

LEZIONE 1

Introduzione

Modalità d'esame: prova scritta - def
 - teoreme
 - risp aperto
 - esercizi
 prova in lab - programmare l' stesso appello

Programma: Calcolabilità - nomenclatura
 - modelli di calcolo
 Complessità - nomenclatura
 - notazione asintotica
 - importante studio algoritmi

• Problema del sudoka

81 celle 24 adeguate / appannaggio • un assegnamento delle 81 celle
 • la verifica di correttezza

Gli ass. possibili sono 9^{81}

Supponendo che il funz corretto sia a metà; $9^{81}/2$, per quello corretto.

Dopo quanto tempo mi aspetto una soluzione? Dato un determinato algoritmo:

$$9^{81}/2 \cdot 10^{-9} \text{ s} = 1,232 \cdot 10^{54} \cdot 10^{-9} \text{ s} = 1,232 \cdot 10^{45} \text{ s} \rightarrow 3,422 \cdot 10^{41} \text{ ore} \rightarrow 1,425 \cdot 10^{40} \text{ giorni} \\ \rightarrow 3,906 \cdot 10^{37} \text{ anni} = \text{non c'è un gran che}$$

Quindi, questo è il motivo per cui studiare le complessità degli algoritmi

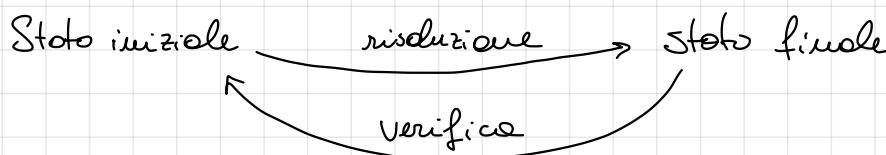
LEZIONE 2

Definizioni, problemi e soluzioni, processi e programmi

Definizione di problema: ha diverse definizioni e queste sono generali.
 L'idea è comunque di voler trovare un processo risolutivo per ottenere un risultato.

Definizione di soluzione: partire da stati iniziali per arrivare a dei risultati.
 Ci sono stati intermedi da cui possiamo.

↪ è l'insieme di azioni che ci porta al risultato, partendo dallo stato iniziale.

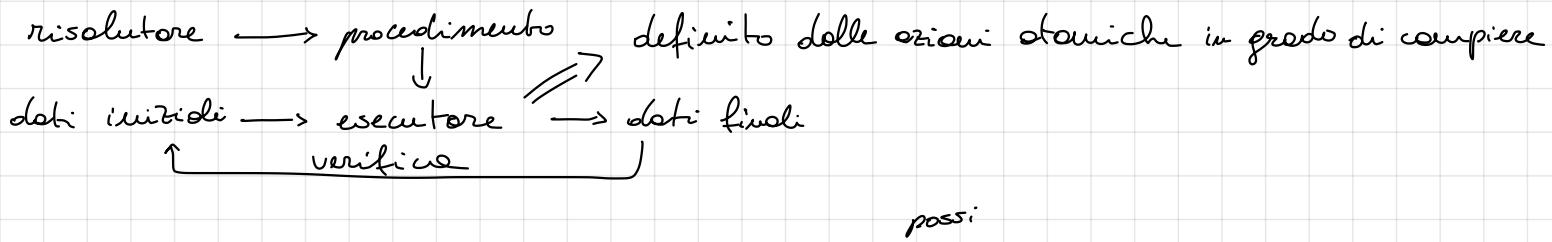


Come possiamo risolvere un problema? Utilizziamo un elaboratore

- Elaboratore: strumento che consente di esprimere la risoluzione di un problema
- Come risolvere un problema con un elaboratore? Utilizziamo un esecutore
- L'esecutore: compie le azioni che portano al risultato finale di un problema
 ↪ è slegato dal problema
 es: esecutore → computer

- Il risolutore: comincia le istruzioni da compiere all'esecutore, utilizzando un opportuno linguaggio, consueto da entrambi

Azioni = istruzioni + esecutore



Processo: azione composta da sequenze di azioni stamche svolte da esecutore

Programma: è la descrizione di un processo, usando un linguaggio comprensibile dell'esecutore

processo = programma + esecutore // descrizione programma

{ - elenco oggetti
 - insieme azioni
 - ordine azioni
 - condizioni
 moroni quanto ci possesse sto zoccole

Algoritmo: è un elenco finito di istruzioni univocamente interpretabili.

- d'esecuzione deve terminare per i possibili input.
- diversi algoritmi possono risolvere lo stesso problema.
- lo stesso algoritmo può risolvere diversi problemi.

Def più formale: è un programma che, in tutti i processi che si possono creare, ha le seguenti caratteristiche:

- | | | | |
|---|----------------------------------|---|--|
| 1 | finitezza descrizione | 6 | limite finito complessità istruzioni |
| 2 | non limitatezza dati ingresso | 7 | disponibilità memoria illimitata |
| 3 | non limitatezza dati uscita | 8 | esecutore opera in modo discreto |
| 4 | non limitatezza possi eseguibili | 9 | terminazione per ogni dato in ingresso valido. |
| 5 | definitezza | | |

NB: non per forze tutti i programmi devono terminare → doverai / agent

- Macchina astratta: Quando un esecutore ha a disposizione:

- input
- output
- memoria virtualmente illimitata
- procedure risolutiva di un problema

allora ha una macchina astratta

- Alfabeto: insieme di simboli uso reale e finito

A

↓

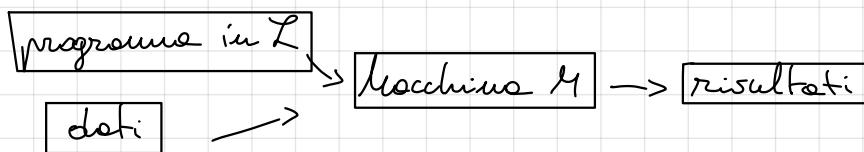
A*

→ Stringhe: sequenze finite generabili con i simboli dell'alfabeto

↓

→ Linguaggio: dato un alfabeto, un linguaggio L è un sottoinsieme di A*

• Gerarchie di macchine:



Un linguaggio ha
 ↓
 Sintassi:
 (correttezza)
 Semantica
 (uso)

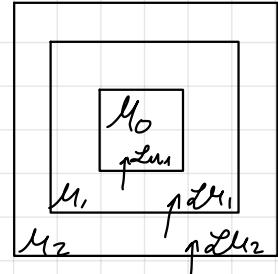
Ogni macchina è descritta completamente: → del proprio linguaggio
 → delle proprie azioni

Interprete: Esegue direttamente gli ordinativi imparititi (istruzioni M_{i-1})

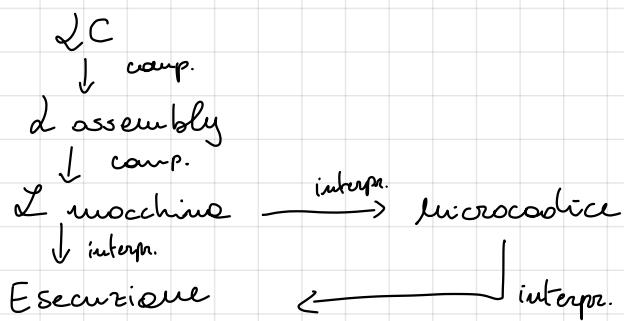
Compilatore: realizza la traduzione da L_i e L_{i-1} per la macchina

↳ le 2 fasi $L_i \rightarrow L_{i-1} \Rightarrow M_{i-1}$
 - Compile-time → run-time

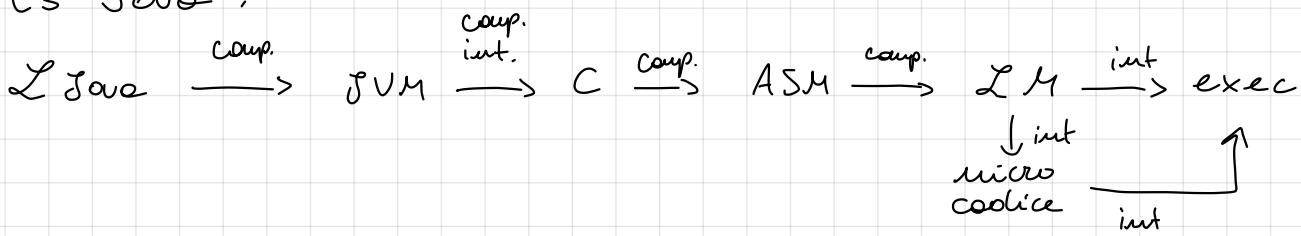
per passare a macchine più interne



Es linguaggio C:



Es Java:



LEZIONE 3 :

• Automa e stati finiti:

Automa: dispositivo in grado di interagire con l'ambiente esterno esibendo un dato comportamento in uscita come risposta a dati di ingresso.

A stati finiti: gli stati interni alla macchina sono finiti.

- Ad ogni passo, leggono un simbolo in input → ad ogni passo producono un simbolo in uscita
- Gli alfabeti dei dati in uscita possono essere diversi da quelli in entrata
- Un simbolo può far cambiare stato all'automa
- Ad ogni istante l'automa si trova in uno stato preciso.

Mu' autome e stati finiti e' un esempio di 7 elementi $M = \langle A, B, Q, F, G, q_1, q_F \rangle$ dove:

$A = \{ \alpha_0, \dots, \alpha_{m-1} \}$ alfabeto input

$B = \{ b_0, \dots, b_{k-1} \}$ alfabeto output

$Q = \{ q_0, \dots, q_{n-1} \}$ insieme finito non vuoto di stati

$F: A \times Q \rightarrow B$ funz. che associa a ogni coppia ammessa (α_i, q_j) il simbolo da produrre

$G: A \times Q \rightarrow Q$ funz. che associa " " " " (α_i, q_j) stato in cui deve portarsi

$q_1 \in Q$ stato iniziale

$q_F \in Q$ stato finale

INGRESSI

Esempio: $A = \{ MG, MP \}$ autome distributore biglietti \rightarrow moneta grande o piccola

$MG \rightarrow MG$ X

$MP \rightarrow MP$ X

$MG \rightarrow MP$ ✓

$MP \rightarrow MG$ ✓

USCITE $B = \{ \text{ancora}, \text{restituisci}, \text{emetti} \}$

STATI $Q = \{ q_0, q_1, q_2 \}$ $q_0 = \text{no moneta inserita}$ $q_1 = \text{MG inserita}$

$q_2 = \text{MP inserita}$

tabelle stati

| | MG | MP |
|-------|-------|-------|
| q_0 | q_1 | q_2 |
| q_1 | q_1 | q_0 |
| q_2 | q_0 | q_2 |

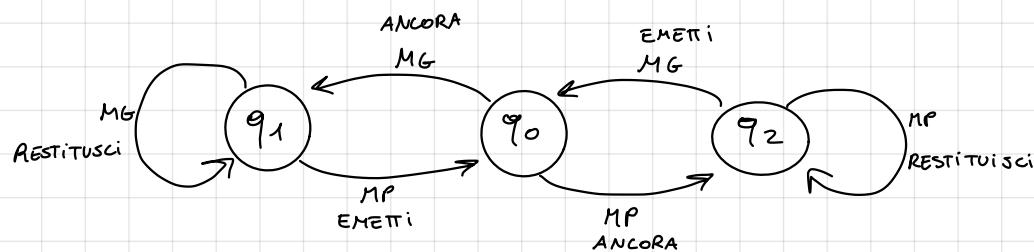
Tabella uscite:

| | MG | MP |
|-------|-------------|-------------|
| q_0 | ancora | ancora |
| q_1 | restituisci | emetti |
| q_2 | emetti | restituisci |

Per stati iniziali e finali

$q_i = q_0$ } nonostante siano uguali non vuol dire
 $q_F = q_0$ } che non succeda nulla

• Graph di transizione \rightarrow per definire gli automi



• Tabella di transizione: riassume le 2 tabelle uscite prima

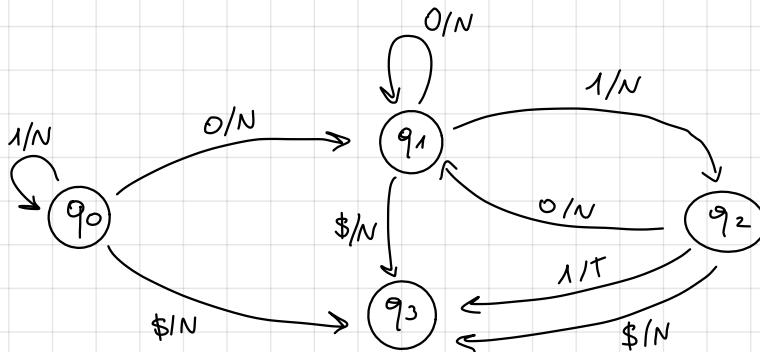
| | MG | MP |
|-------|--------------------------|--------------------------|
| q_0 | q_1/ancora | q_2/ancora |
| q_1 | $q_1/\text{restituisci}$ | q_0/emetti |
| q_2 | q_0/emetti | $q_2/\text{restituisci}$ |

Esempio: trovare stringhe 011 con \$ interrupt

$A = \{0, 1, \$\}$ ingressi $B = \{N, T\}$ uscite

$q_i = q_0$ stato iniziale
 $q_f = q_3$ stato finale

insieme stati $Q = \{q_0, q_1, q_2, q_3\}$
 q_0 = nessun 0 letto
 q_1 = 0 letto
 q_2 = 0 1 letto
 q_3 = 0 11 - stato finale



Grado transizione

Tabelle uscite

| F | 0 | 1 | \$ |
|-------|---|---|----|
| q_0 | N | N | N |
| q_1 | N | N | N |
| q_2 | N | T | N |
| q_3 | | | |

Tabelle stati

| G | 0 | 1 | \$ |
|-------|-------|-------|-------|
| q_0 | q_1 | q_0 | q_3 |
| q_1 | q_1 | q_2 | q_3 |
| q_2 | q_1 | q_3 | q_3 |
| q_3 | | | |

Tabelle transizione

| | 0 | 1 | \$ |
|-------|---------|---------|---------|
| q_0 | q_1/N | q_0/N | q_3/N |
| q_1 | q_1/N | q_2/N | q_3/N |
| q_2 | q_1/N | q_3/T | q_3/N |
| q_3 | | | |

LEZIONE 4

Machines di Turing: macchina di esecutore formalizzata nel 1936 → idea astratta di esecutore
Studio dei metodi di risoluzione di problemi
de macchine di Turing → **Molt** → matematico al lavoro.

Supposizioni: ispirato ad un matematico al lavoro

(no limite)

- foglio e quadretti ma su una unica riga, sequenziali, arbitrariamente lunga
- comportamento → simboli che osserva
 - stato della sua mente
- stato complessivo del sistema
 - seq. simboli scritti sul foglio
 - simboli sotto osservazione del calcolatore
 - stato della mente del calcolatore

Descrizione informale:

- Nastro virtualmente illimitato diviso in caselle, ciascuna contenente un simbolo
 - Testina di lettura/scrittura (TCS) → per leggere/scrivere simboli sul nastro
 - Organo di controllo (OC) con → meccanismo capace di eseguire istruzioni del programma
- ↳ automa a stati finiti

NB: la testina può spostarsi a destra e sinistra

NB: da scrivere su una casella cancella il simbolo precedente

Occorre definire un alfabeto di simboli $A = \{e_0, \dots, e_{n-1}\}$

OC accede al nostro tramite TCS / OC è in grado di assumere uno stato appartenente a un insieme finito non vuoto $Q = \{q_0, \dots, q_{n-1}\}$
 $X = \{D, S\}$ insieme simboli relativi: spostamento TCS

• La Molt opera in modo discreto: At > 0 tra due variazioni succ. dello stato globale.

ad ogni passo OC → $F: A \times Q \rightarrow A$ è una funz. che associa a ogni coppia (e_i, q_j) il simbolo da scrivere nel riposiz.

→ $G: A \times Q \rightarrow Q$, " " " " " " " " " " il nuovo stato dell'OC

→ $Dir: A \times Q \rightarrow X$, " " " " " " " " le dir. in cui la TCS deve muoversi

• Comportamento Molt univocamente determinato da un insieme di quintupleti

$$\langle e_i, q_j, F(e_i, q_j), G(e_i, q_j), Dir(e_i, q_j) \rangle$$

Definizione:

Molt quintuplo $\langle A, Q, P, q_0, q_f \rangle$

- $A = \{q_0, \dots, q_{m-1}\}$ alfabeto dei simboli
- $Q = \{q_0, \dots, q_{n-1}\}$ insieme finito non vuoto degli stati
- $P \subseteq Q \times A \times Q$ l'insieme finito delle quintuple che descrivono il comportamento dello meccanismo
- $q_0 \in Q$ stato iniziale
- $q_f \in Q$ stato finale

Altrimenti



si usa la Matrice Funzionale

Molt in Matrice Funzionale

Molt \rightarrow è buon esecutore
 \rightarrow deterministica

- tante righe quanti sono gli stati
- tante colonne quanti sono i simboli
- elementi $\langle q', q'', x \rangle$ dove
 - $q' \in A$ nuovo simbolo scritto
 - $q'' \in Q$ nuovo stato raggiunto
 - $x \in \{D, S\}$ direzione TLS

Fissate una riga e fissate una colonna avremo il nuovo simbolo, il nuovo stato e dir.

