

## Numeri di Fibonacci:

$$F_n = \begin{cases} 1 & \text{se } n=1,2 \\ F_{n-1} + F_{n-2} & \text{se } n \geq 3 \end{cases}$$

Quindi: 1, 1, 2, 3, 5, 8, 13...

SUPPONIAMO DI SCRIVERE UNA PROCEDURA CHE PREnda UN INTERO  $n$  E RESTITUISCA IL PROSSIMO NUMERO DI FIBONACCI

1) Binet scoprì che l' $n$ -esimo numero di Fibonacci può essere scritto come:

$$F_n = \frac{1}{\sqrt{5}} \left( \varphi^n - \hat{\varphi}^n \right)$$

2) Sappiamo che  $ax^2 + bx + c = 0 \rightarrow x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

3) Se ho  $x^2 - x - 1 = 0 \rightarrow x^2 = x + 1$  (come per Fibonacci!)

$$\begin{aligned} a &= 1 \\ b &= -1 \\ c &= -1 \end{aligned} \quad x_{1,2} = \frac{1 \pm \sqrt{5}}{2}$$

Quindi  $\varphi = x_1 = \frac{1 + \sqrt{5}}{2}$

N.B.  $\varphi$  è un irrazionale:  
la somma di due razionali  
di un intero.

$$\hat{\varphi} = x_2 = \frac{1 - \sqrt{5}}{2}$$

4) Fib<sub>1</sub>(int n)

$$\text{return } \frac{1}{\sqrt{5}} \left( \varphi^n - \hat{\varphi}^n \right); \quad \rightsquigarrow \text{PRIMO ALGORITMO PER CALCOLARE IL N° DI FIBONACCI}$$

Domande da porsi:

- (1) CONVERGE? (cioè si ferma prima o poi?)
- (2) RISOLVE IL PROBLEMA? con quali condizioni?
- (3) GRADO DI COMPLESSITÀ? (quante istruzioni vengono eseguite?)

le nostro algoritmo converge ed esegue un'istruzione, quindi ha grado di complessità basso.

DIMOSTRO CHE FUNZIONA TRAMITE INDUZIONE

Base  $\rightarrow n=1,2$   
Passo ind.  $n \geq 3$

Assumo che valga fino a  $F_{n-1}$  ] INDUZIONE  
e lo dimostro per  $n$

Per  $n=1$

$$F_1 = \frac{1}{\sqrt{5}} (\varphi - \hat{\varphi}) = \frac{1}{\sqrt{5}} \left( \frac{1 + \sqrt{5}}{2} - \frac{1 - \sqrt{5}}{2} \right) = \frac{1}{\sqrt{5}} \left( \frac{2\sqrt{5}}{2} \right) = 1 \quad \text{OK}$$

Per  $n=2$

(controllato anche per 2 perché  $\varphi$  è di 2 sole)

$$F_2 = \frac{1}{\sqrt{5}} \left( \frac{(1+\sqrt{5})^2}{4} - \frac{(1-\sqrt{5})^2}{4} \right) = \frac{1}{\sqrt{5}} \left( \frac{(1+5+2\sqrt{5}) - (1+5-2\sqrt{5})}{4} \right) = \frac{1}{\sqrt{5}} \cdot \sqrt{5} = 1$$

Passo induutivo,  $n=3$

PER DEFINIZIONE,  $F_n = F_{n-1} + F_{n-2}$

**Suppongo vero fino a  $n-1$ , se che  $n \rightarrow$  è calcolabile e quindi posso procedere.]**

**IPOTESI INDUTTIVA:** Vale per  $F_{n-1}$  e per  $F_{n-2}$ , quindi posso sostituire

$$\begin{aligned} F_n &= \frac{1}{\sqrt{5}} \left( \varphi^{n-1} - \hat{\varphi}^{n-1} \right) + \frac{1}{\sqrt{5}} \left( \varphi^{n-2} - \hat{\varphi}^{n-2} \right) \\ &= \frac{1}{\sqrt{5}} \left( \varphi^{n-1} - \hat{\varphi}^{n-1} + \varphi^{n-2} - \hat{\varphi}^{n-2} \right) \\ &= \frac{1}{\sqrt{5}} \left( (\varphi^{n-1} + \varphi^{n-2}) - (\hat{\varphi}^{n-1} + \hat{\varphi}^{n-2}) \right) \\ &= \frac{1}{\sqrt{5}} \left[ \varphi^{n-2} \varphi^2 - \hat{\varphi}^{n-2} \hat{\varphi}^2 \right] = \frac{1}{\sqrt{5}} (\varphi^n - \hat{\varphi}^n) \end{aligned}$$

SICURAMENTE CORRETTA

$$F_n = \begin{cases} 1 & n=1,2 \\ F_{n-1} + F_{n-2} & n \geq 3 \end{cases}$$

19.09.2013

PSEUDOCODICE

```

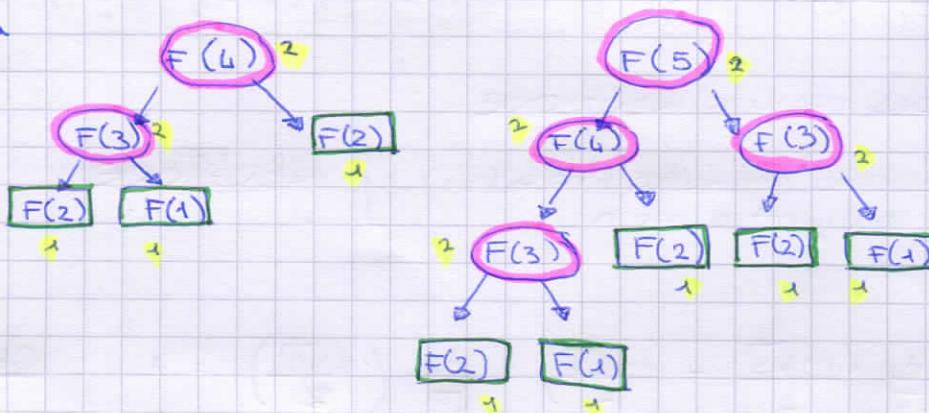
Fib2(int n) → int
  if (n ≤ 2) then return 1
  then return Fib2(n-1) + Fib2(n-2)
  
```

COSA FA NEL CASO...?

(n = 1)	1
(n = 2)	1
(n = 3)	6
(n = 4)	7
(n = 5)	14

→ D NUMERO DI ISTRUZIONI ESEGUITE  
DALL'ALGORITMO

- nodi foglia
- nodi interni
- n° istr.



