$\boxed{17.12}$

Domanda A (6 punti) Dare la definizione formale della classe $O(f(n))$ per una funzione $f(n)$. Mostrare che se $f(n) = O(n^2)$ e $g(n) = O(n)$ allora $f(n) + g(n) = O(n^2)$. ←

O - GRANDE $\left[\begin{array}{l} f \leq c\,g(n) \\ d\,g(n) \leq f \end{array}\right]$

Ω

(O - PICCOLO)

$$\left[\begin{array}{l} O(f(n)) = \{g(n) : \exists c > 0.\ \exists n_0.\ \forall n \geq n_0.\ 0 \leq g(n) \leq c f(n)\} \\ \Omega(f(n)) = \{g(n) : \exists d > 0.\ \exists n_0.\ \forall n \geq n_0.\ 0 \leq d\, f(n) \leq g(n)\} \end{array}\right]$$

$f(n) + g(n) = O(n^2)$

CHAIN

$\rightarrow\ 0 \leq \underbrace{d\, f(n)}_{} \leq g(n) \leq \underbrace{c f(n)}_{}$ $\quad n^2 \quad f(n) = n^2$

$d, c \geq 0 \quad \rightarrow \quad n \geq n_0$

$$\left[\begin{array}{l} 0 \leq f(n) \leq c n^2 \\ 0 \leq d n^2 \leq g(n) \end{array}\right]$$

$n > 0,\ \max\{n_0, n_0\}$] PARSIMONIOSI... (LOGICA)

$0 \leq \dfrac{f(n)}{g(n)} \leq \dfrac{cn^2}{g(n)} \rightarrow \dfrac{cn^2}{d(n)}$

CONCLUSIONE $\Rightarrow \left[\dfrac{c}{d} > 0\right]$

Domanda A (6 punti) Dare la definizione formale delle classi $O(f(n))$ e $\Omega(f(n))$ per una funzione $f(n)$. Mostrare che se $f(n) = O(n^2)$ e $g(n) = \Omega(n)$, con $g(n) > 0$ per ogni n, allora $f(n)/g(n) = O(n)$.

(ES. SIMILE ...)

Soluzione: La classi $O(f(n))$ e $\Omega(g(n))$ sono definite come:

$$O(f(n)) = \{g(n) : \exists c > 0.\ \exists n_0.\ \forall n \geq n_0.\ 0 \leq g(n) \leq cf(n)\}$$
$$\Omega(f(n)) = \{g(n) : \exists d > 0.\ \exists n_0.\ \forall n \geq n_0.\ 0 \leq d\, f(n) \leq g(n)\}$$

Si assuma che $f(n) = O(n^2)$ e $g(n) = \Omega(n)$. Ovvero, esistono $c > 0, n_0$ tali che per ogni $n \geq n_0$

$$0 \leq f(n) \leq cn^2$$

e $d > 0, m_0$ tali che per ogni $n \geq m_0$

$$0 < dn \leq g(n)$$

Quindi, per $n \geq \max\{n_0, m_0\}$ si ha che

$$\begin{aligned} 0 &\leq f(n)/g(n) & \\ &\leq cn^2/g(n) & [\text{poiché } f(n) \leq cn^2] \\ &\leq cn^2/(dn) & [\text{poiché } g(n) \geq dn] \\ &= (c/d)n. & \end{aligned}$$

dato che $c/d > 0$ questo conclude la prova.

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

		S	T	O	R	E
		0	0	0	0	0
S	0	$\boxed{1}$	1	1	1	1
H	0	$\boxed{1}$	$\boxed{1}$	1	1	1
O	0	1	1	$\boxed{2}$	$\boxed{2}$	2
E	0	1	1	2	$\boxed{2}$	3
S	0	2	2	2	2	$\boxed{3}$

LCS = PATTERN MATCHING = MATCH CONTIGUO PIÙ LUNGO

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.

← BRUTEFORCE

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

$M \leq N$

(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?

(n) $M = MAX(\ldots, \ldots)$

$MAX(M, N)$

(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 $X_i = \langle x_1, \ldots, x_i \rangle$ e $Y_j = \langle y_1, \ldots, y_j \rangle$ *che termina con* x_i *e* y_j, rispettivamente.)