Caratteristiche:

- operazioni di complessità finita
- no limite ingresso/uscita in seq. simboli

Molt \rightarrow opera in modo discreto

\rightarrow supporto di memorizzazione virtualmente illimitato

\rightarrow numero illimitato di passi

\rightarrow permette computazioni che non terminano mai

Esempio:

trave successivo (modt) di un numero in base 10, lasciandolo scritto sul nostro

$$A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \quad (11 \text{ finito} > 0 = \text{alf. alfabeto})$$

Conf. iniziale = 0C in q_0

TLS: legge le cifre meno significative

$x < q$ oppure $1 < s$ si ferme

se $e > q$ salta 0 e va a q

se b scrive 1

Visimo 2 stati: $Q = \{q_0, q_1\}$

$q_0 = 0.1$ deve essere sommato

$q_1 = \text{stato finale}, 1$ sommato

| | | | | |
|---|----|---|----|---|
| O | 90 | 1 | 91 | D |
| I | . | 2 | . | D |
| Z | . | 3 | . | D |
| S | . | 4 | . | D |
| L | . | 5 | . | D |
| S | . | 6 | . | D |
| 6 | . | 7 | . | D |
| Z | . | 8 | . | D |
| 8 | . | 9 | 91 | D |
| 9 | . | 0 | 90 | S |
| Y | 90 | 1 | 91 | D |

1) poss simb x 2 stati = 22 possibili quintuple
C) noi prendiamo solo quelle con q_0

$$\text{Esempio: } \begin{array}{c} b \\ | \\ 3 \\ | \\ 6 \\ | \\ b \end{array} \Rightarrow \begin{array}{c} b \\ | \\ 3 \\ | \\ 7 \\ | \\ b \\ | \\ 9 \end{array}$$

$$\text{exemplo: } \begin{array}{|c|c|c|c|c|} \hline & 6 & | & 1 & | & 4 & | & 9 & | & 6 \\ \hline & & | & & & & & & & \\ \hline \end{array} \Rightarrow \begin{array}{|c|c|c|c|c|} \hline & 6 & | & 4 & | & 0 & | & 6 \\ \hline & & \uparrow & & & & & \\ \hline \end{array} \Rightarrow \begin{array}{|c|c|c|c|c|} \hline & 6 & | & 5 & | & 0 & | & 6 \\ \hline & & & & & & & \\ \hline \end{array}$$

$$\text{esempio: } \begin{array}{c} | \\ 1 \\ \downarrow \\ 1 \\ 1 \\ 1 \end{array} \mid \begin{array}{c} 1 \\ 1 \\ 1 \\ 1 \end{array} \mid \begin{array}{c} 1 \\ 1 \\ 1 \end{array} \xrightarrow{q_0} \begin{array}{c} | \\ 1 \\ \uparrow \\ 1 \\ 1 \\ 1 \end{array} \mid \begin{array}{c} 1 \\ 1 \\ 1 \end{array} \xrightarrow{q_0} \begin{array}{c} | \\ 1 \\ \uparrow \\ 1 \\ 0 \\ 1 \\ 1 \end{array} \mid \begin{array}{c} 1 \\ 1 \end{array} \xrightarrow{q_0} \begin{array}{c} | \\ 1 \\ \uparrow \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{array} \mid \begin{array}{c} 1 \\ 1 \end{array} \xrightarrow{q_1} \begin{array}{c} | \\ 1 \\ \uparrow \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{array} \mid \begin{array}{c} 1 \\ 1 \end{array} \end{array}$$

Esempio 2: Prop. molt legge str da o,1 fermino \$ e scive 1 se gli 1 soes disponi

$$A = \{0, 1, \$, \%, \}\}$$

Visione 3 stati: $Q = \{q_0, q_1, q_2\}$ q_0 = gli 1 primi q_1 = gli 1 scelti disponibili
 q_2 = fine

Conf. iniziale: OC in φ_0

TLS: cambia stato a 1

Se $\$ \in \partial C$ $g_0 \rightarrow 0$ non

Se $\$ \in OC_{q_1} \rightarrow 1$ dispori

M. funzionale =

| | 0 | 1 | \$ |
|-----|-------|-------|-------|
| q_0 | 0q_0D | 1q_1D | 0q_2D |
| q_1 | 0q_1D | 1q_0D | 1q_2D |
| q_2 | | | |

| | | | | | |
|-----------|----------------|----------------|----------------|----------------|---|
| quintuple | 0 | q ₀ | 0 | q ₀ | D |
| | 1 | q ₀ | 1 | q ₁ | D |
| \$ | q ₀ | 0 | q ₂ | D | |
| 0 | q ₁ | 0 | q ₁ | D | |
| 1 | q ₁ | 1 | q ₀ | D | |
| \$ | q ₁ | 1 | q ₂ | D | |

Esempio

$10011\$ \rightarrow 10011\$ \rightarrow 10011\$ \rightarrow 10011\$ \rightarrow$

$10011\$ \xrightarrow{q_0} 10011\$ \xrightarrow{q_1} 100111q_2$

Funzioni calcolabili:

- Risolvere un problema significa saper calcolare (in tempo finito) la rel. funzionale che lega i dati ai risultati
- Problemi risolvibili / funzioni calcolabili
- ogni funzione calcolabile è Turing-calcolabile \rightarrow se esiste una MMT che la può calcolare.

LEZIONE 5 - Complessità

Complessità degli algoritmi:

Vogliamo progettare algoritmi che siano $\begin{cases} \text{corretti} \\ \text{e} \\ \text{efficienti} \end{cases}$

Costo di un algoritmo: quantifica le risorse necessarie per l'esecuzione, in termini di:

- numero di operazioni \rightarrow costo temporale
- spazio di memoria \rightarrow costo spaziale

Vedremo il modo per calcolare la complessità \rightarrow per ora approssimiamo
 (\hookrightarrow contiamo le righe di codice)

es: somme interi $1-n$ / potenza x^n / n-esimo Fibonaccii

Esempio: Somma

$$1^{\circ} \text{ Modo} \quad S = \sum_{i=1}^n i \qquad 2^{\circ} \text{ Modo} \quad S = \frac{n \cdot (n+1)}{2} \qquad (n = 1-\infty)$$

Codice in C (progr personale / C / somma.c)

$\%d$ indica il tipo della variabile stampata
 $\%d$ float, double

Il secondo algoritmo cresce di crescere più in
 \hookrightarrow è migliore in costo

Esempio: potenza

$$1^{\circ} \text{ Modo} \quad x^n = \begin{cases} 1 & \text{se } n=0 \\ \prod_{i=1}^n x & \text{se } n>0 \end{cases}$$

Caso peggiore 2^{n-1}

- 1^o \rightarrow dipende da n lineare $O(n)$
- 2^o \rightarrow dipende da n logaritmico $O(\log n)$

$$2^{\circ} \text{ Modo} \quad \downarrow \quad x^n = \begin{cases} 1 & \text{se } n=0 \\ (x^2)^{\lfloor \frac{n}{2} \rfloor} & \text{se } n \text{ pari} \\ (x^2)^{\lfloor \frac{n}{2} \rfloor} \cdot x & \text{se } n \text{ dispari} \end{cases}$$

peggiore nel caso
di numeri alti

LEZIONE 6 - risolvere problemi - complessità in base alle righe di codice

Abbiamo calcolato il costo in eigha dice dice → approssimative, intuitive, ma precise.

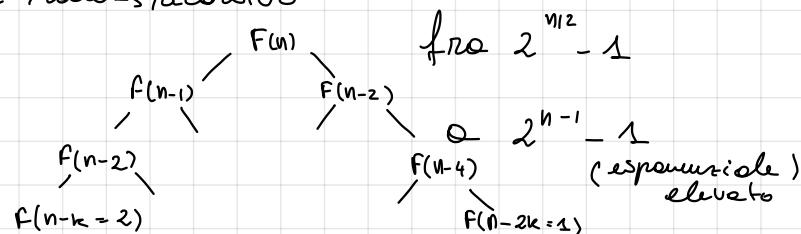
Do $\angle S \rightarrow$ Es: Fibonacci ($1, 2, 3, 5, 8, \dots$)

$$\begin{cases} f_1 = 1 \\ f_2 = 1 \\ f_n = f_{n-1} + f_{n-2} \quad \forall n \geq 3 \end{cases}$$

$$1^{\circ} \text{ modo } f_n = \frac{1}{\sqrt{5}} (\phi^n - \bar{\phi}^n)$$

$$\phi = \frac{1 + \sqrt{5}}{2} \quad \hat{\phi} = \frac{1 - \sqrt{5}}{2}$$

2° Modo → ricorsivo



3º Modo \rightarrow alg. iterativo array

costo: lineare, inferiore al 2^{do} ma ha uno svantaggio: uso + mem del necessario perché usa array [n]
↳ bad costo spaziale

4° modo → iterativo. Crea l'insieme numeri
sempre mentre l'array intero ha solo i 2 che
vengono usati (a, b)

3. $n-2$ = linear uses more memory

6º Modo \rightarrow logarítmico

| 5° Modo: use una matrice
rel. matematica

$$\begin{aligned} J &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \rightarrow \text{per induktiv } J^{n-1} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} = \\ &= \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix} \end{aligned}$$

LEZIONE 7: Complessità - formule

Notazione: Costo computazionale (prescindiamo dalla macchina e tempo exec)

Legge di Moore: la potenza dei calcolatori raddoppia ogni 1/2 anni (p. potenza)

20m 2P

4 am 4P

6 anni 8 P

1

2n omi 2'p

Gressita esponentiale

enziole, rimane esponenziale!

Costo: il costo di un algoritmo viene espresso come funzione di un parmetro n .
ci interessa fare un'analisi per valori "grandi" di n .

NB: anche se un alg non è efficiente ma il valore del parametro n è piccolo,
l'algoritmo risulta comunque efficiente

Ci interessano $\begin{cases} \text{costo temporale} \\ \text{costo spaziale} \end{cases}$

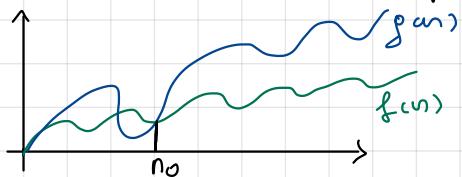
$$T: \mathbb{N} \rightarrow \mathbb{R}^+ \quad n \mapsto T(n)$$

$$S: \mathbb{N} \rightarrow \mathbb{R}^+ \quad s \mapsto S(s)$$

Notazione asintotica:

• $f(n) = O(g(n)) \iff \exists \alpha > 0, \exists n_0 \in \mathbb{N} \text{ t.c. } \forall n \geq n_0 \quad f(n) \leq \alpha \cdot g(n)$

$\hookrightarrow f(n) \text{ è di ordine non superiore a } g(n)$



$O(g(n))$ rappresenta un insieme di funzioni
l'insieme di quelle di ordine non superiore a $g(n)$

Sarebbe più corretto $f(n) \in O(g(n))$

Esempio: $f_1(n) = 5n^2$ voglio dim. che $\exists \alpha > 0, \exists n_0 \in \mathbb{N} \text{ t.c. } \forall n \geq n_0 \quad f_1(n) \leq \alpha \cdot n^2$

è sufficiente considerare che: $\alpha = 5$ $n_0 = 1$ infatti $\forall n \geq 1 \quad 5 \cdot n^2 \leq 5 \cdot n^2$

Esempio: $f_2(n) = 5n^2 + n$ voglio dim. che $f_2(n) \in O(n^2)$

dovendo dim. che $\exists \alpha > 0, \exists n_0 \in \mathbb{N} \text{ t.c. } \forall n \geq n_0 \quad f_2(n) \leq \alpha \cdot n^2$

Se considero l'equazione precedente: $5n^2 + n \leq \alpha \cdot n^2 \iff 5n + 1 \leq \alpha n \iff \alpha \geq 5 + \frac{1}{n}$
 \hookrightarrow è sufficiente considerare $\alpha = 6$, $n_0 = 1 \rightarrow \forall n \geq 1 \quad 5 \cdot n^2 + n \leq 6 \cdot n^2$

Osservazione: Se $f(n) = O(n^2)$ allora $f(n) = O(n^3)$

Dimo: ipotesi: $\exists \alpha > 0, \exists n_0 \in \mathbb{N} \text{ t.c. } \forall n \geq n_0 \quad f(n) \leq \alpha \cdot n^2$

poiché $n^2 \leq n^3 \quad \forall n \in \mathbb{N}$

Se è vera l'ipotesi allora è vero che $\exists \alpha > 0, \exists n_0 \in \mathbb{N} \text{ t.c. } \forall n \geq n_0 \quad f(n) \leq \alpha \cdot n^3$

Generalizzato: se $f(n) = O(n^k)$ allora $f(n) = O(n^h) \quad \forall h \geq k$

ip: $\exists \alpha > 0, \exists n_0 \in \mathbb{N} \text{ t.c. } \forall n \geq n_0 \quad f(n) \leq \alpha \cdot n^k$

\hookrightarrow poiché $n^k \leq n^h \quad \forall n \in \mathbb{N} \quad h > k$

$\hookrightarrow \exists \alpha > 0, \exists n_0 \in \mathbb{N} \text{ t.c. } \forall n \geq n_0 \quad f(n) \leq \alpha \cdot n^h$

Funzioni polinomiali

Proposizione: Se $f(n)$ è un polinomio di grado k , allora $f(n) = O(n^k)$

Dimo: \times ip: $f(n) = c_0 + c_1 \cdot n + c_2 \cdot n^2 + \dots + c_k \cdot n^k$

tesi: $\exists \alpha > 0, \exists n_0 \in \mathbb{N}$ t.c. $\forall n \geq n_0 \quad f(n) \leq \alpha \cdot n^k$

osservo che $f(n) \leq \alpha \cdot n^k \iff \frac{c_0}{n^k} + \frac{c_1}{n^{k-1}} + \frac{c_2}{n^{k-2}} + \dots + \frac{c_k}{1} \leq \alpha$

poiché $\frac{c_i}{n^{k-i}} \leq c_i$ allora basta $\rightarrow \alpha = \sum_{i=0}^k c_i$ e $n_0 = 1$

Esempio: considero $f(n) = 3 \cdot n + 2$ voglio dim. che $f(n) \in O(n)$

devo dim. che $\exists \alpha > 0, \exists n_0 \in \mathbb{N}$ t.c. $\forall n \geq n_0 \quad f(n) \leq \alpha \cdot n$

usando la disugual. prec. $3 \cdot n + 2 \leq \alpha \cdot n \iff \alpha \geq 3 + \frac{2}{n}$

poiché $2 > \frac{2}{n} \quad \forall n \geq 1$ posso considerare $\alpha = 5$ $n_0 = 1$ infatti $\forall n \geq 1 \quad 3 \cdot n + 1 \leq 5 \cdot n$

Definizioni:

| | |
|------------------|-------------------------|
| $f(n) = O(1)$ | costante |
| Se $f(n) = O(n)$ | allora $f(n)$ è lineare |
| $f(n) = O(n^2)$ | quadratico |
| $f(n) = O(n^k)$ | polinomiale |

Def.: O grande

Se esiste finito il $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)}$ allora $f(n) = O(g(n))$

- non vale l'inverso, vale per le funzioni costanti

LEZIONE 8

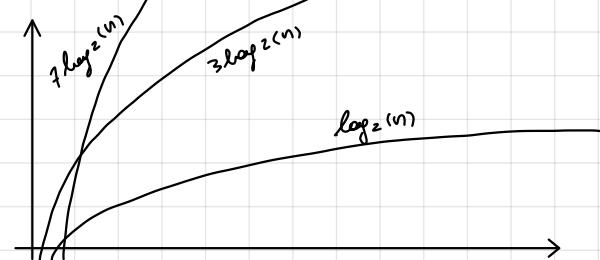
Funzioni logaritmiche:

esempio: Considero $f(n) = 3 \cdot \log_2(n) + 4$ Voglio dim. che $f(n) = O(\log_2(n))$

devo dimostrare che $\exists \alpha > 0, \exists n_0 \in \mathbb{N}$ t.c. $\forall n \geq n_0 \quad f(n) \leq \alpha \cdot \log_2(n)$

considero disugual. prec. $3 \cdot \log_2(n) + 4 \leq \alpha \cdot \log_2(n) \iff \alpha \geq 3 + \frac{4}{\log_2(n)}$

È sufficiente considerare che $n_0 = 2, \alpha = 3 + \frac{4}{\log_2(n_0)} = 7 \rightarrow$ infatti: $\forall n \geq 2, 3 \cdot \log_2(n) + 4 \leq 7 \cdot \log_2(n)$



Proposizione: Sia $b > 1$. Se $f(n) = O(\log_b(n))$, allora $f(n) = O(\log(n))$

Dim: per ip: $\exists \alpha > 0, \exists n_0 \in \mathbb{N}$ t.c. $\forall n \geq n_0 \quad f(n) \leq \alpha \cdot \log_b(n)$

$$\text{per props log } f(n) \leq \alpha \cdot \log_b(n) \iff f(n) \leq \underbrace{\frac{\alpha}{\log(b)}}_{\alpha} \cdot \log(n)$$

Concludo che $\exists \alpha' > 0, \exists n_0 \in \mathbb{N}$ t.c. $\forall n \geq n_0 \quad f(n) \leq \alpha' \cdot \log(n)$

Proposizione: Se $f(n) = O(\log(n))$ allora $f(n) = O(n)$

Per ip: $\exists \lim_{n \rightarrow +\infty} \frac{f(n)}{\log(n)} = l \in \mathbb{R}$

$$\text{quindi: } \lim_{n \rightarrow +\infty} \frac{f(n)}{n} = \lim_{n \rightarrow +\infty} \left(\frac{f(n)}{\log(n)} \cdot \frac{\log(n)}{n} \right) = \lim_{n \rightarrow +\infty} \frac{f(n)}{\log(n)} \cdot \lim_{n \rightarrow +\infty} \frac{\log(n)}{n} = l \cdot 0 = 0$$