## TEMPO IMPIEGATO

$$T(n) = 2 \cdot (T_n) + f(T_n)$$

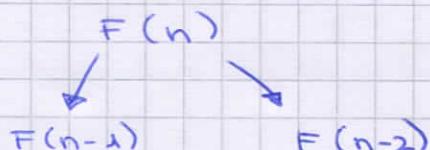
 $i(T_n) = n^{\circ}$  nodi interni $f(T_n) = n^{\circ}$  foglie

QUINDI:

$$\text{Congettura} \rightarrow f(T_n) = F_n ? \quad \forall n \geq 1$$

( $\Rightarrow$  Osserva: è vero che il numero di foglie equivale al numero di Fibonacci?

PROVA INDUTTIVAMENTE:

Per  $n=1, n=2$  valePasso induttivo:  $n \geq 3$ . Ipotesi induttiva: fino a  $(n-1)$  vale.Perciò vale anche per  $n$ .Se è vero per  $F(n-1)$  allora lo sarà per  $n-2$ !

$$\text{Congettura} \rightarrow i(T_n) = f(T_n) - 1 ? \quad \forall n \geq 1$$

( $\Rightarrow$  È vero che il numero di nodi interni equivale al numero di foglie scattato di 1?

PROVA

Base:  $n=1, 2$  vale(come sopra, vale per  $n-1$  quindi vale)

$$i(\tau') = f(\tau') - 1$$

|

$$i(\tau) = i(\tau') + 1$$

|

$$f(\tau) = f(\tau') + 1$$

|

$$i(\tau) = i(\tau') + 1$$

|

$$= f(\tau')$$

|

$$= f(\tau) - 1$$

QUINDI "TEMPO IMPIEGATO" È:

$$T(n) = 2 \cdot (T_n) + F_n$$

PER CASA

(1)

Dimostrare  $F_n \geq 2^{\frac{n}{2}}$   $\forall n > 0$ 

(2)

Calcolare Fib(3)

$\hookrightarrow$  L'algoritmo!

ALGORITMO: insieme di istruzioni definite passo passo per essere eseguite meccanicamente e produrre un determinato risultato.

Misura di tempo  $\rightarrow$  linee di codice

Relazione di ricchezza: il tempo richiesto da una routine è uguale al tempo speso all'interno della routine più il tempo speso per le chiamate ricorsive.

Programmazione dinamica: risolve i sottoproblemi, memorizza il risultato e lo usa poi.

### PROVA A DIMOSTRARE

$$F_n \geq 2^{\frac{n}{2}} \quad \forall n \geq 6$$

Base  $n=6$   
 $F_6 \geq 2^{\frac{6}{2}}$   
 $8 \geq 2^3$   
 $8 \geq 8 \quad \underline{\text{OK}}$

Passo ( $n=7$ )

$$F_n = F_{n-1} + F_{n-2} \text{ per definizione}$$

$$F_{n-1} \geq 2^{\frac{n-1}{2}}$$

$$F_{n-2} \geq 2^{\frac{n-2}{2}}$$

$$\begin{aligned} F_n &= 2^{\frac{n-1}{2}} + 2^{\frac{n-2}{2}} = 2^{\frac{n}{2}} \cdot 2^{-\frac{1}{2}} + 2^{\frac{n}{2}} \cdot 2^{-1} = 2^{\frac{n}{2}} \left( 2^{-\frac{1}{2}} + 2^{-1} \right) = \\ &= 2^{\frac{n}{2}} \underbrace{\left( \frac{1}{\sqrt{2}} + \frac{1}{2} \right)}_{>1} > 2^{\frac{n}{2}} \end{aligned}$$

$$T(n) = 3F_{n-2}$$

Questo algoritmo non funziona bene.  
È ricorsivo, quindi ogni volta ricalcola  $F_n$ . La soluzione è riscrivere iterativamente.

### Fib3 ITERATIVO:

1)

Fib3(int n)  $\rightarrow$  int

Allocare spazio (in modo dinamico)  
per  $F_n$  (array di n interi.)

2)  $F[1] := F[2] := 1$  ② ✗

3) for each  $i=3$  to  $n$  do

4)  $F[i] = F[i-1] + F[i-2]$  ③  $\rightsquigarrow$  verrà eseguito  $n-2$  volte [  $i=k \Rightarrow n-(k+1)$  ]

5) return  $F[n]$  ④

$$T(n) = ③ + n-1 + n-2 = 2n \quad (\text{soluzione iterativa})$$

Fib4(int n)  $\rightarrow$  int

a $\leftarrow$  1 b $\leftarrow$  1 ①

for i=3 to n

c $\leftarrow$  a+b ②

a $\leftarrow$  b ③

b $\leftarrow$  c ④

return b ⑤

$$T(n) = ④n - 5 \approx 4n$$

T(n)  $\rightarrow$  numero di linee di codice mandate in esecuzione

Fib4 è l'algoritmo Fib3 modificato per richiedere meno spazio in memoria.  
Ogni chiamata del ciclo usa i due valori precedenti di  $F_n$ , quindi basta usare 2 variabili al posto dell'array. (Quindi  $2n$  (algoritmo precedente) per  $2 = 4n$ )

26.09.2013

## NOTAZIONE ASINTOTICA

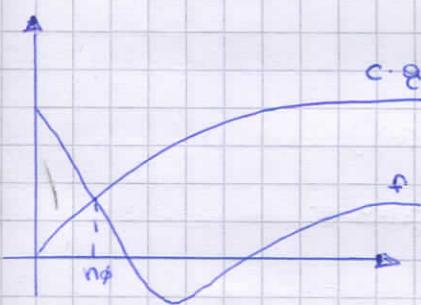
### DEFINIZIONE DI O-GRANDE

$$O(g(n)) = \{ f(n) \mid \exists c > 0, \exists n_0 \in \mathbb{N} \exists \forall n \geq n_0 : f(n) \leq c \cdot g(n) \} \rightsquigarrow \limsup_{n \rightarrow \infty} \frac{|f(n)|}{|g(n)|} < \infty$$

Il suo obiettivo è di caratterizzare le comportamenti di una funzione per argomenti elevati in modo semplice ma rigoroso, per poter confrontare i comportamenti di più funzioni tra loro.

In pratica, posso rappresentare due algoritmi come due funzioni [sempre positive]

### ESEMPIO



N.B! Se scrivo  $f(x) = O(g(x))$  sto dicendo che  $f(x)$  è dell'ordine di  $g(x)$ , non sto affermando che  $f$  è maggiore, in quanto  $O(g(x))$  rappresenta una CLASSE di funzioni.