X, Y stringhe $X \subseteq Y$

$(n) \leftarrow$ LUNGH. $(n) \cdot (n) \rightarrow (n^2)$

STORE / SHOES

```
FOR I
    FOR J
        C[I,J] = 0
```
← INIT

```
FOR I
    FOR J
        IF Xi = XJ
            → M = MAX(C[I,J], M)
              (↖)
        ELSE
            (↑ / ←)
```
// SAVE NEW SOL

LCS(X, Y)

```
m = X.[length]
n = Y.length
for i = 0 to m
    L[i,0] = 0
for j = 0 to n
    L[0,j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i = y_j
            L[i,j] = L[i-1,j-1] + 1
            B[i,j] = '↖'
        else if L[i-1,j] ≥ L[i,j-1]
            L[i,j] = L[i-1,j]
            B[i,j] = '↑'
        else
            L[i,j] = L[i,j-1]
            B[i,j] = '←'
return (L[m,n], B)
```

INIT

$$l(i,j) = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \quad \text{(caso 0)} \\ l(i-1,j-1)+1 & \text{se } i,j > 0 \text{ e } x_i = x_j \quad \text{(caso 1)} \\ \max\{l(i,j-1), l(i-1,j)\} & \text{se } i,j > 0 \text{ e } x_i \neq x_j \quad \text{(caso 2)} \end{cases}$$

PROG. DINAMICA

Esercizio 1 (10 punti) Un ricco possidente deve lasciare in eredità le sue $2n$ proprietà a n figli. Il valore delle proprietà è contenuto in un array di numeri (reali positivi) `A[1..2n]`.

1. Sviluppare un algoritmo `Split(A,n)` che dato in input l'array `A[1..2n]` dei valori delle proprietà e n, verifica se le proprietà possano essere partizionate in n coppie, tutte con lo stesso valore complessivo (di modo da assegnare una coppia di proprietà a ciascun figlio). Si assume $n > 0$.

   Ad esempio, con $n = 2$, se `A=[1,3,4,2]` la risposta è positiva visto che le proprietà possono essere partizionate nelle coppie `1,4` e `3,2`, mentre se `A=[1,3,5,2]` la risposta è negativa.
2. Si valuti la complessità dell'algoritmo proposto.
3. Si dimostri la correttezza dell'algoritmo proposto.

→ 1, 2, 3, 4

_SPLIT(A) → // 1 3 4 2

① SORT(A) // 1 2 3 4

→ 1, 2, 3, 4   1 2 3 4 5 6 7 8

② MERGE_SUM(A, 1, N)

MERGE-SUM(A, P, Q)

① IF(P < R)

② $Q = \lfloor P + R \rfloor / 2$

③ S_1 = MERGE-SUM(A, P, Q)

④ S_2 MERGE-SUM(A, Q+1, R)

⑤ IF($S_1 = S_2$) RETURN TRUE

⑥ RETURN FALSE;

1 2 3 4 5 6 7 8 9

1+1   1+2 . . .   P

<   , >

VALE PER (2N)

$O(n \log(n))$ → MERGE (DEC)

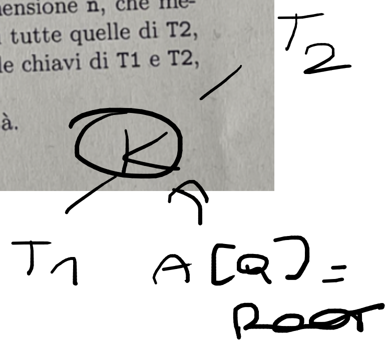
Esercizi

Esercizio 1 (9 punti) In questo esercizio si considerano alberi binari di ricerca (BST) completi, memorizzati in array con la stessa convenzione che si usa normalmente per la memorizzazione di uno heap in un array.

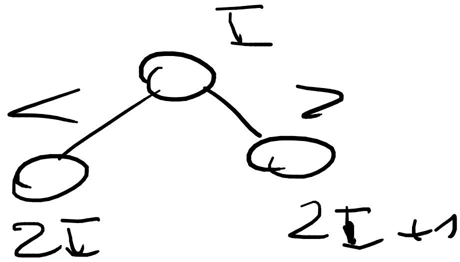
Realizzare una funzione `max(T,n)` che determina il massimo di un array `T` di dimensione `n` che memorizza un BST completo.

