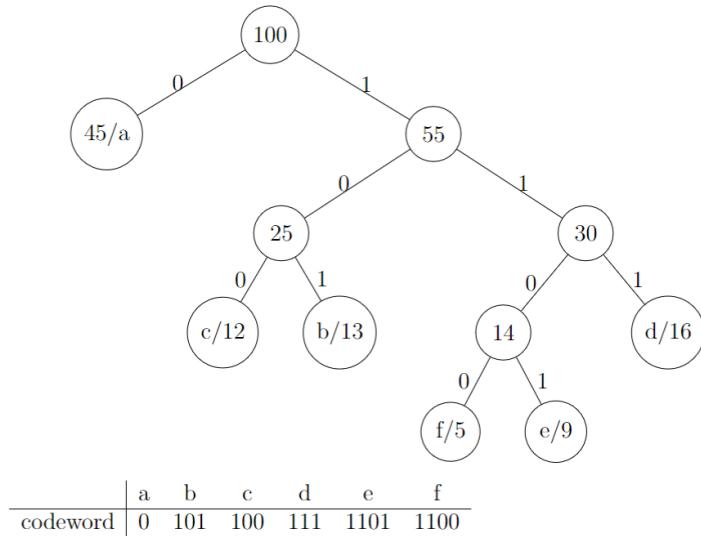
**Soluzione:** La tabella hash  $T$  contiene, in corrispondenza di ciascuna entry  $T[i]$  la lista degli elementi  $x$  tali che  $h(x.key) = i$ . L'inserimento in testa alla lista garantisce complessità dell'inserimento  $O(1)$ .

Si ottiene

0	
1	
2	→ 26 → 10
3	→ 35 → 19
4	→ 28
5	
6	
7	

**Esercizio 1** (10 punti) Realizzare una funzione  $\text{Diff}(A, k)$  che, dato un array  $A[1, n]$  ordinato in senso decrescente, 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, provarne la correttezza e valutarne la complessità.

TYPE ARRAY → WHILE  
 $(1 \leq i \leq n \text{ AND } 1 \leq j \leq n \text{ AND } A[i] - A[j] \geq k)$   
 $i = i - 1$   
 $j = j + 1$

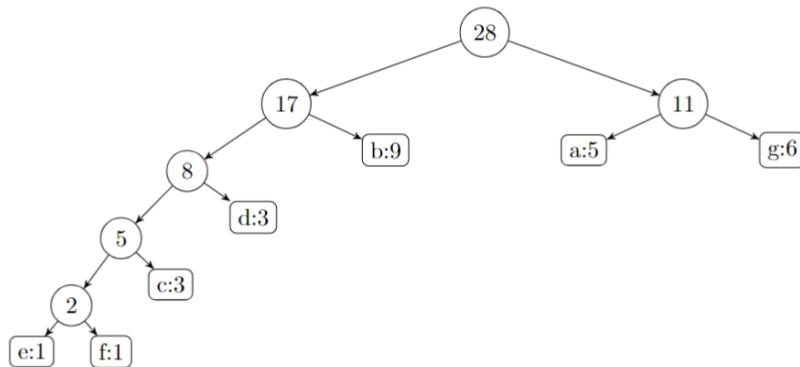


**Domanda 40** Indicare il codice prefisso ottenuto utilizzando l'algoritmo di Huffmann per l'alfabeto  $\{a, b, c, d, e, f, g\}$ , supponendo che ogni simbolo appaia con le seguenti frequenze.

a	b	c	d	e	f	g
5	9	3	3	1	1	6

Spiegare il processo di costruzione del codice.

**Soluzione:**



**Esercizio 1** (9 punti) Si consideri un albero binario  $T$ , i cui nodi  $x$  hanno i campi  $x.l$ ,  $x.r$ ,  $x.p$  che rappresentano il figlio sinistro, il figlio destro e il padre, rispettivamente. Un *cammino* è una sequenza di nodi  $x_0, x_1, \dots, x_n$  tale che per ogni  $i = 0, \dots, n-1$  vale  $x_{i+1}.p = x_i$ . Il cammino è detto *nil-terminated* se  $x_n.l = \text{nil}$  oppure  $x_n.r = \text{nil}$ . Dato un certo  $k$ , diciamo che l'albero è  $k$ -bilanciato se tutti i cammini nil-terminated che iniziano dalla radice hanno lunghezze che differiscono al più di  $k$ . Scrivere una funzione  $\text{bal}(T, k)$  che dato in input l'albero  $T$  e un valore  $k$  verifica se  $T$  è  $k$ -bilanciato e ritorna un corrispondente valore booleano. Valutarne la complessità.

```

bal(T,k)    // verifica se l'albero radicato in x e' k-bilanciato,
            // ovvero i cammini dalla radice terminati da nil hanno
            // lunghezze che differiscono al piu di k

            // si basa su di una funzione ricorsiva che calcola la
            // lunghezza minima e massima dei cammini nil-terminated
            // da un certo nodo.
min,max = nilTerm(T.root)

return max-min <= k

nilTerm(x)
if x = nil
    return (0,0)
else
    (left_min,left_max) = nilTerm(x.left)
    (right_min,r_right_max) = nilTerm(x.right)
    return min(left_min, right_min) + 1, max(left_max,r_right_max) + 1

```

**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  $n$ .

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

**Domanda A** (6 punti)

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

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

**Domanda B** (6 punti) Scrivere la ricorrenza sulle lunghezze  $\ell(i, j)$  per il problema della longest common subsequence (LCS).