→ studio come aumentano il tempo e la memoria a seconda dell'algoritmo

### ORDINI DI FUNZ. PIÙ COMUNI

- $O(1)$  costante
- $O(\log(n))$  logaritmica
- $O(n^2)$  quadratica
- $O(n)$  lineare

P.E.S.

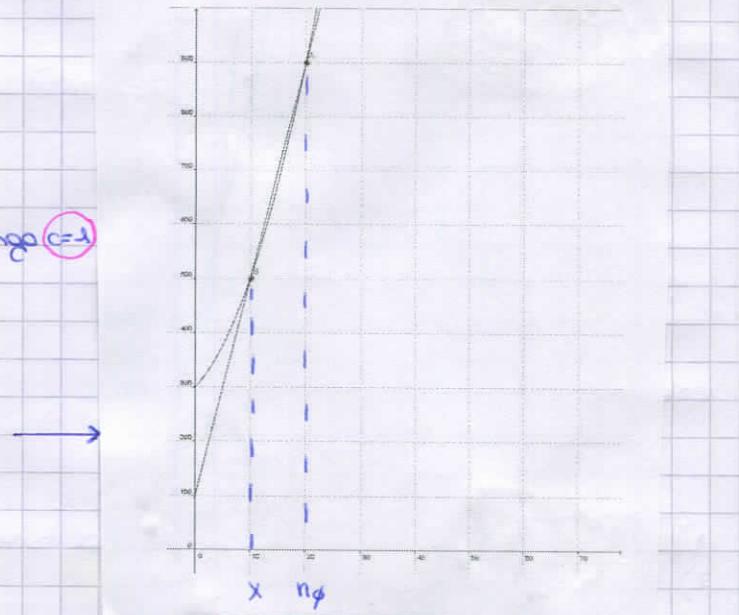
$$\frac{f(n)}{40n + 100} ? = O\left(\frac{g(n)}{n^2 + 10n + 300}\right) \approx \text{range } c=1$$

$$40n + 100 = n^2 + 10n + 300$$

$$40n + 100 - n^2 - 10n - 300 = 0$$

$$n^2 - 30n + 200 = 0$$

$$n_{1,2} = \frac{30 \pm \sqrt{100}}{2} = \frac{10}{20}$$



### DEFINIZIONE DI Ω-GRANDE

$$\Omega(g(n)) = \{ f(n) \mid \exists c > 0, \exists n_0 \in \mathbb{N} \exists \forall n \geq n_0 : f(n) \geq c \cdot g(n) \} \rightsquigarrow \liminf_{n \rightarrow \infty} \frac{|f(n)|}{|g(n)|} > 0$$

### PROPRIETÀ

$$\Leftrightarrow f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n)) \quad ] \star$$

ovvio → "  $f(n)$  ha ordine minore/g. di  $g(n)$  se  $g(n)$  ha ordine maggiore o uguale a  $f(n)$ "

## DIMOSTRAZIONE

( $\Rightarrow$ )

Ipotesi:  $f(n) = O(g(n))$  cioè  $\{ \exists c > 0, \exists n_0 \in \mathbb{N} \exists \forall n \geq n_0 : f(n) \leq c \cdot g(n) \}$

Tesi:

$f(n) = \Omega(g(n))$  cioè  $\{ \exists c' > 0, \exists n_0 \in \mathbb{N} \exists \forall n \geq n_0 : c' f(n) \leq g(n) \}$

Se prendo  $c'' = \frac{1}{c}$   $\Rightarrow c' \cdot f(n) \leq g(n) \Rightarrow \frac{1}{c} \cdot f(n) \leq g(n)$

## DEFINIZIONE DI $\Theta$ -GRANDE

$\Theta(g(n)) = \{ \exists c_1, c_2 > 0, n_0 \in \mathbb{N} \exists \forall n \geq n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \}$

$$\Leftrightarrow 0 < \liminf_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| \leq \limsup_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| < \infty$$

" $f(n)$  ha stesso ordine di grandezza di  $g(n)$ "

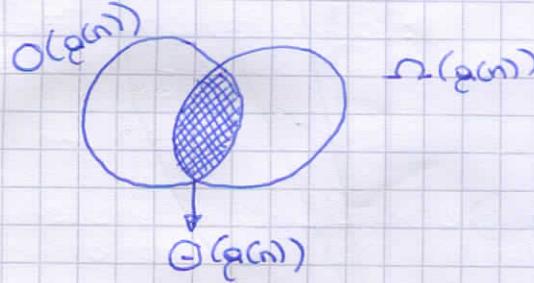
## PROPRIETÀ

$$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$$

" $f(n)$  ha stesso ordine di  $g(n)$  quando  $f(n)$  ha ordine minore/uguale di  $g(n)$  e, allo stesso tempo, maggiore/uguale di  $g(n)$ "

$$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$$

" $f(n)$  ha stesso ordine di  $g(n)$  se  $g(n)$  ha stesso ordine di  $f(n)$ "



## ESEMPIO

$$\sqrt{n+10} = \Theta(\sqrt{n})$$

$$\begin{aligned} 1) \sqrt{n+10} &= O(\sqrt{n}) \\ 2) \sqrt{n+10} &= \Omega(\sqrt{n}) \end{aligned} \quad ] \text{ DEVO DIMOSTRARE}$$

(1)  $\exists c > 0, \exists n_0 \in \mathbb{N} \exists \forall n \geq n_0 : \sqrt{n+10} \leq c\sqrt{n}$ ?  $\leftarrow$  DEFINIZIONE

$$\sqrt{n+10} \leq c \cdot \sqrt{n} \Leftrightarrow (\sqrt{n+10})^2 \leq (c \cdot \sqrt{n})^2 \quad \text{così tolgo le radici}$$

$$\sqrt{n+10} \leq c \cdot \sqrt{n} \quad | \quad \Leftrightarrow n+10 \leq c^2 \cdot n$$

$$n+10 - c^2 \cdot n \leq 0$$

porto tutto da una parte

$$n - c^2 \cdot n \leq 10$$

$$-n + c^2 \cdot n \geq 10$$

cambio il segno (quindi giro  $\leq$  che diventa  $\geq$ )

$$n \cdot (c^2 - 1) \geq 10$$

$$(c^2 - 1) \geq \emptyset$$

Raccolgo. Nota che  $(c^2 - 1) \geq \emptyset$  perché è un quadrato e  $c$  è maggiore di zero per definizione.

$$c > 1$$

$c$  è maggiore stretto di 1 perché se fosse uguale non funzionerebbe più per definizione, che ci dice di prendere un  $c > \emptyset$

