[ALGORITMI] [09/11/2024]

DEF. TEORICA

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 $f(n)/g(n) = O(n)$.

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

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

↓ FUNZ. SOPRA

↑ FUNZ. SOTTO

$f(n) = O(n^2)$ <

$g(n) = \Omega(n)$

$\rightarrow O(n^2)$, ... $0 \leq g(n) \leq cf(n^2)$

$\Omega(n)$, ... $0 \leq df(n) \leq g(n)$

$\rightarrow 0 \leq df(f(n)) \leq g(n) \leq cf(n^2)$

$O(n)$

$\frac{f(n)}{g(n)} = O(n) \rightarrow \left[0 \leq g(n) \leq \frac{cf(n)}{f(n)} \right]$

$0 \geq \frac{f(n)}{g(n)} \geq c$

$k \geq 0, \ \forall n \in \mathbb{N}$ ↑ ITA SOTTO

Domanda 17 Dare la definizione di $\Omega(f(n))$. Mostrare che se $f(n) = \Omega(g(n))$ e $g(n) = \Omega(h(n))$ allora $f(n) = \Omega(h(n))$.

Nei seguenti due casi, si devono dimostrare le condizioni date usando le definizioni formali (con i limiti).
f(n) = omega(g(n)) --> 0 <= c1g(n) <= f(n)

$\Omega \rightarrow 0 \leq c \ldots \leq f$

$\Omega(f(n)) \rightarrow \exists g \mid 0 \leq cg(n) \leq f(n)$

FUNZIONE CHE STA SOTTO

$\Omega(h(n)) \rightarrow \exists$ $0 \leq cf(n) \leq h(n)$

$f(n) = \Omega(h(n))$

STA SOTTO $h(n)$!

$0 \leq cg(n) \leq cf(n) \leq h(n)$

$\exists c_1, c_2 \rightarrow 0 \leq c_1 g(n) \leq c_2 f(n) \leq h(n)$

$\forall n \in \mathbb{N}, \exists n_0 = \max(c_1, c_2)$ — LA CONFRONTIAMO

$\rightarrow \quad 0 \leq n_0 f(n) \leq h(n)$

FUNZIONA PER... TUTTI I NUMERI

Se $f(n) = \Omega(g(n)$ e $g(n) = \Omega(h(n))$ allora esistono $c_1, c_2 > 0$, $n_1, n_2 \in \mathbb{N}$ tali che per ogni $n \geq n_1$

$$0 \leq c_1 g(n) \leq f(n) \tag{1}$$

e per ogni $n \geq n_2$

$$0 \leq c_2 h(n) \leq g(n) \tag{2}$$

Ne consegue che per ogni $n \geq \max\{n_1, n_2\}$, moltiplicando (2) per c_1 si ha

$$0 \leq c_1 c_2 h(n) \leq c_1 g(n) \leq f(n)$$

ovvero, indicato con $n_0 = \max\{n_1, n_2\}$ e $c = c_1 c_2$, per ogni $n \geq n_0$,

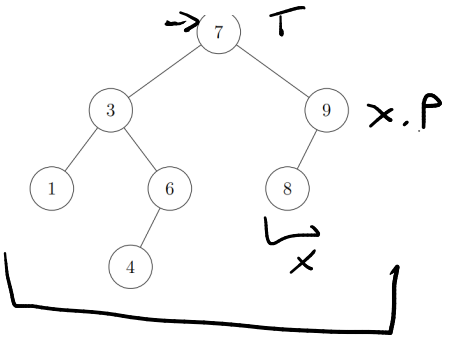
$$0 \leq ch(n) \leq f(n)$$

ovvero $f(n) = \Omega(h(n))$.

Qui metto insieme le costanti c1 e c2 tali che valga per tutte le funzioni considerate, partendo da h(n). "Togliendo di mezzo" c1 e c2 (metto una costante sola, tale che sia il massimo e quindi g(n) diventa "inclusa" in h(n) --> essendo che metto una costante sola e h(n) ne ha due, g(n) diventa trascurabile. A quel punto, notiamo che h(n) è un limite inferiore (omega).