Proposizione: Se $k > 1$ e $f(n) = O(\log^k(n))$ allora $f(n) = O(n)$

dim: uso la dim precedente

Proposizione: Se $f(n) = O(\log(n))$ allora $f(n) = O(n^\epsilon) \quad \forall \epsilon > 0$

dim: segue dalla precedente con $\epsilon = \frac{1}{k}$

Funzioni esponenziali:

esempio: considero $f(n) = 2^n + 4$ voglio dim che $f(n) = O(2^n)$

devo dim che $\exists Q > 0, \exists n_0 \in \mathbb{N}$ t.c. $\forall n \geq n_0 \quad f(n) \leq Q \cdot 2^n$

$$\text{dalla prec. } 2^n + 4 \leq Q \cdot 2^n \iff Q \geq 1 + \frac{4}{2^n}$$

basta considerare $n_0=1 \quad Q=3 \rightarrow \forall n \geq 1 \quad 2^n + 4 \leq 3 \cdot 2^n$

Proposizione: Se $f(n) = O(z^n)$ allora $f(n) = O(\alpha^n)$ $\forall \alpha > z$

x ip $\exists \lim_{n \rightarrow +\infty} \frac{f(n)}{z^n} = l \in \mathbb{R}$

quindi $\lim_{n \rightarrow +\infty} \frac{f(n)}{\alpha^n} = \lim_{n \rightarrow +\infty} \left(\frac{f(n)}{z^n} \cdot \frac{z^n}{\alpha^n} \right) = \lim_{n \rightarrow +\infty} \frac{f(n)}{z^n} \cdot \lim_{n \rightarrow +\infty} \left(\frac{z}{\alpha} \right)^n = 0$

Proposizione: Sia $k > 0$, se $f(n) = O(n^k)$ allora $f(n) = O(z^n)$

x ip: $\exists \lim_{n \rightarrow +\infty} \frac{f(n)}{n^k} = l \in \mathbb{R}$

quindi $\lim_{n \rightarrow +\infty} \frac{f(n)}{z} = \lim_{n \rightarrow +\infty} \left(\frac{f(n)}{n^k} \cdot \frac{n^k}{z} \right) = \lim_{n \rightarrow +\infty} \frac{f(n)}{n^k} \cdot \lim_{n \rightarrow +\infty} \frac{n^k}{z} = 0$

Proposizione per (Fattoriale): Sia $f(n) = n!$ allora $f(n) = O(e^{n \cdot \log(n)})$

Dim: Vale Stirling: $\lim_{n \rightarrow +\infty} \frac{n!}{\sqrt{2\pi n} \left(\frac{n}{e} \right)^n} = 1$

$$\begin{aligned} \lim_{n \rightarrow +\infty} \frac{n!}{e^{n \cdot \log n}} &= \lim_{n \rightarrow +\infty} \frac{\sqrt{2\pi n} \left(\frac{n}{e} \right)^n}{e^{n \cdot \log(n)}} = \lim_{n \rightarrow +\infty} \frac{\sqrt{2\pi n} e^{\log \left(\frac{n}{e} \right)^n}}{e^{n \cdot \log(n)}} \\ &= \lim_{n \rightarrow +\infty} \frac{\sqrt{2\pi n} e^{n(\log(n)-1)}}{e^{n \cdot \log(n)}} = \lim_{n \rightarrow +\infty} \sqrt{\frac{2\pi n}{e^n}} = 0 \end{aligned}$$

Proprietà di O grande:

Se $f(n) = O(g(n))$ allora $\forall c > 0 \rightarrow c \cdot f(n) = O(g(n))$

Se $f_1(n) = O(g_1(n)) \in f_2(n) = O(g_2(n))$ allora $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$

LEZIONE 9

Le f che usiamo vanno da $\mathbb{N} \rightarrow \mathbb{R}^+$ → f. a val. positivi

O grande, ripasso

O grande $f(n) \in O(g(n)) \quad \exists \alpha > 0, \exists n_0 \in \mathbb{N} \text{ t.c. } \forall n \geq n_0 \quad f(n) \leq \alpha \cdot g(n)$
 $f(n)$ è di ordine non superiore a $g(n)$

oppure se $\exists \lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)}$ allora $f(n) = O(g(n))$

• Classi di complessità Θ

| | |
|-----------------------|----------------|
| $f(n) = O(1)$ | costante |
| $f(n) = O(\log(n))$ | logaritmica |
| $f(n) = O(n)$ | lineare |
| $f(n) = O(n \log(n))$ | pseudo lineare |
| $f(n) = O(n^2)$ | quadrati |
| $f(n) = O(n^k)$ | polinomiale |
| $f(n) = O(\alpha^n)$ | esponenziale |

• Classi di complessità Θ

| | |
|----------------------------|----------------|
| $f(n) = \Theta(1)$ | costante |
| $f(n) = \Theta(\log(n))$ | logaritmica |
| $f(n) = \Theta(n)$ | lineare |
| $f(n) = \Theta(n \log(n))$ | pseudo lineare |
| $f(n) = \Theta(n^2)$ | quadrati |
| $f(n) = \Theta(n^k)$ | polinomiale |
| $f(n) = \Theta(\alpha^n)$ | esponenziale |

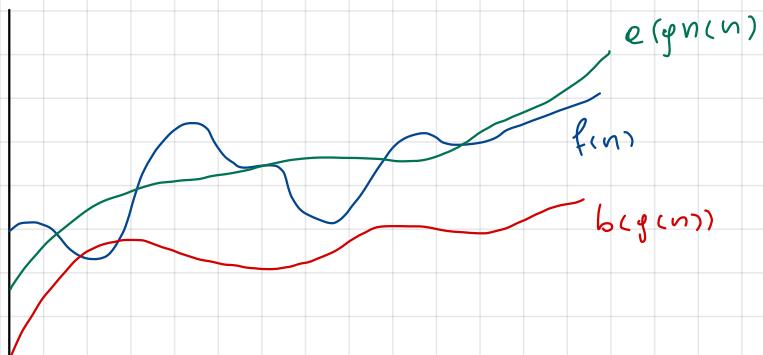
• theta grande Θ

$f(n) = \Theta(g(n)) \quad \text{se } \exists \alpha > 0, \exists b > 0, \exists n_0 \in \mathbb{N} \text{ t.c. } \forall n \geq n_0 \quad b \cdot g(n) \leq f(n) \leq \alpha \cdot g(n)$

$f(n)$ è di ordine uguale a $g(n)$

NB: È vero che se $f(n) = \Theta(g(n))$ allora $f(n) = O(g(n))$, ma non vale il contrario!

NB: dire che $f(n) = \Theta(g(n))$ è uguale a $f(n) = O(g(n))$ e $g(n) = O(f(n))$
ovvero le funzioni sono O l'una, dell'altra è quindi Θ (stesso ordine)



Def limite Θ

Se esiste $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = l \neq 0$
ed è finito

allora $f(n) = \Theta(g(n))$
- non vale l'inverso
- vale x funzioni costanti

Se $f(n) = \Theta(g(n))$ allora $g(n) = \Theta(f(n))$, ma non vale per Θ perché
il limite deve essere finito e $\neq 0$

Proposizione:

Sia $f(n)$ un polinomio di grado k . Allora $f(n) = \Theta(n^k)$

Dim: per ip: $f(n) = c_0 + c_1 \cdot n + c_2 \cdot n^2 + \dots + c_k \cdot n^k$

$$\lim_{n \rightarrow +\infty} \frac{f(n)}{n^k} = \lim_{n \rightarrow +\infty} \frac{c_k n^k}{n^k} = c_k \neq 0$$

Esempio:

Considero $f(n) = 3n + 2$, voglio dim che $f(n) \in \Theta(n)$

dovò dim che: $\exists a > 0, \exists b > 0, \exists n_0 \in \mathbb{N}$ t.c. $\forall n \geq n_0 \quad b \cdot n \leq f(n) \leq a \cdot n$

posso considerare $a = 5$ $n_0 = 1$

considero le diseguaglianze $b \cdot n \leq 3 \cdot n + 2 \iff b \leq 3 + \frac{2}{n}$

Infatti: $\forall n \geq 1 \quad 3 \cdot n \leq 3 \cdot n + 2 \leq 5 \cdot n$

Proposizione: non è vero che se $f(n) = \Theta(n^k)$ allora $f(n) = \Theta(n^h)$ $\forall h > k$

Dim: x ip esiste finito $\lim_{n \rightarrow +\infty} \frac{f(n)}{n^k} = l \neq 0$

Quindi $\lim_{n \rightarrow +\infty} \frac{f(n)}{n^h} = \lim_{n \rightarrow +\infty} \left(\frac{f(n)}{n^k} \cdot \frac{n^k}{n^h} \right) = \lim_{n \rightarrow +\infty} \frac{f(n)}{n^k} \cdot \lim_{n \rightarrow +\infty} n^{k-h} = \infty$

Proposizione: Sia $b > 1$. Se $f(n) = \Theta(\log_b(n))$, allora $f(n) = \Theta(\log(n))$

Dim: x ip se esiste finito $\lim_{n \rightarrow +\infty} \frac{f(n)}{\log_b(n)} = l \neq 0$

Quindi: $\lim_{n \rightarrow +\infty} \frac{f(n)}{\log(n)} = \lim_{n \rightarrow +\infty} \left(\frac{f(n)}{\log_b(n)} \cdot \log_b(e) \right) = l \cdot \log_b(e) \neq 0$

Proposizione: Non è vero che se $f(n) = \Theta(\log(n))$, allora $f(n) = \Theta(n)$

Dim: x ip, se \nexists finito $\lim_{n \rightarrow +\infty} \frac{f(n)}{\log(n)} = l \neq 0$

Quindi: $\lim_{n \rightarrow +\infty} \frac{f(n)}{n} = \lim_{n \rightarrow +\infty} \left(\frac{f(n)}{\log(n)} \cdot \frac{\log(n)}{n} \right) = \lim_{n \rightarrow +\infty} \frac{f(n)}{\log(n)} \cdot \lim_{n \rightarrow +\infty} \frac{\log(n)}{n} = \infty$

Proposizione: Sia $k > 0$, Non è vero che se $f(n) = \Theta(n^k)$ allora $f(n) = \Theta(2^n)$

Dim: x ip se \exists finito $\lim_{n \rightarrow +\infty} \frac{f(n)}{n} = l \neq 0$

$$\text{Quindi: } \lim_{n \rightarrow +\infty} \frac{f(n)}{2^n} = \lim_{n \rightarrow +\infty} \left(\frac{f(n)}{n^k} \cdot \frac{n^k}{2^n} \right) = \lim_{n \rightarrow +\infty} \frac{f(n)}{n^k} \cdot \lim_{n \rightarrow +\infty} \frac{n^k}{2^n} = 0$$

Proposizione: Sia $f(n) = n!$ Allora $f(n) = \Theta(\sqrt{n} \cdot e^{n \log(n) - n})$

Dim: Segue da Stirling $\lim_{n \rightarrow +\infty} \frac{n!}{\sqrt{2\pi n} \left(\frac{n}{e}\right)^n} = 1$

$$\text{Quindi: } \lim_{n \rightarrow +\infty} \frac{n!}{\sqrt{n} \cdot e^{n \log(n) - n}} = \lim_{n \rightarrow +\infty} \frac{\sqrt{2\pi n} \left(\frac{n}{e}\right)^n}{\sqrt{n} \cdot e^{n \log(n) - n}} =$$

$$\sqrt{2\pi} \lim_{n \rightarrow +\infty} \frac{e^{\log\left(\frac{n}{e}\right)^n}}{e^{n \log(n) - n}} = \sqrt{2\pi} \lim_{n \rightarrow +\infty} \frac{e^{n(\log n - \log e)}}{e^{n \log n - n}} = \sqrt{2\pi}$$

• Proprietà Θ grande

- Se $f(n) = \Theta(g(n))$ allora $\forall c > 0$, $c \cdot f(n) = \Theta(g(n))$
- Se $f_1(n) = \Theta(g_1(n))$ e $f_2(n) = \Theta(g_2(n))$ allora $f_1(n) \cdot f_2(n) = \Theta(g_1(n) \cdot g_2(n))$
- Se $f_1(n) = \Theta(g_1(n))$ e $f_2(n) = \Theta(g_2(n))$ allora
 $f_1(n) + f_2(n) = \Theta(g_1(n) + g_2(n)) = \Theta(\max\{g_1(n), g_2(n)\})$
- Se $g_1(n) = \Theta(g_2(n))$ allora $f_1(n) + f_2(n) = \Theta(g_2(n))$
- Se $g_2(n) = \Theta(g_1(n))$ allora $f_1(n) + f_2(n) = \Theta(g_1(n))$

Esempi:

Se $f_1(n) = \Theta(n)$ e $f_2(n) = \Theta(n^3)$ allora $f_1(n) + f_2(n) = \Theta(n + n^3) = \Theta(n^3)$

Se $f_1(n) = \Theta(\log(n))$ e $f_2(n) = \Theta(n)$ allora $f_1(n) + f_2(n) = \Theta(\log(n) + n) = \Theta(n)$

Se $f_1(n) = \Theta(n)$ e $f_2(n) = \Theta(2^n)$ allora $f_1(n) + f_2(n) = \Theta(n + 2^n) = \Theta(2^n)$

LEZIONE 10

- Costo di un algoritmo: quantifica le risorse necessarie per l'esecuzione di un algoritmo in termini di
 - Numero operazioni \rightarrow costo temporale
 - Sposto di memoria \rightarrow costo spaziale

Costo \rightarrow indipendente dal pc. wato

| | | |
|-----------|------------------------------------|--|
| temporale | $T(n) \rightarrow$ # passi base | $T: \mathbb{N} \rightarrow \mathbb{R}^+$ |
| spaziale | $S(n) \rightarrow$ # celle memoria | $S: \mathbb{N} \rightarrow \mathbb{R}^+$ |

[... Riposo...]

Proprietà di \mathcal{O} grande:

- Se $f(n) = \mathcal{O}(g_1(n))$ allora $\forall c > 0 \quad c \cdot f(n) = \mathcal{O}(g_1(n))$
- Se $f_1(n) = \mathcal{O}(g_1(n))$ e $f_2(n) = \mathcal{O}(g_2(n))$ allora $f_1(n) \cdot f_2(n) = \mathcal{O}(g_1(n) \cdot g_2(n))$
- Se $f_1(n) = \mathcal{O}(g_1(n))$ e $f_2(n) = \mathcal{O}(g_2(n))$ allora
$$f_1(n) + f_2(n) = \mathcal{O}(g_1(n) + g_2(n)) = \mathcal{O}(\max\{g_1(n), g_2(n)\})$$
- Se $g_1(n) = \mathcal{O}(g_2(n))$ allora $f_1(n) + f_2(n) = \mathcal{O}(g_2(n))$
- Se $g_2(n) = \mathcal{O}(g_1(n))$ allora $f_1(n) + f_2(n) = \mathcal{O}(g_1(n))$

Esempi proprietà:

- Se $f_1(n) = \mathcal{O}(n)$ e $f_2(n) = \mathcal{O}(n^3)$ allora $f_1(n) + f_2(n) = \mathcal{O}(n+n^3) = \mathcal{O}(n^3)$
- Se $f_1(n) = \mathcal{O}(\log(n))$ e $f_2(n) = \mathcal{O}(n)$ allora $f_1(n) + f_2(n) = \mathcal{O}(\log(n) + n) = \mathcal{O}(n)$
- Se $f_1(n) = \mathcal{O}(n)$ e $f_2(n) = \mathcal{O}(2^n)$ allora $f_1(n) + f_2(n) = \mathcal{O}(n+2^n) = \mathcal{O}(2^n)$

[\mathcal{O} grande...]