Quindi  $n \geq \frac{10}{c^2 - 1}$

Perciò se per esempio prendo  $c = \sqrt{2}$  e  $i > 1$ , avrei  $n_0 = 10$

2)  $\exists c > \emptyset, \exists n_0 \in \mathbb{N} \ni \forall n \geq n_0 : \sqrt{n+10} \geq c \cdot \sqrt{n} \quad \triangleleft \text{DEFINIZIONE}$

$$\sqrt{n+10} \geq c \cdot \sqrt{n}$$

$$(\sqrt{n+10})^2 \geq (c \cdot \sqrt{n})^2$$

$$n+10 \geq c^2 \cdot n$$

$$n+10 - c^2 \cdot n \geq \emptyset$$

$$n - c^2 \cdot n \geq -10$$

$$c^2 \cdot n - n \leq 10$$

$$n(c^2 - 1) \leq 10$$

$$c^2 - 1 \leq \emptyset \rightarrow c^2 \leq 1 \rightarrow c \leq 1$$

ma è comunque positivo, perché  $c > 0$  per definiz. perciò  $0 < c \leq 1$

Potrei per esempio scegliere  $c = i$  e avrei  $n_0 = 1$

### PER CASA

$$\frac{1}{2} n^2 - 3n = \Theta(n^2)$$

$$1) \frac{1}{2} n^2 - 3n = O(n^2)$$

Perchè  $\Theta$ -grande è fatto di  $O$ -grande e  $\Omega$ -grande

$$2) \frac{1}{2} n^2 - 3n = \Omega(n^2)$$

DEFINIZ. DI  $\Theta$ -GRANDE

$$1) \exists c > \emptyset, \exists n_0 \in \mathbb{N} \ni \forall n \geq n_0 : \frac{1}{2} n^2 - 3n \leq c \cdot n^2 \quad \triangleleft$$

$$\frac{1}{2} n^2 - 3n \leq c \cdot n^2$$

$$\frac{1}{2} n^2 - 3n - c \cdot n^2 \leq \emptyset \quad \triangleright \text{divide per } n$$

$$\frac{1}{2} n - 3 - c \cdot n \leq \emptyset$$

$$\frac{1}{2} n - c \cdot n \leq 3$$

$$n \left( \frac{1}{2} - c \right) \leq 3$$

Raccolgo  $n$

$$\frac{1}{2} - c < \emptyset \Rightarrow c > \frac{1}{2}$$

2)  $\exists c > 0, \exists n_0 \in \mathbb{N} \ni \forall n \geq n_0 : \frac{1}{2}n^2 - 3n \geq c \cdot n^2 \leftarrow \text{DEFINIZIONE } O\text{-GRANDE}$

$$\frac{1}{2}n^2 - 3n \geq c \cdot n^2$$

$$\frac{1}{2}n^2 - 3n - c \cdot n^2 \geq 0$$

$$\frac{1}{2}n^2 - 3n - c \cdot n^2 \geq 0$$

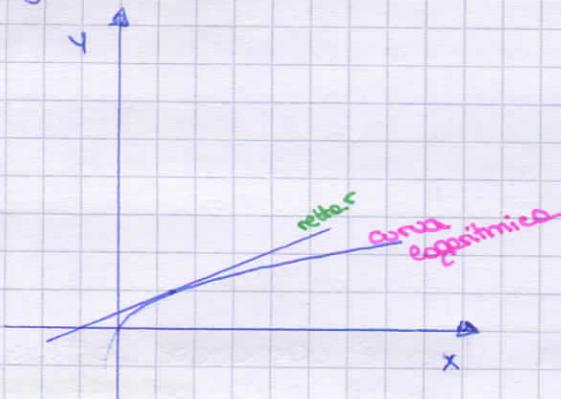
$$n\left(\frac{1}{2} - c\right) \geq 3$$

$$n \geq \frac{3}{\left(\frac{1}{2} - c\right)}$$

$\rightarrow$  Qui lo posso fare perché so che  $\left(\frac{1}{2} - c\right)$  è positivo.  
Nel punto  $\rightarrow$  non si poteva.

### ESERCIZIO 1 COL PROF.

$$\log n = O(n)$$



26.09.2012

### OSSERVAZIONE GRAFICA:

Comunque io prenda un punto sulla curva esponenziale la retta tangente passante per quel punto sarà sempre al di sopra della curva.



DICO CHE, DATO  $\star$ , AVRO':

$$\log x \leq x-1 < x$$

(questo rende  $x$  del una retta)

$$\text{perciò } \log x < x$$

La definizione di  $O$ -grande dice che  $\{\exists c > 0, \exists n_0 \in \mathbb{N} \ni \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$ ,  
perciò se ho che  $\log x$  è  $f(n)$  e  $x$  è  $g(n)$ ,  $c=1$ , è vero che  
 $\log x = O(n)$ .

### ESEMPIO.

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

$$(n \leq 2^n \Rightarrow n \cdot \log n = O(n^2))$$

dove  $n$  è una funzione

### ESERCIZIO 2

$$f(n) = n \sqrt{n} \quad g(n) = 9^{\log_3 n} = n^2 \quad 9^{\log_3 n} = (3^2)^{\log_3 n} = 3^{2 \log_3 n} = 3^{\log_3 n^2} = n^2$$

$$n \sqrt{n} = O(n^2)$$

Sappiamo che  $n \leq n^2$ .  $\sqrt{n} < n$ . Perciò  $n \sqrt{n} \leq n^2$ .

### ESERCIZIO 3

$$n! = O(n)$$

Sappiamo che:

$$n! = n \cdot n-1 \cdot n-2 \dots$$

$$n^n = \underbrace{n \cdot n \cdot n \cdot \dots \cdot n}_{n-\text{ volte}}$$

Perciò  $n! \leq n^n$

### ESERCIZIO 4

$$n! = \Omega(2^n)$$

$$n! = 1 \cdot 2 \cdot \dots \cdot n \geq \underbrace{1 \cdot 2 \cdot 2 \cdot \dots \cdot 2}_{n-1} = 2^{n-1}$$

Dove  $f(n) = n!$   
 $g(n) = 2^n$

$$\forall n \geq 1 \quad n! \geq 2^{n-1} = \left(\frac{1}{2}\right) 2^n \quad f(n) \geq c \cdot g(n)$$