Esercizio 1 (10 punti) Realizzare un arricchimento degli alberi binari di ricerca che permetta di ottenere per ogni nodo x, il grado di bilanciamento del sottoalbero radicato in x, definito come $h_x / \log_2(n_x + 1)$ dove h_x e n_x indicano rispettivamente l'altezza e il numero di nodi del sottoalbero radicato in x (si intende che un albero costituito da un solo nodo abbia altezza 1).

Indicare quali campi occorre aggiungere ai nodi. Fornire lo pseudo-codice per la funzione `bal(x)` che restituisce il grado di bilanciamento del sottoalbero radicato in x e la procedura di inserimento di un nodo `insert(T,z)`. Valutare la complessità delle funzioni definite.



T = TREE

T = x.root

x.L x.R x.p = PARENT

- INSERIMENTO (INSERTING)
- CANCELLAZIONE (TRANSPLANT)
- RICERCA

SI USANO SEMPRE (1)

```
INSERT(T, z)
1   x = T.root
2   y = NIL
3   while x ≠ NIL
4       y = x
5       if z.key < x.key
6           x = x.left
7       else
8           x = x.right
9   z.p = y
10  if y = NIL
11      T.root = z
12  else
13      if z.key < y.key
14          y.left = z
15      else
16          y.right = z
```

COPIA RADICE → RADICE

DOVE INSERIRE → x(x)

BST

INSERISCO IL NODO

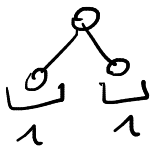
Complessità $O(h)$

TUTTI I CAMPI + BAL(T) ↓ ALBERO → [x.p / x.L / x.R / x.key]

Esercizio 1 (10 punti) Realizzare un arricchimento degli alberi binari di ricerca che permetta di ottenere per ogni nodo x, il grado di bilanciamento del sottoalbero radicato in x, definito come $h_x / \log_2(n_x + 1)$ dove h_x e n_x indicano rispettivamente l'altezza e il numero di nodi del sottoalbero radicato in x (si intende che un albero costituito da un solo nodo abbia altezza 1).

Indicare quali campi occorre aggiungere ai nodi. Fornire lo pseudo-codice per la funzione `bal(x)` che restituisce il grado di bilanciamento del sottoalbero radicato in x e la procedura di inserimento di un nodo `insert(T,z)`. Valutare la complessità delle funzioni definite.

BAL(x)

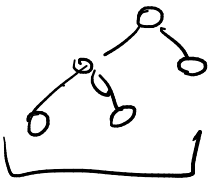


1/1 = 1

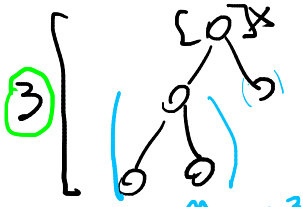
$h_x \rightarrow$ ALTEZZA > LOGALG

$\log_2(N_x + 1)$

H. CLASSICA



$O(h) = \lfloor \log_2(n) + 1 \rfloor$



$\lfloor h_x \rfloor / \log_2(3) + 1 = 3 / 1.5 + 1 = (\ldots)$

NUMERO = GRADO DI BILANCIAMENTO

= N. NODI RADICATI NEL SOTTO ALBERO

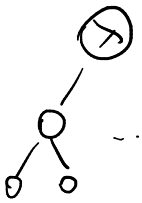
Esercizio 1 (10 punti) Realizzare un arricchimento degli alberi binari di ricerca che permetta di ottenere per ogni nodo x, il grado di bilanciamento del sottoalbero radicato in x, definito come $h_x / \log_2(n_x + 1)$ dove h_x e n_x indicano rispettivamente l'altezza e il numero di nodi del sottoalbero radicato in x (si intende che un albero costituito da un solo nodo abbia altezza 1).

Indicare quali campi occorre aggiungere ai nodi. Fornire lo pseudo-codice per la funzione `bal(x)` che restituisce il grado di bilanciamento del sottoalbero radicato in x e la procedura di inserimento di un nodo `insert(T,z)`. Valutare la complessità delle funzioni definite.