Calcolo della complessità:

Usiamo il criterio di costo uniforme

- **Caso pessimo**: dati in ingresso massimizzano tempo di exec
- **Caso medio**: media dei tempi delle possibili istanze
 - ↑
 - ↓
- **Caso ottimo**: dati in ingresso minimizzano il costo di exec

Aseguimento: il costo è $T(n) \begin{cases} O(1) & \text{se privo di chiamate a funz.} \\ \tilde{T}(n) & \text{se ha chiamate a funz. eseguibili in } \tilde{T}(n) \end{cases}$

es: $\text{int } x; \quad O(1)$
 $\text{int } t; \quad O(1)$
 $x = 1; \quad O(1)$
 $t = (x+3)*8; \quad O(1)$
 $x = f(t); \quad \tilde{T}(n)$

Selezione: costo delle valutazioni di E: $T_0(n)$ $\text{if}(E)$
costo di S_1 : $T_1(n)$ Si
costo di S_2 : $T_2(n)$ else
 $\sim S_2$

Costo caso peggiore $T_0(n) + \max\{T_1(n), T_2(n)\}$

Se $T_0(n) = O(f_0(n))$, $T_1(n) = O(f_1(n))$, $T_2(n) = O(f_2(n))$

allora costo caso peggiore $O(f_0(n)) + \max\{O(f_1(n)), O(f_2(n))\} =$
 $= O(\max\{f_0(n), f_1(n), f_2(n)\})$

Cicli: costo valutazione di E: $T_0(n)$ $\text{while}(E)$
costo S S
num. iterazioni

Costo $\rightarrow T_0(n) + i(n)(T_0(n) + T_1(n))$

Se $T_0(n) = O(f_0(n))$, $T_1(n) = O(f_1(n))$, $i(n) = O(f(n))$

allora il costo è $O(f_0(n)) + O(f(n)) \cdot (O(f_0(n)) + O(f_1(n))) =$
 $= O(f(n) \cdot \max\{f_0(n), f_1(n)\})$

Esempi:

```
int fattoriale (int n) {
    int r=1;
    int s=1;
    while (i≤n) {
        T_0(n)   C(0)   O(1)   } C_1 O(1), ma quante volte viene esec.
        r=r*i;
        i++;
    }
    return r;
}
```

$$T_0(n) + i(n) = C_1 + (n \cdot C_0 + C_1)$$

$\downarrow O(1)$
 $\downarrow O(1)$

$C_1 O(1)$, ma quante volte viene esec.
 $\sim O(n)$ al massimo $O(1)$

caso peggiore $O(n)$

Cicli for: costo volutazionale $E_1 : t_1(n)$
 $E_2 : T_2(n)$
 $E_3 : T_3(n)$ for(E_1, E_2, E_3)
 $S : T_4(n)$ S
 iterazioni: $i(n)$

Costo $t_1(n) + t_2(n) + i(n)(t_2(n) + t_3(n) + t_4(n))$

Se $t_i(n) = O(f_i(n)) \forall i \in \{1, 2, 3, 4\}$, $i(n) = O(f(n))$

Allora il costo e':

$$O(f_1(n) + f_2(n) + O(f(n)) \cdot \max \{f_2(n), f_3(n), f_4(n)\}) \\ = O(\max \{t_1(n) + t_2(n), f(n) \cdot \max \{t_2(n), t_3(n), t_4(n)\}\})$$

int fattoriale (int n) {

 int r = 1;

 int s = 1;

 for (int i = 1; i < n; i++) $\rightarrow O(n) \cdot O(n) = O(n)$
 $r = r * i;$

}

 return r;

}

LEZIONE 11

In questa lezione non c'e' video ma solo come calcolare il costo temporale

- Somma C $\rightarrow t(n) \rightarrow n$ e' una max che vogliamo sommare

- Potenza C $\rightarrow T(n) \rightarrow n$ e' il valore dell'esponente

- Fibonaci C $\rightarrow T(n) \rightarrow n$ e' l' n -esimo numero della successione

1) Somma C



```

int somma1(int n){
    int s = 0;
    for (int i = 1; i <= n; i++)
        s = s + i;
    return s;
}

int somma2(int n){
    if (n < 0)
        return 0;
    return n * (n + 1) / 2;
}

int main () {
    int n = 10;
    int s1, s2;

    s1 = somma1(n);
    printf("La somma vale: %d \n", s1);

    s2 = somma2(n);
    printf("La somma vale: %d \n", s2);
}

```

→ eseguimento, costo costante / inizializz

E_1 $O(1)$ → costo costante / inizializz
 E_2 $t_1(n) = O(1)$ → costo exec volutor $t_1(n) = O(1)$ perché costante
 $t_2(n) = O(n)$ → costo $t_2(n) = O(n)$ perché no dichiarata funz.
 $t_3(n) = O(n)$ → dipende da $n \rightarrow O(n)$

$t_4(n) = O(1)$

→ è solo $O(1)$ perché è un return

$O(n)$ vs $O(1)$

1 2

Il secondo algoritmo è più efficiente

$O(1)$ vince!

Nel primo alg, il costo di n cresce linearmente al crescere di n

2) Potenza C

```

double potenza1(double x, int n){
    double z = 1.0; O(1)
    int i = 1; O(1)
    guardia + statement
    if (n < 0) {
        printf("Errore!\n");
        return 0;
    }
    to(n)
    while (i <= n) {
        z = z * x;
        i++; t_1
    }
    return z;
}

double potenza2(double x, int n){
    double z = 1.0; ] oss
    if (n < 0) { guardia
        printf("Errore!\n");
        return 0;
    }
    while (n != 0){ ]
        if (n&1)
            z = z*x;
        E1 S x = x*x; O(1)
        E2 S n = n/2; O(1)
    }
    return z; cost
}

```

→ eseguimenti costanti

→ costo guardia cost + 5 cost = $O(1)$

n iterazioni costo statemente

$t_0(n) + i(n) \cdot (t_0(n) + t_1(n))$ entrambi t_0 e t_1 sono costanti:

$$t_0(n) + i(n) = t_0(n) + O(n) = O(n)$$

→ costante $O(1)$

$O(n)$ vs $O(\log n) \rightarrow O(\log n)$ vince!

1 2

governo

costo val, expr + $i(n) \cdot (t_1 + t_0)$ / if e cost / S e cost / E, e E2 cost

$$t_0(n) = O(1)$$

n dimezzato a ogni iterazione, dopo k -vinte it avremo $\frac{n}{2^k}$. Ultimo while con $n=1$

$$\hookrightarrow \text{con } k=1 \text{ } n=2^k \rightarrow k = \log_2 n$$

$$i(n) = O(\log_2 n) \text{ costo while } O(\log_2 n)$$

$$\Rightarrow \text{costo tot } O(\log n)$$

3) Fibonacci C



```
double fibonacci_1 (int n) {  
    if (n <= 0) → cost  
        return -1; → cost } O(1)  
  
    return 1 / sqrt(5) * (potenza_intera((1+  
    - potenza_intera((1-  
diamente →  
})
```

\Rightarrow costo costante $O(1)$

$\rightarrow O(\log n)$ per costo return (chiama la funzione) quindi costo TOT

$\Rightarrow \underbrace{O(\log n)}$

```

int fibonacci_2 (int n) { → recursive !
    if (n <= 0)
        return -1;
    else if (n <= 2)
        return 1;
    else
        return fibonacci_2(n-1) + fibonacci_2(n-2);
}

int fibonacci_3 (int n) {
    if (n <= 0) } cost
    return -1;

    int F[n]; → array cost

    F[0] = 1; } → subsequent cost
    F[1] = 1; } → subsequent cost
    for (int i = 2; i < n; i++)
        F[i] = F[i-1] + F[i-2];
    return F[n-1];
}

```

} anche se tutto fosse costante, a livello di righe, il costo complessivo sarà maggiore di $C \cdot 2^{\frac{n}{2}}$
 $\hookrightarrow O(2^{\frac{n}{2}}) \rightarrow \boxed{O(2^n)}$

$O(2^n)$ vs $O(n) \rightarrow O(n), 2$, since
 \downarrow \downarrow

può essere tutto memorizzato in un array, che non si gode quel

Dipende da $i(n)$, quindi avremo $n-2$ iterazioni

$O(n)$. tutti costi costanti $\rightarrow O(n)$

```

int fibonacci_4 (int n) {
    int a = 1; }  

    int b = 1; }  

    int c; }  

    if (n <= 0) }  

        return -1; }  

    for(int i = 3; i <= n; i++) {  

        c = a + b; }  

        a = b; }  

        b = c; }  

    }  

    return b; }  

}

```

↳ *Initial cost*

↳ *OSS = cost*

↳ *Cost*

↳ *E₁*

↳ *E₂*

↳ *E₃*

$S \rightarrow E_3 \rightarrow E_2$ sono tutti costanti, ma quante volte vengono ripetute? $n-3$ iterazioni

$$O(1) + O(n) + O(1) = \underline{O(n)}$$

```

int fibonacci_5 (int n) {
    int M[2][2]; → cost
    if (n <= 0)
        return -1;
    M[0][0] = 1;
    M[0][1] = 1;
    M[1][0] = 1;
    M[1][1] = 0;
    for(int i = 2; i < n; i++) { } → cost
    // calcolo il prodotto tra una generica
    // matrice M e la matrice J e lo scrivo in M
    S
    M[0][0] = M[0][0] + M[0][1];
    M[0][1] = M[0][0] - M[0][1]; | cost
    M[1][0] = M[1][0] + M[1][1];
    M[1][1] = M[1][0] - M[1][1];
}
return M[0][0];
}

```

```

void quadrato(int M[][2]) {
    int a = M[0][0]*M[0][0] + M[0][1]*M[1][0];
    int b = M[0][0]*M[0][1] + M[0][1]*M[1][1];
    int c = M[0][0]*M[1][0] + M[1][0]*M[1][1];
    int d = M[0][1]*M[1][0] + M[1][1]*M[1][1];
}

```

```

M[0][0] = a;
M[0][1] = b;
M[1][0] = c;
M[1][1] = d;
}

```

```

void potenza_matrici(int M[], int k) {
    if (k > 1) { } cost
    potenza_matrici(M, k/2); cost
    quadrato(M); → oss C
}

```

```

K/2
log n
if (k % 2) {
    // calcolo il prodotto tra M e J e
    M[0][0] = M[0][0] + M[0][1];
    M[0][1] = M[0][0] - M[0][1];
    M[1][0] = M[1][0] + M[1][1];
    M[1][1] = M[1][0] - M[1][1];
}
}

```

```

int fibonacci_6 (int n) {
    int M[2][2]; cost
    if (n <= 0) cost
        return -1;
}

```

```

    M[0][0] = 1;
    M[0][1] = 1; cost
    M[1][0] = 1;
    M[1][1] = 0;
    potenza_matrici(M, n-1); cost
    return M[0][0]; cost
}

```

L'ordine di grandezza sarà quindi dato dal numero di volte in cui viene eseguito il for

2 inizio, poi $n \rightarrow n-2$ volte, quindi $\underline{\mathcal{O}(n)}$

tutto dipende da $\tilde{T(n)} \cdot \mathcal{O}(1)$
 \downarrow
 $\mathcal{O}(\log n)$

Quindi $T_{\text{tot}} = \mathcal{O}(\log n)$

END

LEZIONE 12

```

#include <stdio.h>
#include <stdbool.h>
#include <math.h>

bool primo(int n){
    for (int i = 2; i <= sqrt(n); i++) {
        if (n % i == 0) S = cost
        return false; -> cost
    }
    return true;
}

int main () {
    int n = 10;
    bool b;
    b = primo(n);
    if (b == 0)
        printf("Il numero non è primo \n");
    else
        printf("Il numero è primo \n");
}

```

```

double delta=1e-6;
double epsilon=1e-9;

double f(double x) { -> rappresenta un polinomio
    return x*x-2;
}
double bisezione(double a, double b) {
    double xmedio, ymedio;
    while(fabs(b-a)>delta) {
        xmedio=(a+b)/2;
        ymedio=f(xmedio);
        if (fabs(ymedio)<epsilon)
            return xmedio;
        if (ymedio*f(a)<0)
            b=xmedio;
        else
            a=xmedio;
    }
    return xmedio;
}
int main() {
    double a=0.0, b=2.0;
    printf("%s %f\n", "La soluzione vale: ", bisezione(a,b));
    return 0;
}

```

Per vedere se è primo, tra 2 e n : facciamo divisione intera e se c'è un divisore.
Meglio ancora, facciamo fra 2 e lo \sqrt{n} perché se n ha almeno un divisore banale, ne ha altri.

$E_3 \rightarrow \text{cost } E_1 + \text{cost } E_2 + \text{cost check}$

$$X = O(1) \cdot n \cdot \text{itazioni} \rightarrow \sqrt{n} - 2 = \lfloor \sqrt{n} \rfloor - 1 \quad \text{if } = O(\sqrt{n}) \\ O(n^{1/2}) \quad C > O(n)$$

Caso ottimo: in questo caso se n pari il vero, termina con $O(1)$
Caso medio: $O(\sqrt{n}/2)$
Caso peggiore: $O(\sqrt{n})$

$$f(0) = -2 \quad (\text{intervalli e segni discordi}) \\ f(2) = 2$$

Siccome tutti i costi di guardie, funz, statement ecc sono costanti, il costo del while è $O(1) \cdot n \cdot \text{itazioni}$

$$l = \text{ampiezza intervallo} \quad l = (b-a)$$

$$\text{la k-esima volta sarà } \left(\frac{l}{2}\right)^k$$

$$k-1 > \log_2 \frac{l}{\Delta} \Rightarrow k > 1 + \log_2 \frac{l}{\Delta}$$

$$1 + \log_2 l - \log_2 \Delta \Rightarrow n \text{ it } k > 1 + \log_2 l - \log_2 \Delta$$

$$\Leftrightarrow O(\log_2 l) \rightarrow O(\log_2 n)$$

$$\text{costo totale} = O(\log n)$$

LEZIONE 13 → ESERCIZI (automi, mt)

LEZIONE 14 → ESERCIZI (complessità, O, Θ)

LEZIONE 15 → ESERCIZI (calcolo costo complessità)

LEZIONE 16 → correzione esercizi

LEZIONE 17 - Master theorem

- Colore il costo di algoritmi ricorsivi \rightarrow Teorema dell'esperto

Definizione: Sia A un insieme. y è un minorante di A se: $\forall x \in A \quad y \leq x$

Si definisce inf A il maggiore fra tutti i minoranti di A .

$$\inf A = \max \{ y \text{ f.c. } \forall x \in A \ y \leq x \}$$

esempio:

Sia $A = [1, 2]$ o è un minorante di A

1.5 un nove è minorante

ϵ è un minorante di $A \rightarrow \epsilon$ anche $\inf A$ (e min A)

Sia $A = [1, 2]$ un minorante di A

1 è un minorante di $A \rightarrow 1 \in A$ (ma non min A)

Enunciato th. dell' esperto:

Siano $f(n), g(n)$ non decrescenti e siano $m \in \mathbb{N}, v \in \mathbb{N}$, con $m \geq 1$ $v \geq 2$ tali che:

$$f(n) = u \cdot f\left(\left\lceil \frac{n}{v} \right\rceil\right) + g(n)$$

Siamo: $b = \inf \{ h : g(n) = O(n^h) \}$

$$\varrho = \log_v(u) \quad (u = v^\varrho)$$

NB: I punti 2 e 3 volgono se vale le condizioni di repolarità.

Allora :

$$1 - f(n) = O(n^a) \quad \text{se } a > b$$

$$z - f(n) = \Theta(g(n)) \quad \text{se } a < b$$

$$3 - f(n) = O(g(n) \cdot \log(n)) \quad \text{se } \alpha = b$$

$\forall c \in \mathbb{R}, c > 1$ si les deux $f(c \cdot n) \geq c^b \cdot f(n)$

Dimostrazione:

Note: supponiamo che n sia potenza di v ($\exists k \in \mathbb{N} \text{ t.c. } n = v^k$)

sotto queste ipotesi: $\left[\frac{u}{v} \right] = \frac{u}{v}$ quindi l'ipotesi dell'esperto diventa $f(v^k) = u \cdot f(v^{k-1}) + g(v^k)$ (1)

Proposizione: se vale la nota⁽¹⁾, allora vale anche $f(v^k) = v^k \cdot f(1) + \sum_{j=0}^{k-1} v^j \cdot g(v^{k-j})$

dimo. prop: per induzione, $\forall l \in [0 \dots k-1]$ (l intero) vale: $f(v^k) = u^{l+1} \cdot f(v^{k-(l+1)}) + \sum_{j=0}^l u^j \cdot g(v^{k-j})$ (3)

(3) vole per $l=0$ (ipotesi)

(\hookrightarrow suppongo che valga per $l \geq 0$ (ip induttiva) e dimostra che vale per $l+1$)

$$\text{I}^{\text{p}} \text{ inductive: } f(v^k) = u^{k+1} \cdot f(v^{k-(l+1)}) + \sum_{j=0}^l u^j \cdot g(v^{k-j})$$

$$\text{ip proposizione} \rightarrow \text{tesi: } f(v^k) = u^{l+1} \cdot (u \cdot f(v^{k-(l+1)-1}) + g(v^{k-(l+1)})) + \sum_{j=0}^l u^j \cdot g(v^{k-j}) =$$

$$u^{\underline{l+1+i}} \cdot f(v^{k-(l+1+i)}) + \sum_{j=0}^{l+1} u^j \cdot g(v^{k-j})$$

Per ind. vole (3) \rightarrow se $l = k-1$ l'eq (3) e' $f(v^k) = u^k \cdot f(1) + \sum_{j=0}^{k-1} u^j \cdot g(v^{k-j})$ coincide con fesi precedente

Dimostrazione (caso $a > b$):

Sia $b < t < a$. Allora $f(n) = O(n^t)$ i.e.: $\exists c > 0, \exists n_0 \in \mathbb{N}$ t.c. $\forall n \geq n_0 \quad f(n) \leq c \cdot n^t$ (4)

Quindi, $\forall n (= v^k) \geq n_0$:

$$\begin{aligned} f(v^k) &\stackrel{(2)}{=} u^k \cdot f(1) + \sum_{j=0}^{k-1} u^j \cdot g(v^{k-j}) \stackrel{(4)}{\leq} u^k \cdot f(1) + \sum_{j=0}^{k-1} u^j \cdot c \cdot (v^{k-j})^t \\ &= u^k \cdot f(1) + c \cdot v^{kt} \cdot \sum_{j=0}^{k-1} \left(\frac{u}{v^t}\right)^j \end{aligned}$$

||

$$\begin{aligned} f(v^k) &\leq u^k \cdot f(1) + c \cdot v^{kt} \cdot \sum_{j=0}^{k-t} \left(\frac{u}{v^t}\right)^j \\ &= u^k \cdot f(1) + c \cdot v^{kt} \cdot \frac{u^k - v^{kt}}{u^t - 1} \cdot \frac{1}{\frac{u}{v^t} - 1} \\ &= u^k \cdot f(1) + c(u^k - v^{kt}) \cdot \frac{1}{\frac{u}{v^t} - 1} \\ &= u^k \left(f(1) + c \cdot \frac{1}{\frac{u}{v^t} - 1}\right) - c \cdot v^{kt} \cdot \frac{1}{\frac{u}{v^t} - 1} \\ &\leq u^k \left(f(1) + c \cdot \frac{1}{\frac{u}{v^t} - 1}\right) \quad \text{osservare che } n = v^k \rightarrow \\ &\quad \rightarrow n^a = (v^k)^a = (v^a)^k = u^a \end{aligned}$$