### ESERCIZIO 5

$$\log n! = O(n \log n)$$

$$\log n! = \log \prod_{i=1}^n i = \sum_{i=1}^n \log i \leq n \log n$$

definizione di  $O$ -grande

Le somma dei logaritmi del prodotto è la somma dei logaritmi

$n!$  potrebbe essere un fattoriale grande. Per calcolarlo posso usare la FORMULA DELL'APPROSSIMAZIONE DI STIRLING :

$$n! = \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n \left(1 + \frac{O(n)}{n}\right) \quad \text{dove } e = \text{esponentiale}$$

$$f(n) + g(n) = \Theta(\max\{f(n), g(n)\})$$

$\forall n \in \mathbb{N} \quad h(n) = \max\{f(n), g(n)\}$

$$\rightarrow \forall n \geq 1 \quad f(n) + g(n) \leq 2h(n)$$

$$f(n) = \max\{f(x), g(x)\} \quad \vee \quad g(n) = \max\{f(x), g(x)\}$$

$$f(n) + g(n) \geq \max\{f(n), g(n)\}$$

### ESERCIZIO 6

$$a, b \in \mathbb{R} \quad b > \phi \quad (n+a)^b = \Theta(n^b)$$

$$\exists c_1, c_2 > \phi, \exists n_0 \in \mathbb{N} \ni \forall n \geq n_0 : c_1 \cdot n^b \leq (n+a)^b \leq c_2 \cdot n^b \quad \text{DEFINZ. DI } \Theta$$

Considero due casi :  $a > \phi$  (I)  
 $a < \phi$  (II)

I)  $a > \phi$

$$c_1 \cdot n^b \leq (n+a)^b \leq c_2 \cdot n^b$$

$$c_1^{1/b} \cdot n \leq n+a \leq c_2^{1/b} \cdot n$$

$\rightarrow$  divido gli esponenti per  $b$

$$n+a \leq c_2^{1/b} \cdot n$$

$$n+a - c_2^{1/b} \cdot n \leq \phi$$

$$n - c_2^{1/b} \cdot n \leq -\phi$$

$$-n + c_2^{1/b} \cdot n \geq \phi$$

Considero sotto  $(n+a)^b \leq c_2 \cdot n^b$   
perchè ho  $a > \phi$

$$n \left( c_2^{\frac{1}{b}} - 1 \right) \geq a$$

$$n \geq \frac{a}{c_2^{\frac{1}{b}} - 1}$$

$$\begin{aligned} c_2^{\frac{1}{b}} - 1 &> \phi, \quad c_2^{\frac{1}{b}} > 1, \quad c_2 > 1 \\ n \geq a & \quad n_{\phi} = \lceil a \rceil \end{aligned}$$

Se per esempio prendo  $c_1 = 1$   
 $c_2 = 2^b$  ho  $n_{\phi} = \lceil a \rceil \leftarrow$  "parte intera superiore"

II)  $a < \phi$

Dovrò trovare  $c_2$ . Se  $c_2$  fosse = 1, avrei  $n$ ,  $n$  è sempre maggiore di  $n-a$

$$c_1^{\frac{1}{b}} \cdot n \leq n + a$$

$$-c_1^{\frac{1}{b}} \cdot n + n \geq -a$$

$$n \left( 1 - c_1^{\frac{1}{b}} \right) \geq -a$$

$\underbrace{\quad}_{> \phi}$

$$\begin{aligned} 1 - c_1^{\frac{1}{b}} &> \phi \\ c_1 &> 1 > c_1^{\frac{1}{b}} \\ c_1 &> c_1 \end{aligned}$$

perché  $a$  in questo caso è negativo, quindi ho  $(n+a)$

$$a, b \in \mathbb{R}, b > 0$$

$$\text{Se prendo } c_1 = \left(\frac{1}{2}\right)^b, \quad c_2 = 1, \quad n_{\phi} = \lceil 2|a| \rceil$$

### DEFINIZIONE DI O-PICCOLO

$$O(g(n)) = \{ f(n) \mid \forall c > \phi, \exists n_0 \in \mathbb{N} \exists \forall n \geq n_0 : \phi \leq f(n) < c \cdot g(n) \}$$

PROPRIETA'

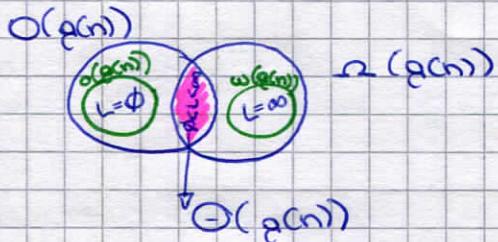
$$f(n) = O(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \phi$$

### DEFINIZIONE DI O-PICCOLO

$$O(g(n)) = \{ f(n) \mid \forall c > \phi, \exists n_0 \in \mathbb{N} \exists \forall n \geq n_0 : \phi \leq c \cdot g(n) < f(n) \}$$

$$f(n) = O(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty$$

Per la notazione asintotica si possono utilizzare anche i limiti.



## CLASSI DI COMPORTAMENTO ASINTOTICO

### LIMITE DI UNA SUCCESSIONE

$$\{a_n\}_{n \in \mathbb{N}}$$

Studiare il comportamento asintotico = calcolare il limite

$$\lim_{n \rightarrow \infty} a_n = L \in \mathbb{R}$$

"Comunque io prenda  $n$ , la successione  $a_n$  si avvicina sempre di più a  $L$ ".

### DEFINIZIONE DI LIMITE (SE LA SUCCESSIONE CONVERGE)

$$\forall \epsilon > 0 \exists n_0 \in \mathbb{N} \exists' \forall n \geq n_0 : |a_n - L| \leq \epsilon$$

↓  
distanza tra  $a_n$  e  $L$

$$\begin{aligned} \phi &\leq f(n) < \infty \\ \phi'' &\\ \Rightarrow a_n &\leq \epsilon \end{aligned}$$

$$f(n) = o(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \phi \cancel{\neq} f(n) = O(g(n))$$

↳ perché  $\phi$ -piccolo è sottoinsieme di  $O$ -grande (non vale il contrario)

### DEFINIZIONE DI LIMITE (SE LA SUCCESSIONE NON CONVERGE)

$$\lim_{n \rightarrow \infty} a_n = +\infty$$

"Comunque prenda  $n$ , da un certo punto in poi la successione sarà più grande".

$$\forall c > \phi \exists n_0 \in \mathbb{N} \exists' \forall n \geq n_0 : c \leq a_n$$

$$\omega(g(n)) = \phi \leq c < \frac{f(n)}{g(n)}$$

$$f(n) = \omega(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty = f(n) = \Omega(g(n))$$

↳ come prima

### RIASSUMENDO

$$\forall c > \phi \exists n_0 \in \mathbb{N} \exists' \forall n \geq n_0 :$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \begin{cases} \phi \Rightarrow f(n) = O(g(n)) = o(g(n)) \\ L (\text{con } \phi < L < +\infty) \Rightarrow f(n) = \Theta(g(n)) \\ \infty \Rightarrow f(n) = \Omega(g(n)) = \omega(g(n)) \end{cases}$$