BAL(x)
RETURN BAL-REC(x.ROOT)

- BAL-REC(x, H) ↙ ERROR(...)
- → 1 IF x = NIL RETURN (NIL, NIL)
- → 2 LEFT_NODES = BAL_REC(x.LEFT, H+1)

H
N_NODES



→ 3 RIGHT_NODES = BAL_REC(x.RIGHT, H+1)

→ 4 RETURN (H / (LEFT_NODES + RIGHT_NODES))

PAOLONI - LUKS:

Soluzione: L'idea è quella di arricchire la struttura facendo in modo che ogni nodo x, oltre ai campi usuali, abbia due campi aggiuntivi: x.`h` che contiene l'altezza del sottoalbero radicato in x e x.`size` che contiene il numero dei nodi in tale sottoalbero.

A questo punto è immediato realizzare la funzione `bal(x)` di tempo costante

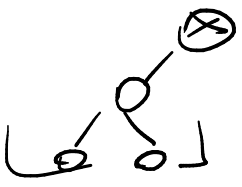
```
bal(x)
  if x <> nil
     return x.h/log_2(x.size)
  else
     error
```

Invece per l'inserimento, si modifica la procedura standard, che discende l'albero dalla radice fino alla posizione in cui inserire il nuovo nodo, facendo in modo che tutti i nodi attraversati e che quindi conterranno nel sottoalbero il nuovo nodo, abbiano il campo `size` incrementato di 1. Inoltre, una eventuale modifica dell'altezza va propagata verso l'alto. Questo accade quando il parent del nodo inserito era una foglia e quindi la sua altezza, prima dell'inserimento, era 1.

- CORRETTEZZA

→ 0 (x) → 1/1 (BASE)