Osservazione

- $t < a \rightarrow v^t < v^a = u \rightarrow \frac{u}{v^t} > 1$
- Sia $q \in \mathbb{R}$ vale la seguente eguaglianza:

$$\sum_{j=0}^{k-1} q^j = \frac{q^k - 1}{q - 1}$$
- Considero $q = \frac{u}{v^t}$ nella sommatoria prec.

$$\sum_{j=0}^{k-1} \left(\frac{u}{v^t}\right)^j = \frac{u^k - 1}{u^t - 1} = \frac{u^k - v^{kt}}{u^t - v^t} = \frac{u^k - v^{kt}}{v^{kt} - v^t} \cdot \frac{1}{\frac{u}{v^t} - 1}$$

↳ Sost. questo risultato nella diseguagli. precedente

Abbiamo dimostrato che $\forall n \geq n_0 \quad f(n) \leq (f(1) + c \cdot \frac{1}{\frac{u}{v^t} - 1}) \cdot n^a$

Cioè esiste una $L > 0$ t.c. $\forall n \geq n_0 \quad f(n) \leq L \cdot n^a$ cioè $f(n) = O(n^a)$

→

Dimostrazione (caso $a < b$)

Per regolarità ($c=v$): $g(v \cdot v^{k-1}) \geq v^b \cdot g(v^{k-1})$

Dimo per induzione che $\forall j \in [0 \dots k] \quad g(v^k) \geq v^{jb} \cdot g(v^{k-j})$

l'uguaglianza precedente vale $\forall j=1$, supponiamo che valga per j (ip. induc.) e dimo che vale per $j+1$

$$g(v^k) > v^{jb} \cdot g(v^{k-j}) \geq v^{jb} \cdot v^b \cdot g(v^{k-j-1}) \quad \text{quindi} \quad g(v^k) > v^{(j+1)b} \cdot g(v^{k-(j+1)})$$

||

$$\begin{aligned} f(v^k) &\stackrel{(2)}{=} u^k \cdot f(1) + \sum_{j=0}^{k-1} u^j \cdot g(v^{k-j}) \stackrel{(5)}{\leq} u^k \cdot f(1) + \sum_{j=0}^{k-1} u^j \cdot \frac{g(v^k)}{v^{jb}} \\ &= u^k \cdot f(1) + g(v^k) \cdot \sum_{j=0}^{k-1} \left(\frac{u}{v^b}\right)^j \leq u^k \cdot f(1) + g(v^k) \cdot \sum_{j=0}^{+\infty} \left(\frac{u}{v^b}\right)^j \end{aligned}$$

• $a < b \rightarrow u = v^a < v^b \rightarrow \frac{u}{v^b} < 1$

• Sia $q \in \mathbb{R}, |q| < 1$ vale l'uguaglianza $\sum_{j=0}^{+\infty} q^j = \frac{1}{1-q}$

• considero $q = \frac{u}{v^b}$ nella serie precedente

$$\sum_{j=0}^{+\infty} \left(\frac{u}{v^b}\right)^j = \frac{1}{1 - \frac{u}{v^b}} = \frac{v^b}{v^b - u}$$

• sostituisco il risultato nella precedente →

$$f(v^k) \leq u^k \cdot f(1) + g(v^k) \cdot \frac{v^b}{v^b - u}$$

Abbiamo dimostrato che $f(n) \leq n^a \cdot f(1) + g(n) \cdot \frac{v^b}{v^b - u}$ poiché $a < b$ allora $n^a = O(g(n))$ i.e.: $\exists c > 0, \exists n_0 \in \mathbb{N}$ t.c. $\forall n \geq n_0 \quad n^a \leq c \cdot g(n)$

Quindi possiamo concludere che $\forall n \geq n_0 \quad f(n) = (c \cdot f(1) + \frac{v^b}{v^b - u}) g(n)$ cioè $f(n) = \Theta(g(n))$

→

Dim $\coso a = b$

Poiché $a = b$, allora $u = v^a = v^b$ e $u^k = u^a = n^b$.

Dalle (2) e dalle regolarità avevamo ottenuto: $f(v^k) \stackrel{(6)}{\leq} u^k \cdot f(1) + g(v^k) \cdot \sum_{j=0}^{k-1} \left(\frac{u}{v^b}\right)^j = n^b \cdot f(1) + g(v^k) \cdot k$

quindi $f(n) \leq f(1) \cdot n^b + \log v(n) \cdot g(n)$ osservi che per come è definito b , $\forall \epsilon > 0$

$n^{b-\epsilon} = O(g(n))$ quindi $n^b = O(g(n) \cdot n^\epsilon) = O(g(n) \cdot \log(n))$ quindi $f(n) = O(g(n) \cdot \log(n))$

LEZIONE 18 - Esercizi: (Master theorem)

LEZIONE 19 -

• Relazioni di ricorrenza lineari e omogenee:

Alle volte per algoritmi di tipo ricorsivo non è possibile usare il master theorem. Si usano quindi le rel. di ric. lineari e omogenee.

- Una r.d.r.l.o di ordine r è una relazione del tipo:

$$f(n) = c_1 \cdot f(n-1) + c_2 \cdot f(n-2) + \dots + c_r \cdot f(n-r) \quad (1)$$

c_1, \dots, c_r costanti

Il polinomio caratteristico delle rel. di ricorrenza lin. omog. di ordine r è:

$$P(x) = x^r - c_1 \cdot x^{r-1} - c_2 \cdot x^{r-2} - \dots - c_r$$

$$P(\lambda) = 0$$

• Teorema: Se consideriamo una r.d.r.l.o. di ordine r e sia λ una radice del suo polinomio caratteristico con molteplicità m . Allora:

$\lambda^n, \lambda^n \cdot n, \lambda^n \cdot n^2, \dots, \lambda^n \cdot n^{m-1} \rightarrow$ Sono soluzioni delle relazioni

OSS: Al variare di λ tra le radici del polinomio caratteristico si ottengono r soluzioni di questo tipo (sol. base rel.).

Le sol. delle rel. di ric. sono tutte le comb. lineari (a coeff. complessi) delle r soluzioni base.

Corollario: Se il polinomio caratteristico di una rel. di ric. omog. ha n radici distinte $\lambda_1, \dots, \lambda_n$

Allora la soluzione generale della relazione di ricorrenza è:

$$c_1 \cdot \lambda_1^n + c_2 \cdot \lambda_2^n + \dots + c_n \cdot \lambda_n^n$$

con c_1, \dots, c_n cost. complesse.

Esempio:

rel. ric. lin. om.
 $f(n) = 3 \cdot f(n-1)$ $x^2 - c_1 \rightarrow$

$$\text{pol. caratt. } p(x) = x - 3 \rightarrow \text{le sol di } p(x) = 0 \text{ e } x = 3$$

Quindi, tutte le sol. dell'equazione di ricorrenza sono: $f(n) = q \cdot 3^n$

fine esercizio - cerco i conti in più

Suppongo la condizione $f(1) = 1 \iff 3q = 1 \iff q = 1/3$ \rightarrow Verifica

Quindi, le sol delle r.d.r. con $f(1) = 1$ è $f(n) = \frac{1}{3} \cdot 3^n$

Esempio 2:

$$x^2 - c_1 = x^2 - 4$$

$$f(n) = 2^{\overline{n}} \cdot f(\overline{n-2}) \quad \text{pol. caratt. } p(x) = x^2 - 4 \quad \text{le sol di } p(x) = 0 \text{ sono } x=2, x=-2$$

Quindi le sol. sono $f(n) = q \cdot 2^n + b(-2)^n$

fine

Suppongo $f(1) = 2$ e $f(2) = 8$

$$f(1) = 2 \iff 2q - 2b = 2 \iff q - b = 1$$

$$f(2) = 8 \iff 4q - 4b = 8 \iff q + b = 2$$

Vel che soddisfano le condizioni sono: $q = 3/2$ $b = 1/2$

Quindi le sol con $f(1) = 2$ e $f(2) = 8$ è

$$f(n) = \frac{3}{2} \cdot 2^n + \frac{1}{2} (-2)^n$$

• Relazioni di ricorrenza lineari e non omogenee:

es: ordine 2 $f(n) = c_1 f(n-1) + c_2 f(n-2) + \dots + c_r f(n-r) + h(n)$ c'è una fin più

Proposizione: Le sol generale di una r.d.r.l. non omogenea si ottiene aggiungendo una sol particolare alla sol. generale della sua parte omogenea.

Teorema: sia $h(n)$ un polinomio di grado k :

- Se 1 non è radice del polinomio caratteristico, una soluzione particolare è un polinomio di grado k del tipo $f(n) = a_0 + a_1 \cdot n + \dots + a_k \cdot n^k$

Se 1 è una radice del polinomio caratteristico di molteplicità μ , una sol. particolare è del tipo $f(n) = n^\mu (a_0 + a_1 \cdot n + \dots + a_\mu \cdot n^\mu)$

Corollario: Se $b(n)$ è costante, ma sol porticolore delle ricorrenze non omogenee è:

- una cost., se 1 non è radice del polinomio caratteristico
- una soluzione del tipo $\alpha \cdot n^k$, se 1 è una radice del pol. caract. di mult. k .

$$x - 3 = 0 \quad x = 3$$

Esempio 1) $f(n) = 3 \cdot f(n-1) + 2$ $p(x) = 0 \Leftarrow x = 3$ sol porticolore $\alpha \cdot 3^n$

Quindi tutte le sol dell'eq. di ric. sono $f(n) = Q \cdot 3^n - 1$

Impongo $f(1) = 1 \Leftrightarrow 3Q - 1 = 1 \Leftrightarrow Q = 2/3$

Quindi le sol dell'eq. di ric con $f(1) = 1$ è $f(n) = \frac{2}{3} \cdot 3^n - 1$

Esempio 2) $f(n) = f(n-1) + 2$ $p(x) = x - 1$ $p(x) = 0 \rightarrow$ sol $x = 1$

Mus sol porticolore delle non omogenee è del tipo $\alpha \cdot n$

Impongo che $\alpha \cdot n$ soddisfi la rel $\alpha \cdot n = \alpha(n-1) + 2$ da cui $\alpha = 2$

Quindi tutte le sol dell'eq. sono $f(n) = \alpha + 2 \cdot n$

Impongo $f(1) = 1 \Leftrightarrow \alpha + 2 = 1 \Leftrightarrow \alpha = -1$

Quindi le sol dell'eq. di ricorrenza con $f(1) = 1$ è $f(n) = 2 \cdot n - 1$

Esempio 3) $f(n) = 2 \cdot f(n-1) + n$ $p(x) = x - 2$ $p(x) = 0 \rightarrow$ sol $x = 2$

Mus sol porticolore delle non omogenee è del tipo $\alpha_0 + \alpha_1 \cdot n$

Impongo che $\alpha_0 + \alpha_1 \cdot n$ soddisfi la rel: $\alpha_0 + \alpha_1 \cdot n = 2 \cdot (\alpha_0 + \alpha_1 \cdot (n-1)) + n$ da cui $\alpha_0 = -2$

Quindi tutte le sol dell'eq. di ric. sono $f(n) = Q \cdot 2^n - 2 - n$

Impongo $f(1) = 1 \Leftrightarrow Q = 2$

Quindi le sol dell'eq. con $f(1) = 1$ è $f(n) = 2 \cdot 2^n - 2 - n$

[+ esercizi polf]

LEZIONE 20

Problemi con algoritmi ricorsivi:

1) Calcolare $\sum_{i=0}^n i$ oss. avevamo usato un alg iterativo

summa.c

```
#include <stdio.h>
int summa(int n){
```

if ($n \leq 0$) return 0; \rightarrow costo guardia + return = c
if ($n == 1$) return 1; \rightarrow " " " " = c
else return $\underline{n} + \underline{\text{summa}(n-1)}$; costo else c + n \rightarrow $\underline{\underline{O(n)}}$

}

↓ risolvere la rel.

$$T(n) = \begin{cases} c_1 & n=1 \\ T(n-1) + c_2 & n>1 \end{cases}$$

costo ottimo $O(1)$
costo peggiore $O(n)$

int main()

int n=10;

int s

s = summa(n);

[...]

$$T(n) = \begin{cases} c_1 & \text{tollerabile per le case} \\ T(n-1) + c_2 & \text{augmentare} \Rightarrow T(n) = T(n-1) \end{cases}$$

$p(x) = x-1$ ha radice 1 \rightarrow 1 sol pert. e' $T(n) = \alpha n$
impongo che sia sol $\alpha \cdot n = \alpha(n-1) + c_2 \leftrightarrow \alpha = c_2$

da sol sono $T(n) = \alpha + c_2 \cdot n$ quindi $T(n) = O(n)$
costo lineare

2) N-esimo num succ. fibonacci

Calcolare funzione dove

$$\begin{cases} F_1 = 1 \\ F_2 = 1 \\ F_n = F_{n-1} + F_{n-2} \quad \forall n \geq 3 \end{cases}$$

int fibonacci_2(int n){

if ($n \leq 0$) return 0 \rightarrow costante $O(1)$
else if ($n \leq 2$) return 1; \rightarrow costante $O(1)$
else return fibonacci_2($\underline{n-1}$) + fibonacci_2($\underline{n-2}$);
 $\downarrow \rightarrow$

$$T(n) = \begin{cases} c_1 & n=1 \\ c_2 & n=2 \\ T(n-1) + T(n-2) + c & n>2 \end{cases}$$

eq. ric. lin. non omog. ordine 2

$p(x) = x^2 - x - 1$ ha radici $\frac{1+\sqrt{5}}{2}, \frac{1-\sqrt{5}}{2}$ una sol pert. $T(n) = -c_2$

$p(x) = 0 \rightarrow x^2 - x - 1 = 0$ \rightarrow $\frac{1+\sqrt{5}}{2}^n + b \cdot \left(\frac{1-\sqrt{5}}{2}\right)^n - c_2$

Quindi $O(n^2) \rightarrow$ esponenziale

3) Calcolo radice quadrata: Sia $f: D \rightarrow \mathbb{R}$ una funzione continua, esiste un unico $\epsilon \in [a, b]$ t.c. $f(\epsilon) = 0$. con $f(a) \cdot f(b) < 0$. Calcola ϵ

Sol: bisezione

1 → calcolo c a metà fra a e b

2 → se $f(c) = 0$ allora $c \epsilon \epsilon$

3 → altrimenti se $f(a) \cdot f(c) < 0$ si cerca ϵ fra a - c
 $f(b) \cdot f(c) < 0$ si cerca ϵ fra c - b

double delta = 1e-6;

double epsilon = 1e-8;

double f(double x) return x * x - 2;

double bisezione (double a, double b){
 double xmedio, ymedio;
 xmedio = (a + b) / 2;
 ymedio = f(xmedio); } parte costante

[if (fabs(b - a) < delta || fabs(ymedio) < epsilon)] guardia + verifica + return
 return xmedio; costante
 elseif (ymedio * f(a) < 0)] guardia + verifica + return
 return bisezione (a, xmedio);
 else] richiedono le fasi /
 return bisezione (xmedio, b); + 1/2
 concatenazione if else

$$T(l) = \begin{cases} c_1 & l \leq \delta \\ T\left(\frac{l}{2}\right) + c_2 & l > \delta \end{cases} \quad \begin{matrix} u=1 & v=2 \\ a = \log_2(1) = 0 & b = 0 \\ a = b \end{matrix}$$

Quindi $T(n) = O(\log(n))$ costo logaritmico

4) Sia v un vettore e k un valore. Stabilire se c'è un elemento di v che è k.

bool ricerca_l_r (int v[], int k, int n){

```
if (n == 1){  

    if (v[n-1] == k)  

        return true;  

    else return false;  
}
```

```
if (v[n-1] == k)  

    return true;  

else return ricerca_l_r (v, k, n-1);
```

$$T(n) = \begin{cases} c_1 & n=1 \\ T(n-1) + c_2 & n>1 \end{cases}$$

i lin. non adegua

$$T(n) = O(n)$$

cioè lineare (M.T.)