DICOLOSTRO CHE:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = L \in \mathbb{R} [0 < L < +\infty] \Rightarrow f(n) = \Theta(g(n)) \star$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = L \Rightarrow O(g(n))$$

Se  $L = \phi$  sarebbe  $\phi$ -piccolo  $\Rightarrow O$   
 Se  $L > \phi$  sarebbe nell'intersezione

PER DEFINIZIONE  $\left| \frac{f(n)}{g(n)} - L \right| < \varepsilon$

È vero se

$$\varepsilon < \frac{|f(n)|}{|g(n)|} - L < \varepsilon$$

$$(L-\varepsilon) < \frac{|f(n)|}{|g(n)|} < (L+\varepsilon)$$

$$(L-\varepsilon) \cdot |g(n)| < |f(n)| < (L+\varepsilon) \cdot |g(n)|$$

Se è vero per ogni  $\varepsilon$ , significa che è vero anche per  $\varepsilon = 1$

ottengo  $(L-1) \cdot |g(n)| < |f(n)| < (L+1) \cdot |g(n)|$

mi ricordo c1 e c2!

Quindi ho dimostrato  $\star$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = L \Rightarrow f(n) = \Theta(g(n))$$

$\hookrightarrow$  come sopra

### ESERCIZIO

$$(n+a)^b = \Theta(n^b)$$

$$\lim_{n \rightarrow \infty} \frac{(n+a)^b}{n^b} = \lim_{n \rightarrow \infty} \left( \frac{n+a}{n} \right)^b = \lim_{n \rightarrow \infty} \left( 1 + \frac{a}{n} \right)^b = 1 = L$$

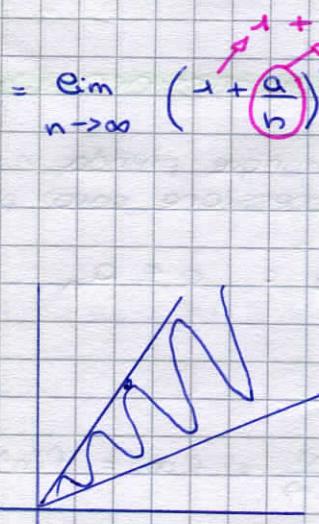
### ESERCIZIO

$$f(n) = n \underset{\leq 2}{\cancel{(1 + \sin(n))}}$$

$$g(n) = n$$

$$n(1 + \sin(n)) \leq 2n$$

proviamo con l'elimite



$$\lim_{n \rightarrow \infty} \frac{y(1 + \sin(n))}{n} = \lim_{n \rightarrow \infty} 1 + \sin(n) \quad \text{MA } \sin(n) \text{ non ha limite!}$$

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

### ESERCIZIO

$$f(n) = \underset{\text{prende}}{\cancel{n^8 + 7n^7 - 10n^5 - 2n^4 + 3n^2 - 17}} = \Theta(n^8) \quad \text{ma Questo algoritmo ha complessità } n^{8+k}$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \frac{n^8 + 7n^7 - 10n^5 - 2n^4 + 3n^2 - 17}{n^8} = \lim_{n \rightarrow \infty} \left( 1 + \frac{7}{n} - \frac{10}{n^3} - \frac{2}{n^4} + \frac{3}{n^5} - \frac{17}{n^8} \right) \underset{\Rightarrow \emptyset}{\cancel{}}$$

$\psi(n)$  è polinomio di ordine  $k$ , allora  $\psi(n) = \Theta(n^k)$

## ESERCIZIO

$$\log(n) \quad \sqrt{n}$$

$$\lim_{n \rightarrow \infty} \frac{\log(n)}{\sqrt{n}} \stackrel{\text{Hôpital}}{=} \lim_{n \rightarrow \infty} \frac{1/n}{\frac{1}{2} n^{-1/2}} = \lim_{n \rightarrow \infty} \frac{2\sqrt{n}}{n} = \lim_{n \rightarrow \infty} \frac{2}{\sqrt{n}} = \emptyset$$

↑ derivate  
↓ derivate

## ESERCIZIO PER CASA

$$1) f(n) + o(f(n)) \stackrel{?}{=} O(f(n))$$

Come nel polinomio, cb' lo termine che prevede ( $f(n)$ ) e gli altri sono o-piccoli

$$O(f(n)) \rightarrow \lim_{n \rightarrow \infty} f(n) = L \quad \emptyset < L < +\infty$$

$$f(n) + o(f(n)) = f(n) + \emptyset = f(n) \quad \text{OK}$$

$$2) \log n + \sqrt{n} \stackrel{?}{=} O(\log(n))$$

$$\lim_{n \rightarrow \infty} \frac{\log n}{\log n} + \frac{\sqrt{n}}{\log n} = 1 + \text{qualcosa} = L \rightarrow \emptyset < L < +\infty$$

## ESERCIZIO

$$n = O(n \log \log n)$$

Cosa succede se considero  $n^{1+\epsilon} = O(n \log \log n) \quad \epsilon > \emptyset \quad ?$

$$n \log \log n = O(n^{1+\epsilon})$$

$$\lim_{n \rightarrow \infty} \frac{n \log \log n}{n^{1+\epsilon}} = \lim_{n \rightarrow \infty} \frac{\cancel{n} \log \log n}{\cancel{n} \cdot n^\epsilon} \stackrel{\text{Hôpital}}{=} \lim_{n \rightarrow \infty} \frac{\frac{1}{n} \cdot \frac{1}{\log n}}{\epsilon n^{\epsilon-1}} =$$

↑ derivate  
↓ derivate

$$= \lim_{n \rightarrow \infty} \frac{1}{\epsilon} \cdot \frac{1}{n^\epsilon \log n} = \emptyset = O(n^{1+\epsilon})$$

$$O(1) \Rightarrow f(n) = C \quad \text{cioè: } O\text{-grande di } 1 = \text{costante}$$

## PER CASA

Ordina secondo O-grande:

$$n^2, n \log n, n^3 + \log n, \sqrt{n}, n^2 + 2n \log n, \log \log n, 17 \log n, 10n^{3/2}, n^5 - n^4 + 2n, \\ 5n^2 \log \log n, 3n^2 + n^3 \log n, n + 6 \log n$$

## SOLUZIONE (DAL PIÙ GRANDE AL PIÙ PICCOLO)

$$n^5 - n^4 + 2n, 3n^2 + n^3 \log n, 10n^{3/2}, n^3 + \log n, n + 6 \log n, n^2, 2n \log n, \sqrt{n}, 17 \log n, \\ n \log n, \log \log n, 5n^2 \log \log n.$$

Ricerca Seq (lista L, elemento x)

```

for each y ∈ L do
    if (y = x) then return TRUE
return FALSE

```

$T_{best}(n) = O(1) = \text{Costante}$  ← TEMPO MIGLIORE (NON DIPENDE DALLA DIM. DEL DATO IN INPUT)

$T_{worst}(n) = C \cdot n = \text{costante} \cdot n = O(n)$

$T_{medio}(n) = C \cdot \frac{n}{2} = O(n)$  ↳ bisognerebbe conoscere la distribuzione

C'iponiamo sempre nel caso peggiore, con  $T_{worst}$ .