(x) → x! = NIL



→ BAL

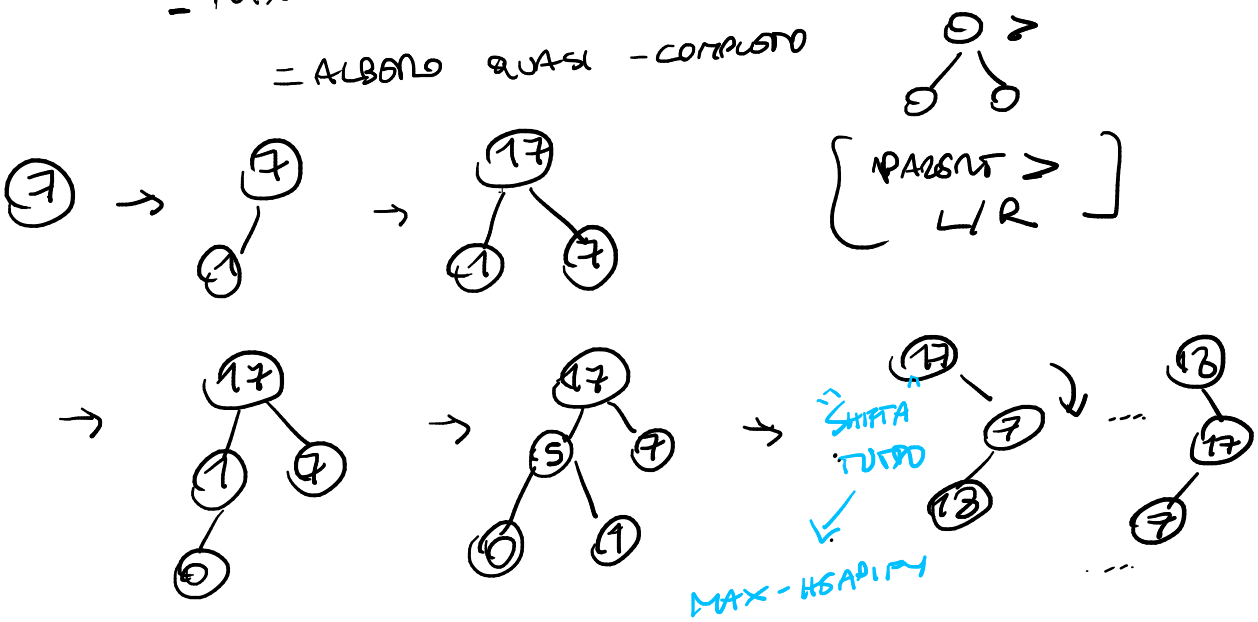
x.L / x.R L/R

Σ = SOMMA

Domanda B (7 punti) Dare la definizione di max-heap. Data la sequenza di elementi `7, 1, 17, 0, 5, 18,` si specifichi il max-heap ottenuto inserendo, a partire da uno heap vuoto, uno alla volta questi elementi nell'ordine indicato e infine rimuovendo 0. Si descriva sinteticamente come si procede per arrivare al risultato.

INSERIMENTO

- MAX-HEAP
= ALBERO QUASI-COMPLETO

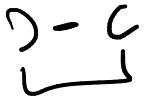


SITO: VISUALGO.NET/EN/HEAP

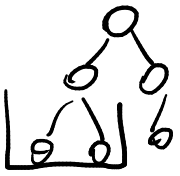
(MAX-HEAP)

[PRIORITY QUEUE]

[UP NEXT] - NO SPOILER



DIVIDE AND CONQUER



↑ SOTTOPROBLEMI = DEVONO ESSERE RICALCOLATI

PROGR. DINAMICA

↖ CREA UNA STRUTTURA DATI PER LA MIGLIORE SOLUZIONE

ITERATIVO

RICORSIVO

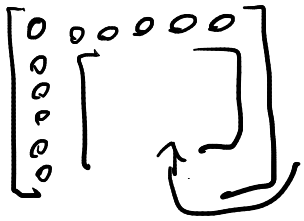
MATRICE → M[i,j] → [→ ↓]

LCS → SOTTO STRINGA PIÙ LUNGA COMUNE

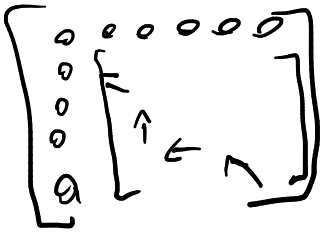
LONGEST INCREASING SUB SEQUENCE

ABCD

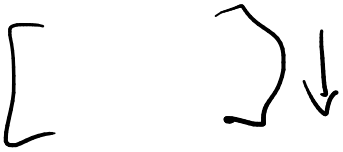
BBC

M=  SOLUZIONE

RICORRENZA SUL COSTO

M=  → NO RI CALCOLO = HO GIÀ LA SOLUZIONE

BOTTOM - UP 

TOP - DOWN 

LCS(X, Y)

```
1  m = X.[length]
2  n = Y.length
3  for i = 0 to m
4      L[i, 0] = 0
5  for j = 0 to n
6      L[0, j] = 0
7  for i = 1 to m
8      for j = 1 to n
9          if x_i = y_j
10             L[i, j] = L[i − 1, j − 1] + 1
11             B[i, j] = '↖'
12         else if L[i − 1, j] ≥ L[i, j − 1]
13             L[i, j] = L[i − 1, j]
14             B[i, j] = '↑'
15         else
16             L[i, j] = L[i, j − 1]
17             B[i, j] = '←'
18 return (L[m, n], B)
```