5) Sia v un vettore ordinato crescente e k un val. $\exists! e v = k$?

bool ricerca - 2 (int $v[]$, int k , int primo, int ultimo) {

 int med = (primo + ultimo) / 2;

 if ($v[med] == k$)
 return true;

 else return false;

}

 if ($v[med] < k$) return ricerca - 2 ($v, k, med + 1, ultimo$);

 else return ricerca - 2 ($v, k, primo, med - 1$);

$$T(n) = \begin{cases} c_1 & n=1 \\ T\left(\frac{n}{2}\right) + c_2 & n>1 \end{cases}$$

$$T(n) = O(\log(n)) \rightarrow \text{logaritmico} \quad (MT)$$

LEZIONE 21 - Alg per array / InsertionSort

- Un array è una struttura dati statica e omogenea
- d'accesso a ciascun elemento dell'array è diretto → le operazioni lettura / scrittura costo $O(1)$
- Problema della ricerca: dato array V e valore k , stabilire se $\exists i \in V = k$
 - ↳ ricerca lineare
 - ↳ ricerca dicotomica
- Problema dell'ordinamento: dato array V , determinare una permutazione dei valori contenuti nell'array V tale da ordinare gli elem. dell'array
 - ↳ sempre ordinamento crescente
 - ↳ alg. analoghi per ordinamento decrescente
- Algoritmi di ordinamento: possono essere confrontati in base a:
 - Complessità computazionale
 - Stabilità: un algoritmo si definisce tale se non altera l'ordine relativo di elem. dell'array uguali fra di loro.
 - Operano in loco: se la dimensione delle strutture dati ausiliarie di cui necessita è indipendente dal numero di elem. dell'array da ordinare

• Insertion Sort

- Al generico posso prevedere che l'array V di n elementi sia suddiviso in:
 - una sequenza $V[0], \dots, V[i-1]$ già ordinata
 - una sequenza $V[i], \dots, V[n-1]$ ancora da ordinare
- È un algoritmo stabile
- È un algoritmo che opera in loco
ma solo la var temp
 - de complessità dipende da quante volte viene eseguito il ciclo: $C + C.i$
- Implementazione:

```
void insertionSort (int v[], int n) {  
    int temp;  
    int j;  
    for (j = i-1; (j >= 0) && (v[j] > temp); j--) ]  
        v[j+1] = v[j];  
    if (j != i-1)  
        v[j+1] = temp;  
}
```

```
int main () {  
    int n = 10;  
    int v[] = {8, 4, 5, 10, 2, 1, 3, 9, 7, 6};  
    insertionSort (v, n);  
    for (int i = 0; i < n; i++)  
        printf ("%d ", v[i]);  
    printf ("\n");
```

Esempio

$3, 7, 1, 5, 8, 6 \rightarrow \begin{matrix} 1 & 3 & 7 \\ J=0 & J=1 & i=2 \end{matrix} \rightarrow \begin{matrix} 1 & 3 & 7 & 5 & 8 & 6 \\ J=1 & J=2 & i=3 \end{matrix} \rightarrow$

$1, 3, 5, 7, 8, 6 \rightarrow \begin{matrix} 1 & 3 & 5 & 7 & 8 & 6 \\ J=3 & i=4 & J=2 & J=3 & J=4 & i=5 \end{matrix} \rightarrow 1, 3, 5, 6, 7, 8$

Caso peggiore: / caso ottimo $T(n) = \Theta(n)$

for interno: $c_1 \cdot i$

corpo for esterno: $c_1 \cdot n + c_2$

for esterno:

$$\sum_{i=0}^{n-1} (c_1 \cdot i + c_2) = c_1 \cdot \frac{(n-1) \cdot n}{2} + c_2 \cdot (n-1) \rightarrow \text{costo complessivo: } \frac{c_1}{2} \cdot n^2 + (c_2 - \frac{c_1}{2}) \cdot n - c_2 + c_3$$

Costo temporale asintotico $T(n) = \Theta(n^2)$

LEZIONE 22 - Selection Sort

Selectionsort:

- Al generico posso i prevede che l'array v di n elementi sia suddiviso in due sequenze $v[0], \dots, v[i-1]$ già ordinate e $v[i], \dots, v[n-1]$ ancora da ordinare
- È un algoritmo di ordinamento stabile nel confronto tra $v[j] < \min$ e $\min < v[i]$
- È un algoritmo di ordinamento che opera in loco, ma solo le variabili auxiliarie \min e $i-\min$

Costo: non dipende dalla disposizione iniziale dei valori.

- costo for più esterno $c_1 \cdot (n-1-i) + c_2$

- costo for esterno:

$$\sum_{i=0}^{n-2} (c_1 \cdot (n-1-i) + c_2) = c_1 \cdot \sum_{j=1}^{n-1} j + c_2 \cdot (n-1)$$

costo complessivo: $T(n) = \Theta(n^2)$

Implementazione:

```
void selectionsort (int v[], int n)
```

```
    int min;
```

```
    int i_min;
```

```
    for (int i = 0; i < n-1; i++) {
```

```
        min = v[i];
```

```
        i_min = i;
```

```
        for (int j = i+1; j < n; j++)
```

```
            if (v[j] < min) {
```

```
                int min();
```

```
                int n=10;
```

```
                int v[] = {8, 4, 5, 10, 2, 1, 3, 9, 7, 6};
```

```
                selectionsort(v, n);
```

```
min = V[J];
```

```
i_min = J;
```

```
}
```

```
if(i_min != i) {
```

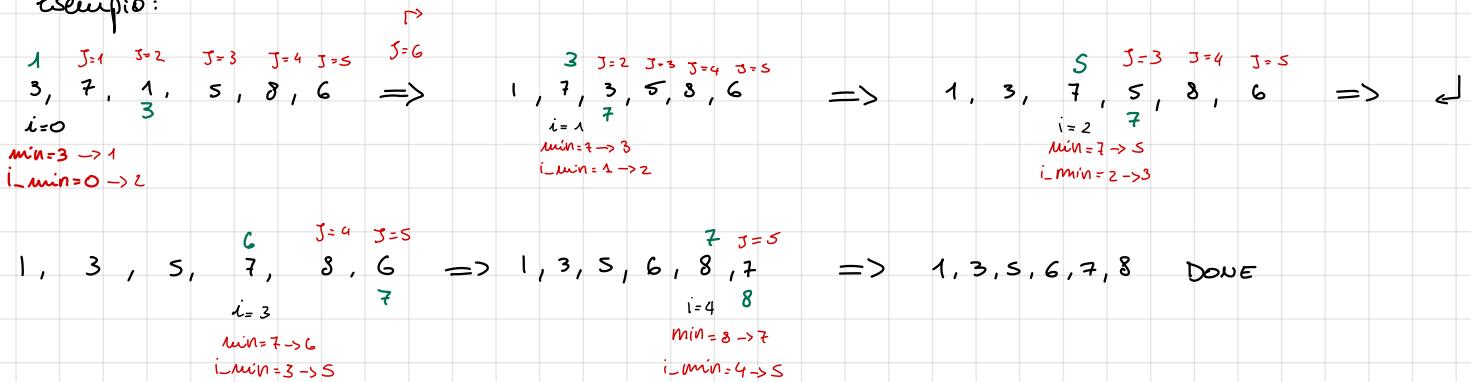
```
V[i_min] = V[i];
```

```
V[i] = min;
```

```
}
```

```
}
```

Esempio:



Bubblesort

- Al generico posso: prevede che l'array v di elementi sia suddiviso:

una sequenza $V[0], \dots, V[i-1]$ già ordinata

una sequenza $V[i], \dots, V[n-1]$ da ordinare

• È stabile → confronto fra $V[j]$ e $V[s-t]$ usa ϵ

• Opera in loco → usa la var auxiliaria temp

Implementazione:

```
Void bubble_sort (int v[], int n)
```

```
int temp;
```

```
for (int i=1; i<n; i++)  
    for (int j=n-i; j>i; j--)  
        |  
        if (V[j] < V[j-1]) {  
            temp = V[j-1];  
            V[j-1] = V[j];  
            V[j] = temp  
        }
```

```
int main()
```

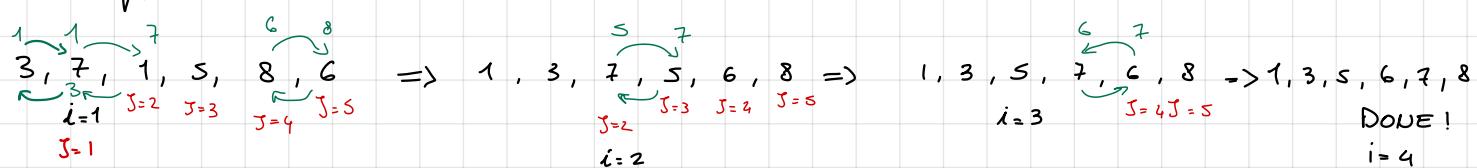
```
int n = 10
```

```
int v[] = {8, 4, 5, 10, 2, 1, 3, 9, 7, 6}
```

```
bubblesort(v, n);
```

```
for (int(i=0; i<n; i++))  
    printf ("%d", v[i]);  
    printf ("\n")  
return 0;
```

Esempio:



LEZIONE 23 - Merge-Sort

I 3 alg. di ordinamento hanno in comune l'iteramento, dove all'1-esimo passo una "metà" è ordinata e l'altra no. Moltre sono tutti e 3 in loco.
Hanno tutti costo quadratico nel caso peggiore $O(n^2)$

• Merge Sort

caratteristiche ↗ ricorsivo
 ↗ stabile → nel confronto fra $v[i]$ e $v[j]$ usiamo il minore
 ↗ non in loco → perché usa un array di appoggio

implementazione:

void mergesort (int v[], int sx, int dx)

int mx;

```
if (sx < dx) {
    mx = (sx + dx) / 2;
    merge_sort (v, sx, mx);
    merge_sort (v, mx + 1, dx);
    merge (v, sx, mx, dx)
}
```

```
int main() {
    int n;
    int v[] = { ... };
}
```

```
merge_sort (v, 0, n - 1)
for (int i = 0; i < n; i++)
    printf ("%d", v[i]);
printf ("\n");
return 0;
}
```

esempio: $mx = 0 + 5 / 2 = 2$

merge() { .. } su file

3 7 1 5 8 6

merge (v, 0, 0, 1) $b[0] \leftarrow v[0]$ $b[1] \leftarrow v[1]$

3 7 1 5 8 6

merge (v, 0, 1, 2) $b[0] \leftarrow v[2]$ $b[1] \leftarrow v[0]$ $b[2] \leftarrow v[1]$

1 3 7 5 8 6

merge (v, 3, 3, 4) $b[0] \leftarrow v[3]$ $b[1] \leftarrow v[4]$

1 3 7 5 8 6

merge (v, 3, 4, 5) $b[0] \leftarrow v[3]$ $b[1] \leftarrow v[5]$ $b[2] \leftarrow v[4]$

1 3 7 5 6 8

merge (v, 0, 2, 5) $b[0] \leftarrow v[0]$ $b[1] \leftarrow v[2]$ $b[2] \leftarrow v[3]$ $b[3] \leftarrow v[4]$ $b[4] \leftarrow v[2]$ $b[5] \leftarrow v[5]$

1 3 5 6 7 8

Costo:

- non dipende dalla disp. iniziale dei valori

- il costo della funzione merge.sort:

$$T(n) = C_1 + 2 \cdot T\left(\frac{n}{2}\right) + T_1(n)$$

\uparrow
costo merge()
 $\hookrightarrow C_2 \cdot n + C_3$

$$\Rightarrow \begin{cases} T(1) = k \\ T(n) = 2 \cdot T\left(\frac{n}{2}\right) + C_2 n + C_4 \end{cases}$$

$$n=2 \quad v=2 \quad \alpha = \log_2 2 = 1 \quad b=1$$

$$\alpha=6 \rightarrow \text{caso 3}$$

$$\Rightarrow T(n) = O(n \cdot \log(n))$$

LEZIONE 24 - Quicksort

• Quicksort:

- è ricorsivo
- non è stabile → nei confronti $v[i], v[j]$ e pivot uso $> e <$ anziché $\geq e \leq$
- opera in loco → usa var. ausiliarie temp e pivot

• Implementazione:

```
void quicksort (int v[], int sx, int dx) {  
    int pivot = v[(sx+dx)/2];  
    int i = sx;  
    int j = dx;  
    int temp;  
  
    while (i < j) {  
        while (v[i] < pivot)  
            i++;  
        while (v[j] > pivot)  
            j++;  
        if (i < j) {  
            if (i < j) {  
                temp = v[i];  
                v[i] = v[j];  
                v[j] = temp;  
            }  
            i++; j--;  
        }  
        if (sx < j)  
            quick_sort (v, sx, j);  
        if (i < dx)  
            quick_sort (v, i, dx);  
    }  
}
```

• Esempio:

```
quicksort(v, 0, 5)  
quicksort(v, 1, 5)  
quicksort(v, 1, 2)  
quicksort(v, 3, 5)  
quicksort(v, 3, 4)
```

| | | | | | | | |
|---|---|---|---|---|---|----------|----------|
| 3 | 7 | 1 | 5 | 8 | 6 | $v[0]=1$ | $v[2]=3$ |
| 1 | 7 | 3 | 5 | 8 | 6 | $v[1]=5$ | $v[3]=1$ |
| 1 | 5 | 3 | 7 | 8 | 6 | $v[1]=3$ | $v[2]=5$ |
| 1 | 3 | 5 | 7 | 8 | 6 | $v[4]=6$ | $v[5]=8$ |
| 1 | 3 | 5 | 7 | 6 | 8 | $v[3]=6$ | $v[4]=7$ |
| 1 | 3 | 5 | 6 | 7 | 8 | | |

• Costo:

- non dipende dalla disposizione iniziale dei valori
- il costo del quick-sort è

$$\begin{cases} T(1) = c_1 \\ T(n) = T(n_1) + T(n_3) + (c_1 \cdot n + c_2) \end{cases}$$

Quindi per MT $T(n) = \mathcal{O}(n \cdot \log n)$

LEZIONE 25 - Strutture dati array

L'idea è usare degli array per operare con i polinomi
implementazione:

const int D_max = 100; → così facendo possiamo
rappresentare polinomi finiti
al grado 99

```
typedef struct {  
    double coefficienti[D_max];  
    int d; → grado effettivo del polinomio  
} polinomio;
```

$O(n^2)$

$O(n \cdot m)$

i due gradi ≠ possibili

void stampa_polinomio (polinomio p)

```
for(int i=p.d; i>0; i--)  
    printf ("%lf \n", p.coefficienti[i], "x^", i, "+");  
printf ("%lf \n", p.coefficienti[0]);  
return;
```

polinomio calcola_somma (polinomio p1, polinomio p2) {

```
polinomio s;  
int dmax;  
if (p1.d > p2.d) dmax = p1.d  
else dmax = p2.d  
s.d = dmax
```

Cost:

- stampa_polinomio: $O(d)$
- calcola_somma: $O(d)$
- calcola_prodotto: $O(n \cdot m)$
- calcola_derivate: $O(d)$
- valuta_polinomio: $O(d)$

```
for (int i=0; i <= dmax; i++)  
    s.coefficienti[i] = p1.coefficienti[i] + p2.coefficienti[i];  
return s;
```

polinomio calcola_prodotto (polinomio p1, polinomio p2) {

```
polinomio p;  
p.d = p1.d + p2.d;  
for (int k=0; k <= p.d; k++)  
    p.coefficienti[k] = 0;  
if (p.d > D_max) {  
    printf ("grado troppo grande")  
    return p;
```

```
for (int i=0; i <= p1.d; i++)  
    for (int j=0; j <= p2.d; j++)  
        p.coefficienti[i+j] += p1.coefficienti[i] * p2.coefficienti[j];  
return p;
```

LEZIONE 26 - rappresentazione dati:

Tecniche:

- **Rappresentazione indiretta:** (locale)
 - dati contenuti in array
 - Pro: accesso diretto e dati mediante indice
 - Contro: dimensione fissa (allocazione \rightarrow tempo lineare)
- **Rappresentazione collegate:** (globale)
 - dati contenuti in record collegati mediante puntatori
 - Pro: dimensione variabile (aggiunta/rimozione record: tempo costante)
 - Contro: accesso sequenziale ai dati

da rappresentazione collegate è una lista

- **liste:** si dice lista una triple $L = (E, \epsilon, s)$ dove:
 - E è un insieme di elementi
 - $\epsilon \in E$ è detto testa
 - $s \subseteq E \times E$ è una relazione binaria su E che soddisfa le seguenti proprietà:
 - $\forall e \in E, (e, t) \notin s$
 - $\forall e \in E$, se $e \neq t$ allora \exists uno ed uno solo $e' \in E$ t.c. $(e', e) \in s$
 - $\forall e \in E$, esiste al più un $e' \in E$ t.c. $(e, e') \in s$
 - $\forall e \in E$, se $e \neq t$ allora e è raggiungibile da t , cioè $\exists e'_1, \dots, e'_k \in E$ (k >= 2) t.c.:
 - $e'_1 = t$
 - $(e'_i, e'_{i+1}) \in s$ per ogni $i \in \{1, \dots, k-1\}$
 - $e'_{ik} = e$

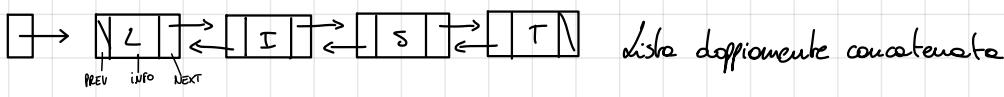
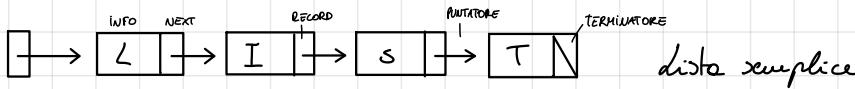
- **lista concatenata:**

In una lista **simplicemente concatenata**, ciascun nodo è costituito da:

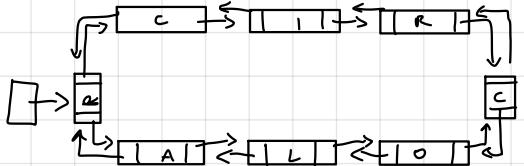
- il valore di una chiave
- un puntatore al modo successivo nella lista (o nullo)
- il primo elemento nella lista è la testa
- l'ultimo elemento nella lista è la coda

- **Liste doppicamente concatenate:**

- il valore di una chiave
- un puntatore al modo successivo nella lista (o nullo)
- un puntatore al modo precedente nella lista (o nullo)
- il primo elemento nella lista è la testa (no predecessore)
- l'ultimo elemento nella lista è la coda (no successore)



lista circolare doppemente concatenata



Def: lista ordinata:

Una lista $L = (E, t, S)$ è detta ordinata se le chiavi contenute nei suoi elementi sono disposte in modo tale da soddisfare una relazione d'ordine totale:

per ogni $e_1 \in E$, $e_2 \in E$, se $(e_1, e_2) \in S$ allora la chiave di e_1 precede la chiave di e_2 nella relazione d'ordine totale.

Problemi per liste

- visita
- ricerca
- inserimento
- rimozione

OSS: array e lista sono strutture dati lineari.

LEZIONE 27 - Algoritmi per liste

Array → accesso diretto agli indici dell'elemento → costo costante / dim fissate

Lista → accesso sequenziale agli elementi → costo lineare / dim variabili

→ elementi adiacenti memorizzati in posti diverse della mem.