Scrivere quindi una funzione `merge(T1,T2,k)` che dati due array `T1` e `T2` di dimensione `n`, che memorizzano BST completi e una chiave `k` maggiore di tutte le chiavi di `T1` e minore di tutte quelle di `T2`, restituisce un array (di dimensione `2n+1`) che memorizza un BST che contiene tutte le chiavi di `T1` e `T2`, e la chiave `k`.

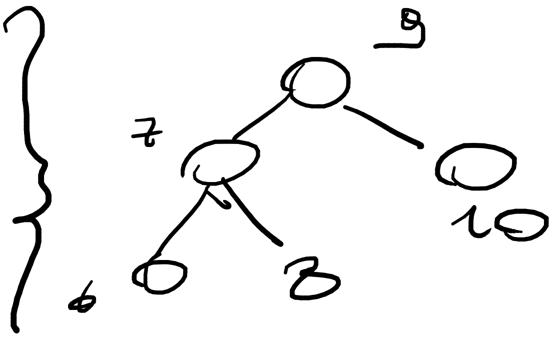
In entrambi i casi motivare la correttezza delle funzioni e valutarne la complessità.



- `A[i]` è il nodo genitore;
- `A[2i]` è il figlio sx del nodo `A[i]`;
- `A[2i+1]` è il figlio dx.



TREE



① MAX (T, n)

② RETURN MAX_REC(T, 1, n)

MAX_REC (T, P, Q)

① Q = P+R/2

② IF (MAX == A[Q] RETURN A[Q]

③ IF (P < R)

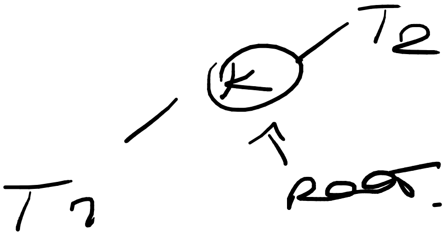
④ MAX = MAX_REC (T, P+1, Q)

$T\left(\frac{n}{2}\right) \approx O(\log(n))$

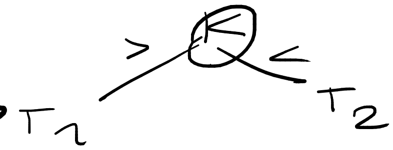
$A[i]$ = PARENT

$A[2i]$ = LEFT

$A[2i+1]$ = RIGHT

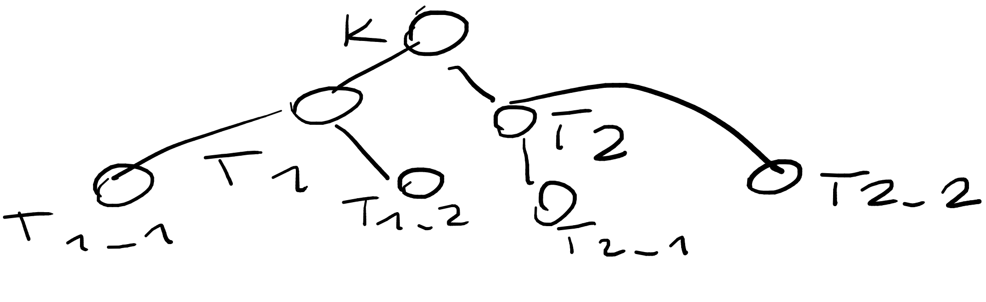


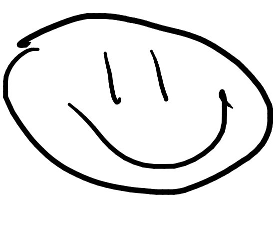
MERGE(T_1, T_2, K)



1. MAX_T1 = MAX(T_1, K)
2. IF (MAX_T1 $\geq$ K) ERROR("NOPS!")
3. ELSE
4. T.R = K
5. FOR (i = 1 TO N)
6. T[2i] = (T1[i]) // LEFT
7. T[2i+1] = (T2[i]) // RIGHT

[K - T1T2 - T1_1 - T1_2 - T2_1 - T2_2]





- SIAMO SODDISFATTI!