INIZIALIZZAZIONE

$X = \langle b, d, c, d \rangle$

$Y = \langle a, b, c, b, d \rangle$

Restituisci $LCS(X, Y)$ e $|LCS(X, Y)|$

$L =$

		a	b	c	b	d
	0	0	0	0	0	0
b	0	0	1	1	1	1
d	0	0	1	1	1	2
c	0	0	1	2	2	2
d	0	0	1	2	2	3

$B =$

	a	b	c	b	d
b	↑	↖	←	↖	←
d	↑	↑	↑	↑	↖
c	↑	↑	↖	↑	↑
d	↑	↑	↑	↑	↖

$LCS(X, Y) = \langle b, c, d \rangle$ $\quad |LCS(X, Y)| = 3$

Pseudocodice Memoizzato

INIT-LCS(X, Y)

```
1  m = X.length
2  n = Y.length
3  if (m = 0) or (n = 0)
4      return 0
5  for i = 0 to m
6      L[i, 0] = 0
7  for j = 0 to n
8      L[0, j] = 0
9  for i = 1 to m
10     for i = 1 to n
11         L[i, j] = −1
12 return R-LCS(X, Y, m, n)
```

→ MEMOIZZAZIONE

R-LCS(X, Y, i, j)

```
1  if L[i, j] = −1
2      if x_i = y_j
3          L[i, j] = R-LCS(X, Y, i − 1, j − 1)
4      else if R-LCS(X, Y, i − 1, j) ≥ R-LCS(X, Y, i, j − 1)
5          L[i, j] = L[i − 1, j]
6      else
7          L[i, j] = L[i, j − 1]
8  return L[i, j]
```

Definisco

$l(i, j) = |LCS(X_i, Y_j)|$

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

→ RIEMPIO LA MATRICE

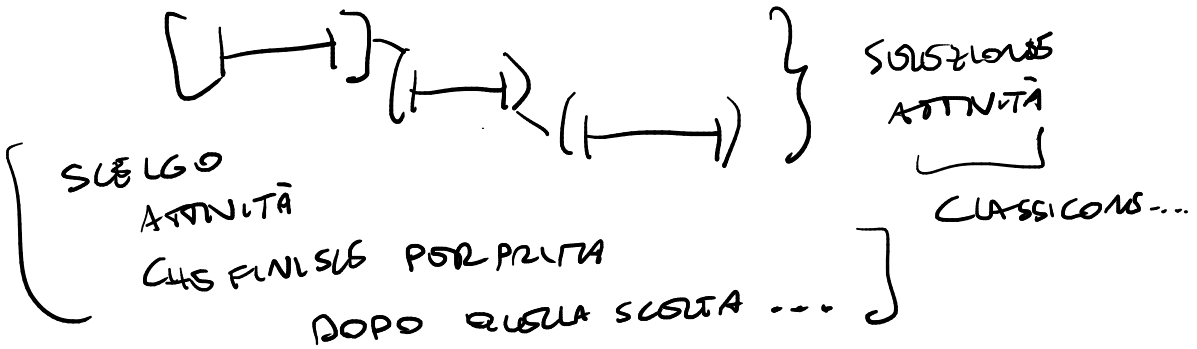
Alla fine ci interessa calcolare $l(m, n)$.

Per calcolare $l(i, j)$ mi possono servire tre valori:

$$L = \begin{bmatrix} (i-1, j-1) & (i-1, j) \\ \nwarrow & \uparrow \\ (i, j-1) & \leftarrow \quad (i, j) \end{bmatrix}$$

Scansione "row-major": riempio la tabella per righe, da sinistra a destra.

GREEDY → FAI LA SCELTA MIGLIORE



↑ VEDREMO! (:))

[POO] → TEMPLATE <T>

TYPENAME

```
CLASS QUEUE <T> {
// ....
}
```

↑ T = SEGNAPOSTO → INT / DOUBLE

CLASSE ANNIDATA → QUEUE ITEM

↑

CLASSE ITERATORE

FRIEND - VEDE I SUOI CAMPI PRIVATI

```
template<class T>
class C {
private:
  T t;
public:
  C(const T&);
  friend void f_friend<T>(const C<T>&);
  // amicizia associata
};

template<class T>
C<T>::C(const T& x) : t(x) {}

template<class T>
void f_friend(const C<T>& c){
  cout << c.t << endl; // per amicizia
}
```

→ VEDE TUTTO PRIVATO

LISTA

-[NODO] → ITERATORE

OVERLOADING (RIDEFINIZIONE)

++, →, ==, !=

<<

LISTA L<T>;