$T_{worst}(n) = \max \text{ istanze} I \dim(n) - T(I)$

$T_{best}(n) = \min \text{ istanze } I \dim(n)$

$$T_{avg} = \sum_{\substack{\text{istanza} \\ \text{di dim } n}} \varphi(I) \cdot T(I)$$

↳ Probabilità di istanza

## ALGORITMO

1) Operazione logico-aritmetica / assegnazione  $\equiv O(1)$  [costante]

2) if (test) then Body<sub>1</sub> else Body<sub>2</sub>

Tempo di esecuzione di un if =  $T(\text{test}) + \max \{T(B_1) + T(B_2)\}$

↳ Il più peggiore tra i due di B<sub>1</sub> e B<sub>2</sub>

3) for i=1 to n

Body

$$\sum_{i=1}^n T_i \longrightarrow t_{\text{exe corpo}}$$

4) while (test) do

Body

5) repeat

Body  
until (Test)

$$\sum_{i=0}^m T_i + \sum_{i=0}^{n-1} T_i$$

6) funz(x)

7) seq di istc.  
 $I_1$   
 $I_2$   
 $I_3$

## Fattoriale (n)

```

iris = 1
for i=1 to n do
    iris = iris * i
return iris

```

$$T(n) = C_1 + \sum_{i=1}^n C_2 = C_1 + C_2 n = \Theta(n)$$

PER CASA

Scrivere un algoritmo in pseudocodice che prenda da lista 

3	5	4	6	3	2	9
---	---	---	---	---	---	---

 e determini il massimo sonale (maggiorre di precedente o successivo) LOCAL MAX

Svolgimento

LocalMax (lista L)

```

L = 3, 5, 4, 6, 3, 2, 9
while (L != '0')
    if (L > L-1 && L < L+1)
        max = L
    else
        if (L-1 > L)
            max = L-1;
        else
            max = L+1;
    return max;

```

ALGORITMI RICORSIVI

RICORRENZA

$$T(n) = \begin{cases} 1 & n=1 \\ C + T\left(\frac{n}{2}\right) & n \geq 2 \end{cases}$$

METODO DELL'ITERAZIONE

$$T(n) = C \cdot T\left(\frac{n}{2}\right)$$

$$= C + C + T\left(\frac{n}{2^2}\right) = 2C + T\left(\frac{n}{2^2}\right) = 2C + \left[C + T\left(\frac{n}{2^3}\right)\right] = 3C + T\left(\frac{n}{2^3}\right)$$

$$\hookrightarrow kC + T\left(\frac{n}{2^k}\right) \rightarrow k = \log_2 n$$

$$C \cdot \log_2 n + 1 = \Theta(\log n)$$

$$T(n) = \begin{cases} 9T\left(\frac{n}{3}\right) + n & n > 1 \\ 1 & n = 1 \end{cases}$$

$$T(n) = 9T\left(\frac{n}{3}\right) + n$$

$$= 9 \left[ 9T\left(\frac{n}{3^2}\right) + \frac{n}{3} \right] = 9^2 T\left(\frac{n}{3^2}\right) + (3n + n)$$

$$= 9^3 \cdot T\left(\frac{n}{3^3}\right) + (3^2 n + 3n + n) = 9^3 \cdot T\left(\frac{n}{3^3}\right) + (3^2 n + 3^n + 3^0 n)$$

$$T(n) = q^k T\left(\frac{n}{3^k}\right) + n \sum_{i=0}^{k-1} 3^i$$

progressione geometrica

$$= q^k T\left(\frac{n}{3^k}\right) + n \frac{3^{k-1}}{2}$$

Impongo  $\frac{n}{3^k} = 1$ ,  $k = \log_3 n$

$$\begin{aligned} T(n) &= q^{\log_3 n} + n \frac{3^{\log_3 n - 1}}{2} = \\ &= 3^{\log_3 n} + n \frac{n-1}{2} = 3^{\log_3 n} + \frac{n(n-1)}{2} = \Theta(n^2) \end{aligned}$$

9.10.13

### ABBIAMO VISTO

- 1) Metodo delle iterazioni
- 2) Metodo di sostituzione

### ESEMPIO

$$T(n) = \begin{cases} 1 & \text{se } n=1 \\ T\left(\lfloor \frac{n}{2} \rfloor\right) + n & \text{se } n > 1 \end{cases}$$

$$T(n) = O(n)? \quad \exists c > 0 \text{ tale che } T(n) \leq c \cdot n$$

$$\begin{aligned} T(n) &= T\left(\lfloor \frac{n}{2} \rfloor\right) + n \\ &\leq c \left\lfloor \frac{n}{2} \right\rfloor + n \leq c \cdot \frac{n}{2} + n = \left(\frac{c}{2} + 1\right) n \stackrel{?}{\leq} c \cdot n \end{aligned}$$

$$\text{dove } \frac{c}{2} + 1 \leq c \rightarrow c \geq 2$$

### DIVIDE ET IMPERA

$$T(n) = \underbrace{T_D(n)}_{\text{DIVIDE}} + \underbrace{\sum_{i=1}^a T(n_i)}_{\text{COMBINE}} + T_C(n)$$

$$\boxed{T(n) = \begin{cases} a T\left(\frac{n}{b}\right) + f(n) & n > 1 \\ 1 & n = 1 \end{cases}}$$

★

$a \geq 1$   
 $b > 1$   
 $d = \log_b a$

### TEOREMA MASTER

Sia  $T(n)$  definita come in ★, con  $a \geq 1$ ,  $b > 1$ ,  $f(n)$  asintoticamente non negativa.

- 1)  $T(n) = \Theta(n^a)$ , se  $f(n) = O(n^{a-\epsilon})$ , per  $\epsilon > 0$
- 2)  $T(n) = \Theta(n^a \log n)$ , se  $f(n) = \Theta(n^a)$
- 3)  $T(n) = \Theta(f(n))$ , se
  - [3.1]  $f(n) = \Omega(n^{a+\epsilon})$ ,  $\epsilon > 0$
  - [3.2]  $\exists c < \epsilon \text{ c.c. } af\left(\frac{n}{b}\right) \leq cf(n) \text{ per } n \text{ suffic. grande}$

### ESEMPIO 1

$$T(n) = 3T\left(\frac{n}{2}\right) + n^2$$

Siamo nelle condizioni di usare master?