→ per implementarne vennero usati i puntatori (collegamenti)

• Implementazione liste in C. (soravano ricorsive)

- Le def di un tipo struct deve essere nata per potere includere membri di quel tipo in altre struct

struct s {

X errore

struct s next; // tipo sconosciuto

}

struct s {

... ✓ funziona

struct s *next

} punta a struct s

⇒ strutture di questo tipo sono adatte per costruire strutture dati dinamiche

NB1: I puntatori hanno le stesse dim in memoria

NB2: Per dichiarare una variabile var di tipo s → struct s var;

Altamente: → typedef struct _s {

...

struct _s *next;

} s;

↳ s var; per dichiararla

```

struct elemento {
    int valore;
    struct elemento *successivo;
}

```

```

struct elemento_dc {
    int valore;
    struct elemento_dc *successivo, *precedente;
}

```

```

struct elemento *crea_lista () {
    struct elemento *p, *punt;
    int i, n;
    printf ("In Specificare il # di elens... ");
    scanf ("%d", &n);
    if (n==0) p=NULL;
    else {
        p = malloc (sizeof(struct(elemento)));
        printf "\n ins val ... (%d", &p->valore);
        scanf ("%d", &p->valore);
        punt = p;
        for int(i=2; i<=n, i++)
            [...]
    }
}

```

Algoritmi per le liste

- Algoritmo di visita:

```

void visita (elemento *testa)
{
    elemento *e;
    for (e=testa; e!=NULL; e=e->next)
        elabora (e->valore)
}

```

Costo: Se la lista ha n elementi e l'elaborazione del valore di ogni elemento si svolge in al più d passi, la complessità è:

$$T(n) = c_1 + n \cdot (c_2 + c_3) = O(n)$$

- Algoritmo di ricerca:

```

bool cerca (struct elemento *testa, int valore) {
    struct elemento *e;
    e = testa;
    while (e!=NULL)
        if (valore == e->valore) return true;
        else e = e->successivo;
    } return false
}

```

Costo: Se la lista ha n elementi:

nel caso peggiore (valore non presente nella lista)

$$T(n) = c_1 + n \cdot c_2 + c_3 = c_2 \cdot n + c_4 = O(n)$$

nel caso ottimo (valore = 1° elem della lista)

$$T(n) = O(1)$$

- Algoritmo di inserimento:

- varie a seconda delle posizioni in cui l'operazione è effettuata
- gli inserimenti alle estremità sono trattati in seguito
- 1 se inserisce, 0 se no

(consideriamo ordinato)

Costo: Caso peggiore $T(n) = c_1 + n \cdot c_2 + c_3 = c_2 \cdot n + c_4 = O(n)$

Caso ottimo $T(n) = c_1 = O(1)$

- Algoritmo di rimozione

→ viene a seconda delle posizioni in cui l'operazione è effettuata

- gli inserimenti alle estremità sono trattati in seguito

- 1 se inserisce, 0 se no

(consideriamo ordinato)

Costo: Caso peggiore $T(n) = c_1 + n \cdot c_2 + c_3 = c_2 \cdot n + c_4 = O(n)$

Caso ottimo $T(n) = c_1 = O(1)$

LEZIONE 28 - Code e pile

• Code: è una lista governata dal principio FIFO.

- I nodi vengono tutti inseriti nella stessa estremità.

- I nodi vengono tutti rimossi nella parte opposta.

- Per estrarre inserimento e rimozione → si individua l'indirizzo del primo elemento (uscita) e quello dell'ultimo (entrata), sono indefiniti se la coda è vuota.

• Implementazione

Vedi file

struct → crea lista() → stampa lista() [...]

• Costo: coda con n elem

- inserimento: $T(n) = O(1)$

- rimozione: $T(n) = O(1)$

• Pile: lista governata dal principio LIFO

- inserimenti e rimozioni avvengono alla stessa estremità

- è individuata tramite l'indirizzo del suo primo elemento (cima), indefinita solo se la pile è vuota

Costo: pile con n elementi $\begin{cases} \text{inserimento: } T(n) : O(1) \\ \text{rimozione: } T(n) : O(1) \end{cases}$

LEZIONE 29 - ALBERI

- **Definizione di Albero:** Una triple $T = (N, r, B)$ dove
 - N è un insieme di nodi
 - $r \in N$ è detta radice
 - $B \subseteq N \times N$ è una relazione binaria su N che soddisfa le props:
 - $\forall n \in N \ (n, n) \notin B$
 - $\forall n \in N$, se $n \neq r$ allora $\exists! n' \in N$ t.c. $(n', n) \in B$
 - $\forall n \in N$, se $n \neq r$ allora n è raggiungibile da r , cioè esistono $n'_1, \dots, n'_{k-1} \in N$ ($k \geq 2$) t.c.
 - $n'_1 = r$
 - $(n'_i, n'_{i+1}) \in B \quad \forall i \in \{1, \dots, k-1\}$
 - $n'_{k-1} = n$

- **Definizione alternativa:** triple $T = (N, r, B)$ dove:

- N è un insieme di nodi
- $r \in N$ è la radice
- $B \subseteq N$ è una relazione binaria su N che soddisfa le props:
 - $\forall n \in N \ (n, n) \notin B$
 - $\forall n \in N$, se $n \neq r$ allora esistono e sono unici n'_1, \dots, n'_{k-1} ($k \geq 2$) t.c.
 - $n'_1 = r$
 - $(n'_i, n'_{i+1}) \in B$ per ogni $i \in \{1, \dots, k-1\}$
 - $n'_{k-1} = n$

Oss: Una lista è un caso particolare di albero con un unico percorso, senza diramazioni.
Gli alberi sono strutture dati non lineari.

Definizioni: Sia $T = (N, r, B)$

- Se $(n, n') \in B$, allora n è il **padre** di n'
- Se $(n, n') \in B$, allora n' è il **figlio** di n
- Se $(n, n_1) \in B, (n, n_2) \in B$, allora n_1 e n_2 sono detti **fratelli**
- I nodi primi di figli sono detti **foglie**
- I nodi che non sono foglie sono detti **nodi interni**
- Gli elementi di B sono detti **rami**

- **Grado e livello:** Sia $T = (N, r, B)$ un albero:

- Si dice **grado** di T il massimo numero di figli di un nodo di T
 $o(T) = \max_{n \in N} |\{n' \in N \mid (n, n') \in B\}|$
- Il nodo radice r è al **livello 0**
- Se $n \in N$ è al livello i e $(n, n') \in B$, allora n' è al livello $i+1$

- Altezza (profondità) e ampiezza: Si dà $T(N, r, B)$ un albero.
 - si dice **profondità** di T il massimo numero di nodi che si attraversano nel percorso di T che va dalla radice alle foglie più distante
 - Si dice **ampiezza** di T il massimo numero di nodi di T che si trovano allo stesso livello $\rightarrow b(T) = \max_{1 \leq i \leq h(T)} |\{n \in N \mid n \text{ è a livello } i\}|$

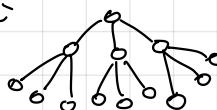
Albero pieno:

Un albero si dice **d-ario** se ogni nodo ha al massimo d figli \rightarrow cioè di grado T
Un albero d-ario si dice **pieno** se ogni nodo interno ha esattamente d figli



Albero completo: Un albero d-ario si dice **completo** se è pieno e ogni foglia ha la stessa profondità

Albero ternario completo

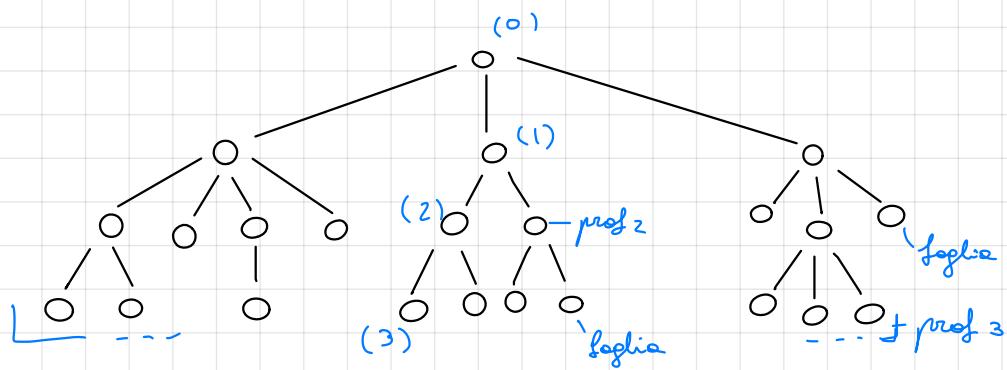


Osservazioni:

- Un albero di grado $d > 1$ può avere al massimo di nodi al livello i
 - Un albero di grado $d > 1$ e altezza h può contenere al massimo un numero di nodi pari a
- $$n(d, h) = \sum_{i=0}^h d^i = \frac{d^{h+1} - 1}{d - 1}$$
- Un albero di grado $d > 1$ e altezza h può contenere al massimo un numero di nodi pari a
- $$\tilde{n}(d, h) = \sum_{i=0}^{h-1} \frac{d^{h+i} - 1}{d - 1}$$

LEZIONE 30 - ALBERI

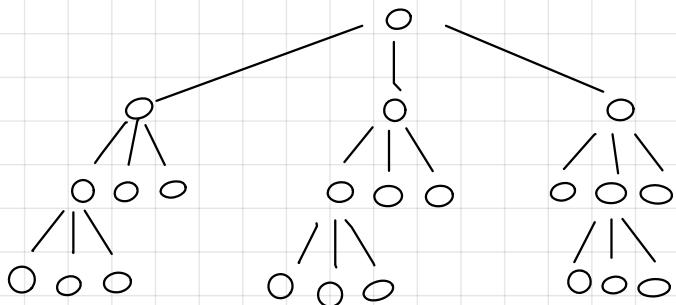
es 1



$\exists!$ nodo 4 figli, e nessuno ne ha > di 4
 L> grado dell'albero = 4

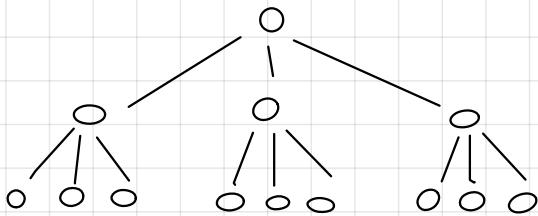
$$\begin{aligned} \text{prof max} &= 3 \\ \text{ampiezza} &= 10 \end{aligned}$$

es 2



E' un albero ternario pieno
 l'ampiezza è $\rightarrow 9$ e
 le profondità è $\rightarrow 3$ h
 grado = 3
 TOT = 22 nodi

es 3



E' un albero ternario pieno - completo
 profondità = 2 h
 ampiezza 9
 grado 3 3
 TOT = 13 nodi = n max n nodi at. p. 2

o **Sottoalbero**: definizione - Siano $T(N, r, B)$ e $n \in N$

Si dice **sottoalbero** generato da n l'albero $T'(N', n, B')$ dove:

- N' è il sottoinsieme di nodi di N raggiungibili da n
- $B' = B \cap (N' \times N')$

Props: Sia $T(N, r, B)$ e $T_1 = (N_1, r_1, B_1)$ e $T_2 = (N_2, r_2, B_2)$ sottoalberi generati da $n_1 \in N$ e $n_2 \in N$

allora: - $N_1 \cap N_2 = \emptyset$ oppure

- $N_1 \subseteq N_2$ oppure
- $N_2 \subseteq N_1$

o Albero binario: un albero $T(N, n, B)$ è detto binario se

- non contiene nodi
- è composto da tre insiemini disgiunti di nodi:

- un nodo radice
- un albero binario T_S detto sottoalbero sinistro della radice
- un albero binario T_D detto sottoalbero destro della radice

- Def alternativa: se:

$$B = B_S \cup B_D$$

$$B_S \cap B_D = \emptyset$$

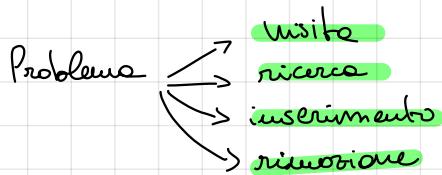
$$\forall n \in N, n_1 \in N, n_2 \in N, \text{ se } (n, n_1) \in B_S \text{ (risp } B_D) \text{ e } (n, n_2) \in B_S \text{ (risp } B_D) \text{ allora } n_1 = n_2$$

Se $(n, n') \in B_S$ (risp B_D) allora n' è figlio sinistro (risp. destro) di n

- Il numero di nodi di un albero binario completo di altezza h è $n(h) = \sum_{i=0}^h 2^i = 2^{h+1} - 1$

- Il numero di nodi interni $n(h) = \sum_{i=0}^{h-1} 2^i = 2^h - 1$

o Algoritmi per gli alberi:



o Implementazione Albero binario:

```
typedef struct nodo {  
    int valore;  
    struct nodo *sx, *dx;  
}
```



```
struct nodo {  
    int valore;  
    struct nodo *sx, *dx;
```

Note: I nodi di un albero binario non sono necessariamente memorizzati in modo consecutivo.

L'accesso a ogni nodo necessita dell'indirizzo della radice; indefinito se l'albero è vuoto.

L'accesso al nodo in questione avviene scorrendo tutti i nodi che precedono il nodo lungo l'unico percorso che comprende il primo nodo dell'albero al nodo dato.

o Algoritmi per Alberi - Visita in ordine Anticipato

```
void visita_anticipato (struct nodo *n) {
    if (n != NULL) {
        elabora (n->valore);
        visita_anticipato (n->sx);
        visita_anticipato (n->dx);
    }
}
```

- Prima si elabora il valore contenuto nel nodo al quale si è punti
- Poi si visitano con lo stesso algoritmo il sottoalbero destro e sinistro

o Visita in ordine simmetrico

```
void visita_simmetrica (struct nodo *n)
{
    if (n != NULL) {
        visita_simmetrica (n->sx);
        elabora (n->valore);
        visita_simmetrica (n->dx);
    }
}
```

- Prima si visita con lo stesso algoritmo il sottoalbero sinistro del nodo al quale si è punti
- Poi si elabora il contenuto del nodo
- Infine si visita con lo stesso algoritmo il sottoalbero destro.

o Visita in ordine posticipato

```
void visita_posticipato (struct nodo *n) {
    if (n != NULL) {
        visita_posticipato (n->sx);
        visita_posticipato (n->dx);
    }
}
```

- Prima si visitano con lo stesso algoritmo il sottoalbero sinistro e poi quello destro del nodo
- Poi si elabora il valore del nodo al quale si è punti

o Costo della visita:

Sia n il numero di nodi dell'albero binario e d il numero di possibili eseguiti durante l'elaborazione del valore del nodo.

k # nodi sottoalbero sx

$n-1-k$ # nodi sottoalbero dx

$$\begin{cases} T(0) = C_0 \\ T(n) = T(k) + T(n-1-k) + (d+c) \end{cases} \quad n \geq 1$$



dimostra che $T(n) = n \cdot (d + c + C_0) + C_0$

Se $n=1$

$$T(1) = T(0) + T(0) + (d+c) = (d+c+C_0) + C_0$$

Suppongo che valga $\forall j \in \{1, \dots, n-1\}$

tesi vale per $n=1$

$$\begin{aligned} \text{Allora } T(n) &= k(d+c+C_0) + C_0 + (n-1-k) \cdot (d+c+C_0) + C_0 + (d+c) = \\ &= (n-1) \cdot (d+c+C_0) + C_0 + (d+c+C_0) = n(d+c+C_0) + C_0 \end{aligned}$$

Albero binario n nodi =
 caso peggiore $T(n) = O(n)$
 caso ottimo $T(n) = O(1)$

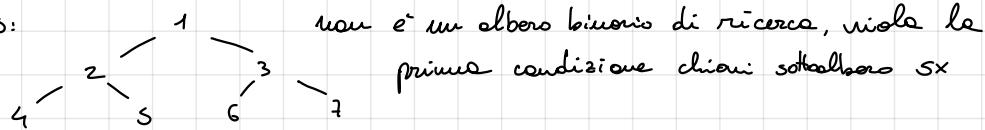
Oss: I tre alg di visita possono essere usati per la ricerca se modificati opportunamente

LEZIONE 31

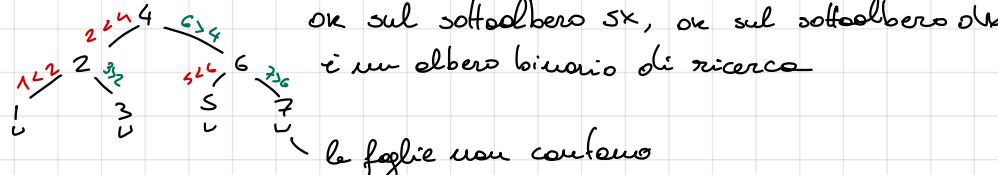
Albero binario di ricerca:

- è un albero binario f.c. ogni nodo, se il nodo contiene una chiave di valore k
 - ogni nodo sottoalbero sx contiene una chiave di val minore o uguale a k
 - ogni nodo sottoalbero dx contiene una chiave di val maggiore o uguale a k

es:



es:



Algoritmi $\xrightarrow{\text{ricerca}}$ inserimento $\xrightarrow{\text{dove rimuovere un albero binario di ricerca}}$
 $\xrightarrow{\text{rimozione}}$

Costo: nel caso peggiore: $T(n) = O(n)$ \rightarrow tutto il percorso, dipende da h

valore cercato nella radice
caso ottimale $\xrightarrow{\text{valore da inserire nella radice}}$
 $\xrightarrow{\text{rimozione in un albero vuoto}}$

nel caso medio: $T(n) = O(\log n)$

$T(n) = O(1)$

OSS: Il costo delle varie operazioni, nel caso peggiore dipende dalla struttura dell'albero

OSS: Per avere $O(\log n)$ nel caso peggiore, bisogna cercare di le foglie tutte allo stesso livello
(solo così l'altezza si cresce come il log del numero di nodi)

↓ continua

LEZIONE 32 -

OSS:

- Inserimento \rightarrow nuovi nodi nei liv. già esistenti per riempirli prima di creare altri
- rimozione \rightarrow ridistribuire i nodi rimasti nei livelli in modo uniforme
l'albero binario di ricerca deve rimanere tale dopo le due operazioni

Bilanciamento in altezza: è bilanciato in sé, per ogni nodo dell'albero, l'h del sottoalbero sx e dx differiscono max di 1.