↓ ALL'ESTERNO DELLA CLASSE

→ TEMPLATE CLASS<T> OSTREAM& OPERATOR<< (CONST OSTREAM& OS, CONST LISTA<T> L);

FAI PASSO DI TEMPLATE

Dichiarazione nel template di classe **C** di un template di classe o di funzione friend **non associato**

→ NON ESISTE GIÀ NELLA CLASSE

```
template<class T>
class C {

  template<class Tp>        // amicizia con template
  friend int A<Tp>::fun();  // di metodo

  template<class Tp>        // amicizia con template
  friend class B;           // di classe

  template<class Tp>        // amicizia con template
  friend bool test(C<Tp>); // di funzione
};
```

SIA CLASSI CHE METODI

```
class A {
public:
  A(int x=0) {cout << x << "A() ";}
};

template<class T>
class C {
public:
  static A s;
};
template<class T>
A C<T>::s=A(); // stampa 0A()

int main() {
  C<double> c;
  C<int> d;
  C<int>::s = A(2);
}
// stampa: 0A() 2A()
// NOTA BENE: non stampa 0A() 0A() 2A() !!
```

→ COSTRUITO UNA VOLTA SOLA

STAMPA

→ TEMPLATE ASSICURA UNA SOLA COSTRUZIONE

```
class Z {
public:
  ...
};

template <class T1, class T2=Z>
class C {
public:
  T1 x;
  T2* p;
};

template<class T1,class T2>
void fun(C<T1,T2>* q) {
  ++(q->p); // NO PUNTO OPERATOR ++
  if(true == false) cout << ++(q->x);
  else cout << q->p;
  (q->x)++;
  if(*(q->p) == q->x) *(q->p) = q->x;
  T1* ptr = &(q->x);
  T2 t2 = q->x;
}

main(){
  C<Z> c1; fun(&c1); C<int> c2; fun(&c2);
}
```

INIZIALIZZAZIONE TIPO = Z

T1 / T2 → GENERICI / TYPENAME (T)

(!) OPERATOR ++

A(++) → POSTFIX

++A A

PREFIX

NO NEED FOR "="

NO OPERATORI (=) → SONO OGGETTI

NO COPIA

Si considerino le precedenti definizioni. Fornire una dichiarazione (non è richiesta la definizione) dei membri pubblici della classe `Z` nel **minor numero possibile** in modo tale che la compilazione del precedente `main()` non produca errori. **Attenzione:** ogni dichiarazione in `Z` non necessaria per la corretta compilazione del `main()` sarà penalizzata.

```
class Z {
public:
  int operator++() {} // 1
  Z& operator++(int) {} // 2
  bool operator==(const Z& x) const {} // 3
  Z(int x=0) {} // 4
```

BISOGNO 1 FARE ++

→ == Z / != Z

BOOLEANI (T F)

↑ COSTRUTTORE Z

Esercizio 2

```
class A {
  bool x;
public:
  virtual ~A() = default;
};

class B {
  bool y;
public:
  virtual void f() const { cout << "B::f "; }
};

class C: public A {};

class D: public B {
public:
  void f() const { cout << "D::f "; }
};

class E: public D {
public:
  void f() const { cout << "E::f "; }
};

template<class T>
void Fun(const T& ref) {
  try{ throw ref; }
  catch(const C& c) {cout << "C ";}
  catch(const E& e) {cout << "E "; e.f();}
  catch(const B& b) {cout << "B "; b.f();}
  catch(const A& a) {cout << "A ";}
  catch(const D& d) {cout << "D ";}
  catch(...)        {cout << "GEN ";}
}

C c; D d; E e; A& a1 = c; B& b1 = d; B& b2 = e; D& d1 = e; D* pd = dynamic_cast<E*>(&b2);
```

→ ES. TIPO
ESAME
(VECCHIO ...)

PAO → RACCOLTA_ESERCIZI → TUTTO
(CLONE CACHE)

ALGORITMI → RACCOLTA
TUTTI_GLI_ESERCIZI
ALBERI / HEAP / RICORSIONE