\rightarrow Gli alberi bilanciati in altezza sono anche detti AVL

Fattore di bilanciamento $B(n)$ di un nodo n è: la differenza tra la h del sott.al. sx - dx

$$B(n) = h(sx(n)) - h(dx(n))$$

OSS:

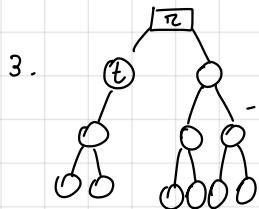
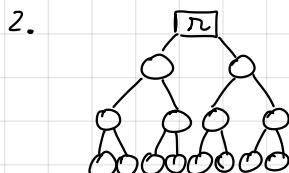
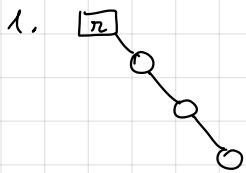
Un albero binario di ricerca è AVL se $\forall n \in N, |B(n)| \leq 1$

OSS:

In un albero binario di ricerca, per implementare un algoritmo di ricerca non è necessario visitare tutti i nodi sistematicamente

- Alberi bilanciati: esempi

Il fattore di bilanciamento è zero quando l'albero degenera in una lista (1)
 è 0 quando l'albero è completo (2)

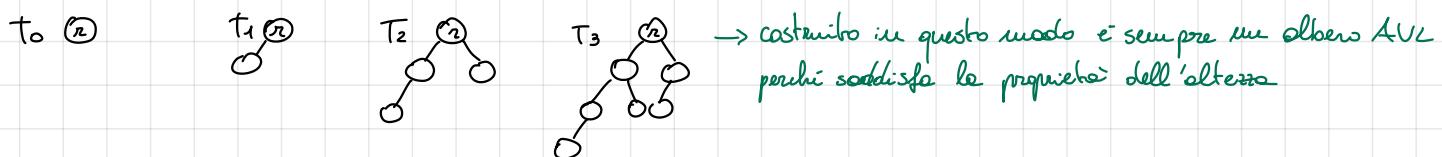


OSS: Per volerare l'h di un albero AVL, consideriamo gli alberi più sbilanciati che si possono costruire e ponete di numero di nodi

• Alberi di Fibonacci

- Un AdF con h=0 è un albero con solo la radice
- Un AdF con h=1 è un albero con radice + un figlio (sx)
- Un AdF con h>1 può essere costruito ponendo, tranne l'aggiunta di un nodo radice, un AdF di h-1 e un AdF di h-2

Esempi AdF



Oss: tra tutti gli alberi di altezza h, quello di Fibonacci è quello con meno nodi possibili e anche il più sbilanciato completo → più bilanciato

Props: Sia T_h un albero di F di altezza h e sia n_h il num. di nodi.

$$\text{Allora } h = \Theta(\log(n_h))$$

Dim: Per costruzione $n_h = 1 + n_{h-1} + n_{h-2}$

- induzione $n_h = F_{h+3}-1$ caso base: se $h=0$ $n_0=1 = F_3-1$

- ip. ind: supponiamo che la tesi valga $\forall k \in \{0, \dots, h\}$

- dim che tesi vale $\times h+1$

$$\Rightarrow \text{dato che } F_h = \frac{1}{\sqrt{5}} (\phi^h - \bar{\phi}^h) = \Theta(\phi^h) \text{ e quindi } n_h = \Theta(\phi^{h+3}) = \Theta(\phi^h) \text{ da cui } h = \Theta(\log(n_h))$$

Corollario: un albero AVL con n nodi ha h $\Theta(\log(n))$

Dim: Sia h e sia n # nodi; un AVL con h h ha n_h nodi. con $n_h < n$

→ ricordo che $h = \Theta(\log(n_h))$

→ si ha che $h = \Theta(\log(n))$

OSS: costo caso peggiore algoritmi per AVL $\rightarrow \Theta(\log(n))$

LEZIONE 33

→ implementazione per albero AVL.

1. struct
2. crearnodo } da prendere su file alberiAVL.c
3. creeralbero

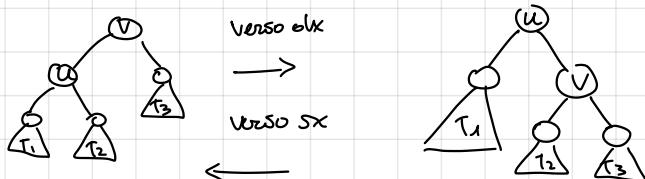
Alberi bilanciati: Rotazioni:

Nella rotazione di base, un nodo pernò viene fatto ruotare verso destra o sinistra - tempo $O(1)$

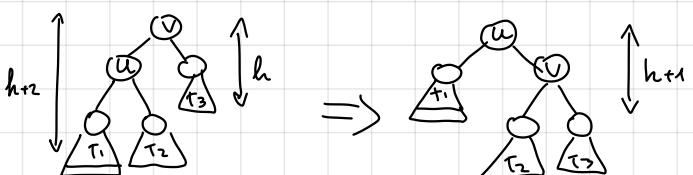
Bilanciamento tramite rotazioni:

- le rotazioni sono effettuate su nodi sbilanciati
- sia n un nodo con fattore di bilanciamento
- \exists almeno un sotto-albero S di n che sbilancia l'albero
- 4 scorrimenti delle posizioni di S si hanno 4 casi:
 - SX-SX ; SX-DX ; DX-SX ; DX-DX

Rotazione semplice:

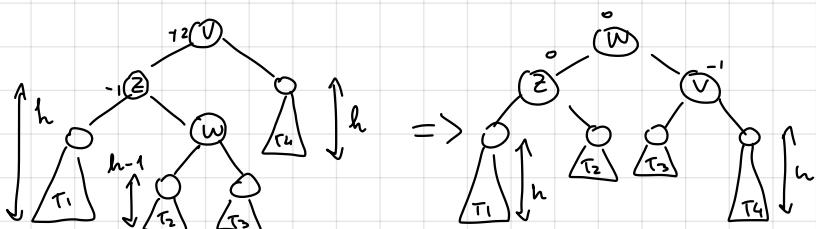


• Rotazione SS: si applica una rot. semplice verso dx al nodo d'altezza dell'albero coinvolto posso da $h+2$ a $h+1$



Rotazione SD:

- si applicano due rotazioni semplici:
rot. s. di z verso sx
rot. s. di v. verso dx



LEZIONE 34

• Albero rosso-nero: è un albero binario di ricerca rosso-nero se:

- ogni nodo è o rosso o nero
- la radice è nera
- se un nodo è rosso \rightarrow entrambi i figli sono neri
- ogni foglia è nera. Non ha contenuto informativo
- Il nodo, tutti i percorsi dal nodo a una foglia del suo sottoalbero attraversano lo stesso numero di nodi neri

• Altezza nera: hn di un nodo n di un a.b.r.n.n è il numero di nodi neri che si attraversano in qualche cammino da n a una foglia nel suo sottoalbero (senza n stesso)

• Proposizione: Ogni nodo n in un albero RN contiene nel suo sottoalbero un numero di nodi interni maggiore o uguale a $2^{hn(n)-1}$

• Proposizione: L'altezza hn di un a.r.b.r.m. con n nodi soddisfa $hn = \Theta(\log(n))$

• Implementazione:

```
struct nodo_rn {  
    int valore;  
    char c;  
    struct nodo *sx, *dc, *pdre;  
};
```

↓

Gli alg di ricerca/ins/rim
hanno complessità $\Theta(\log(n))$
caso peggiore

- Oss: - Gli alberi rosso-neri possono essere rappresentati aggiungendo un nodo sentinella, come figlio di tutte le foglie.
- Un a.b.r.r.n è individuato attraverso l'indirizzo delle sentinelle.
- Visto che la sentinella ha più padri, l'indirizzo del padre della sentinella è NULL
- L'indirizzo del padre della radice è l'indirizzo della sentinella

• Alberi 2-3: albero in cui ogni nodo interno ha 2 o 3 figli e tutti i cammini radice-foglia hanno la stessa lunghezza

• Prop: Sia T un albero 2-3 con n nodi, f foglie e altezza hn. Valgono le seguenti diseguaglianze: $2^{h+1}-1 \leq n \leq (3^{h+1}-1)$ e $2^h \leq f \leq 3^h$
↓
- può avere grado 2 o 3

• Dimo: $hn=0$ allora albero \rightarrow 1 nodo che è foglia. $n=f=1$
L'tesi vale $\forall k \in \{0, \dots, hn\}$; p.ind.
L'dim che vale $\times hn+1$

• Corddorario: $hn = \Theta(\log(n)) \Rightarrow$ poiché $2^{h+2}-1 \leq n \leq (3^{h+2}-1)/2$ si ha che $2^{h+2} \leq n+1$
 $2^{h+2} \leq 3^{h+2}$
e quindi $\log_3(2n+1) - 2 \leq hn \leq \log_3(n+1) - 2$

- Ogni nodo di un albero 2-3 contiene 2 info suppl:

$S[n]$ = massima chiave nel sottoalbero radicato nel figlio di n

$C[n] = \begin{cases} = & \text{figlio di n è nero} \\ \neq & \text{figlio di n è rosso} \end{cases}$

LEZIONE 35

Grafo: si dice grafo diretto (orientato) una coppia $G = (V, E)$ dove

- V è un insieme di vertici
- $E \subseteq V \times V$ è una relazione binaria su V

OSS: Un albero è un caso particolare di grafo orientato in cui esiste un unico vertice dal quale giungono degli altri vertici e raggiungibile attraverso un unico percorso.

Grafo completo: un grafo orientato $G = (V, E)$ si dice completo se $E = V \times V$

• **Sottografo:** Se $G = (V, E)$ un grafo orientato. Allora $G' = (V', E')$ si dice sottografo di G se:

- $V' \subseteq V$
- $E' \subseteq E \cap (V' \times V')$

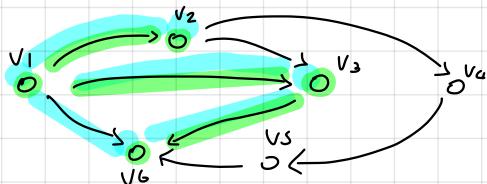
Se $E' = E \cap (V' \times V')$ allora G' si dice sottografo indotto.

• **Grafo trasposto:** Se $G = (V, E)$ un grafo orientato. Allora $G' = (V', E')$ si dice grafo trasposto di G se:

- $V' = V$
- $E' = \{(v', v) \mid (v, v') \in E\}$

Vertici adiacenti: Se $(v, v') \in E$ allora v' è adiacente a v

Esempio di grafo



$$\begin{aligned} E &= \{(v_1, v_2), (v_1, v_3), (v_1, v_6), (v_2, v_3), (v_2, v_4), (v_3, v_5), (v_4, v_5), (v_5, v_6)\} \\ V &= \{v_1, v_2, v_3, v_4, v_5, v_6\} \\ G &= (V, E) \end{aligned}$$

- Se scegliessi questi vertici e questi archi, otterrei un sottografo
- Sottografo indotto

• **Grado:** Se $G = (V, E)$ un grafo orientato e sia $v \in V$

- Si dice **grado uscente** di $v \in V$ il numero di vertici adiacenti a v : $d_o(v) = |\{v' \in V \mid (v, v') \in E\}|$
- Si dice **grado entrante** di $v \in V$ il numero di vertici a cui v è adiacente: $d_i(v) = |\{v' \in V \mid (v', v) \in E\}|$
- Si dice **grado di** $v \in V$ il numero di archi in cui v è coinvolto: $d(v) = d_o(v) + d_i(v)$
- Si dice **grado di** G il massimo grado di un vertice G : $d(G) = \max_{v \in V} \{d(v)\}$

• **Definizioni**

- Se $d_o(v) = 0$ e $d_i(v) > 0$ allora v è un **vertice terminale**
- Se $d_i(v) = 0$ e $d_o(v) > 0$ allora v è un **vertice iniziale**
- Se $d(v) = 0$ allora v è un **vertice isolato**

- $w \in V$ è raggiungibile da $v \in V$ se esiste un percorso da v a w
- percorsi

- un percorso è semplice se tutti i vertici che lo compongono sono distinti, eccetto il più primo e ultimo.
- un percorso è un ciclo se il suo primo vertice coincide con il suo ultimo vertice. Si dice grafo ciclico.
- G è connesso se $\forall v_1 \in V, v_2 \in V$ esiste un percorso da v_1 a v_2 o da v_2 a v_1 .
- G è fortemente connesso se per ogni $v_1 \in V, v_2 \in V$ esistono un percorso da v_1 a v_2 e uno da v_2 a v_1 .

• Grafo pesato: è una triple $G = (V, E, w)$ dove $G = (V, E)$ è un grafo $w: E \rightarrow \mathbb{R}$ è la f. della peso

Il peso associato ad un arco rappresenta di solito un tempo, una distanza, una capacità o un guadagno possibile

LEZIONE 3G

• Implementazione: Grafo

- può essere rappresentato come una struttura dati dinamica reticolare detta lista di adiacenze e formata da
 - una lista primaria dei vertici
 - più liste secondarie degli archi
- La lista primaria contiene un elemento per ciascun vertice del grafo, il quale contiene a sua volta le teste delle relative liste secondarie.
- Le liste secondarie associate ad un vertice descrive tutti gli archi incidenti su quel vertice, in quanto contiene gli indirizzi di tutti i vertici adiacenti al vertice in questione.

```
typedef struct _vertice {
    int valore;
    struct _vertice *vertice_succ;
    struct arco *lista_archi;
} vertice;
```

```
typedef struct _arco {
    double peso;
    struct _vertice *vertice_adiacente;
    struct _arco *arco_succ;
} arco;
```

• Impl. - liste di adiacenze:

- usando la rappresentazione mediante liste di adiacenze, i vertici e gli archi di un grafo non sono necessariamente memorizzati consecutivamente.
- l'accesso ad ognuno di essi necessita dell'indirizzo del primo elemento delle liste primarie.
 - indefinito se il grafo è vuoto.
- l'accesso ad un vertice avviene scorrendo tutti i vertici che precedono il vertice in questione nelle liste primarie.
- l'accesso ad un arco avviene individuando 1) l'elem. delle liste primarie associato al vertice di partenza e 2) l'elemento della corrispondente lista secondaria associata al vertice di arrivo.

Ocupa spazio $O(|V| + |E|)$ de grafo $G = (V, E)$

• Comune nei grafi sparsi
($|E| \ll |V|^2$)

• Verificare l'arco
 $O(|V| + |E|)$ caso pessimo

o Matrice di edicenza: Un grafo $G = (V, E)$ può essere rappresentato in tal modo:

- la matrice ha tante righe e colonne quanti sono i vertici del grafo

- l'elemento sulla i -esima riga e j -esima colonna vale: 1 se \exists arco tra vertice i e j
0 altrimenti

- Se $G = (V, E)$ è un grafo pesato, l'elem sulla i -esima riga e j -esima colonna vale:

↳ - w_{ij} se w_{ij} è il peso associato all'arco tra i vertici i e j

↳ - 0 se non \exists alcun arco tra i vertici i e j .

⇒ consente la verifica dell' \exists di un arco tra due vertici dati in tempo cost $O(1)$

⇒ occupazione memoria $O(|V|^2) \rightarrow$ non conviene nei grafi sparsi.

Algoritmo di Dijkstra:

Problema → calcolare il cammino minimo su un grafo orientato

Input → $G = (V, A)$ orientato, $l_{ij} \geq 0 \quad \forall (i, j) \in A$

→ s e t , vertice iniziale e finale.

Output → Cammino minimo e la sua lunghezza.

Notazione:

$\lambda(i)$ = lung. corrente del cammino minimo da s a i .

$p(i)$ = predecessore di i lungo il cammino minimo corrente da s a i .

T = insieme dei nodi temporanei, per i quali $\lambda(i) > u(i)$ (inizialmente $V = T$)

P = insieme dei nodi permanenti, per i quali abbiamo già cammino minimo (alla fine $V = P$)

Algoritmo:

ultimo nodo diventato permanente.

Passo 0 - init: consideriamo $s=1$. Si pone $\lambda(1)=0$, $\lambda(i)=\infty$, $i \neq 1$ - $p(i)=0 \quad \forall i \in V - \{s\}$

Passo 1 - Aggiornamento etichette nodi in T : \forall successore j di s t.c. $j \in T$, se $\lambda(j) > \lambda(s) + l_{sj}$
allora $\lambda(j) \leftarrow \lambda(s) + l_{sj}$ e $p(j) \leftarrow s$

Passo 2 - Scelta del nodo permanente succ.:

Sia $h \in T$ t.c. $\lambda(h) = \min_{j \in T} \lambda(j)$ si pone $-P = P \cup \{h\}$
 $-T = T \setminus \{h\}$

Passo 3 - se $t \neq h$, passo 1!

altrimenti: $\lambda(t) = u(t)$; $c_0 = t$, $c_1 = p(c_0)$, $c_2 = p(c_1), \dots, c_q = s = p(c_{q-1})$

#END