$a = 3$   
 $b = 2$   
 $d = \log_2 3$   
 $f(n) = n^2$

Si

Ora devo vedere se è  $\Theta - \Theta - \Omega$

$$n^2 = \Omega\left(n^{\log_2 3 + \varepsilon}\right) \quad \varepsilon > 0 \quad \rightarrow \text{verificata in 3.1}$$

$$\varnothing < \varepsilon \leq 2 - \log_2 3 \quad \text{rende vero} \quad \checkmark$$

$$\log_2 3 + \varepsilon \leq 2$$

$$n^2 = \Omega\left(n^{\log_2 3 + \varepsilon}\right)$$

$$\exists c < 1 \quad t.c. \quad 3\left(\frac{n}{2}\right)^2 \leq c \cdot n^2 \quad ? \quad \rightarrow \text{verificato in 3.2}$$

$$3\left(\frac{n}{2}\right)^2 \leq c \cdot n^2$$

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

$$\hookrightarrow \frac{3}{4}n^2 \leq cn^2 \quad \rightarrow \frac{3}{4} \leq c < 1$$

### ESEMPIO 2

$$T(n) = 6T\left(\frac{n}{2}\right) + n^2$$

$$a = 6 \quad d = \log_2 6 = 2.58 \quad f(n) = n^2$$

$$n^d = n^2 \quad \rightarrow \text{caso 2} \quad \rightarrow T(n) = \Theta(n^2 \log n)$$

### ESEMPIO 3

$$T(n) = T\left(\frac{n}{2}\right) + 2^n$$

$$a = 1 \quad d = \log_2 2 = 1 \quad f(n) = 2^n$$

devo confrontare  $2^n$  con  $n^d$

$$\varepsilon > 0. \quad f(n) = 2^n = \Omega(n^\varepsilon)$$

$$2^n = \Omega(n) \Rightarrow \text{scelgo } \varepsilon = 1$$

$$\exists c < 1 \quad t.c. \quad af\left(\frac{n}{b}\right) \leq c \cdot f(n) \quad ?$$

$\hookrightarrow$  ho che  $\Omega(n^{0+1}) = \Omega(n)$   
 [in def. dice  $\Omega(n^{d+\varepsilon})$ ]

$$\sqrt[n]{2^n} \leq c \cdot 2^n = c \cdot \sqrt[n]{2^n} \cdot 2^{\frac{n}{2}}$$

$$\hookrightarrow 1 \leq c \cdot 2^{\frac{n}{2}}$$

$$\hookrightarrow \frac{1}{2^{\frac{n}{2}}} \leq c < 1$$

concluiso che è il caso 3

### ESEMPIO

$$T(n) = 2^n T\left(\frac{n}{2}\right) + n^2$$

$$\begin{aligned} a &= 2^n \\ b &= 2 \\ d &= \log_2 2^n = n \end{aligned}$$

$\rightarrow$  NO TEOREMA MASTER PERCHÉ  $a$  NON È COSTANTE!

### ESEMPIO

$$T(n) = 16 T\left(\frac{n}{4}\right) + n$$

$$\begin{aligned} a &= 16 \\ b &= 4 \\ d &= \log_4 16 = 2 \end{aligned}$$

$\varepsilon > \phi$  t.c.  $n = O(n^{2-\varepsilon})$  se prendo  $\varepsilon = 1$  ottengo:  $n = O(n^1) \rightarrow n = O(n)$

Primo caso,  $T(n) = \Theta(n^2)$

### ESEMPIO

$$T(n) = \frac{1}{2} T\left(\frac{n}{2}\right) + \frac{1}{2}$$

$$\begin{aligned} a &= \frac{1}{2} \quad \text{ma} \rightarrow a < 1 \text{ quindi NO MASTER} \\ b &= 2 \\ d &= \log_2 \frac{1}{2} = -1 \end{aligned}$$

### ESEMPIO

$$T(n) = 2T\left(\frac{n}{2}\right) + n \log n$$

$$\begin{aligned} a &= 2 \\ b &= 2 \\ d &= \log_2 2 = 1 \end{aligned}$$

Dimostra che non si può usare master perché non si verifica nessuna delle opzioni

1)  $n \log n = O(n^{1-\varepsilon}) \quad \varepsilon > \phi$  ?

$c > \phi$  per  $n$  sufficientemente grande  $n \log n \leq c \cdot n^{1-\varepsilon} = \frac{n}{n^\varepsilon} \rightarrow n^\varepsilon \log n \leq c$

2)  $n \log n = \Theta(n) \rightarrow n \log n \in \Theta(n)$

$n \log n = O(n)$  ?

$\lim_{n \rightarrow \infty} \frac{n \log n}{n} = +\infty$  Quindi non è mai  $O(n)$ , quindi non vale

3)  $n \log n = \Omega(n^{1+\varepsilon})$  per  $\varepsilon > \phi$

$$\lim_{n \rightarrow \infty} \frac{n \log n}{n^{1+\varepsilon}} = \lim_{n \rightarrow \infty} \frac{n \log n}{\sqrt[1+\varepsilon]{n}} = 0$$

$\downarrow$  premesse

Quindi  $n \log n = o(n^{1+\varepsilon})$ .  $o$ -piccolo e  $\Omega$ -grande sono disgiunti. Quindi non vale.

### ESERCIZI PER CASA

$$\textcircled{1} \quad T(n) = 2T\left(\frac{n}{4}\right) + n^{0.51}$$

$$\textcircled{2} \quad T(n) = 3T\left(\frac{n}{3}\right) + \sqrt{n}$$

$$\textcircled{3} \quad T(n) = 64T\left(\frac{n}{8}\right) - n^2 \log n$$

# RISOLVO ESERCIZI PER CASA

$$\textcircled{1} \quad T(n) = 2T\left(\frac{n}{4}\right) + n^{0.5^1}$$

$$a = 2 \quad d = \log_4 2 = \frac{1}{2}$$

$$b = 4$$

$$f(n) = n^{0.5^1}$$

$$\textcircled{2} \quad T(n) = 3T\left(\frac{n}{3}\right) + \sqrt[n]{n^1}$$

$$a = 3$$

$$b = 3$$

$$d = \log_3 3 = 1$$

$$f(n) = \sqrt[n]{n^1}$$

$$\textcircled{3} \quad T(n) = 64T\left(\frac{n}{8}\right) + \underline{n^2 \log n}$$

$$a = 64$$

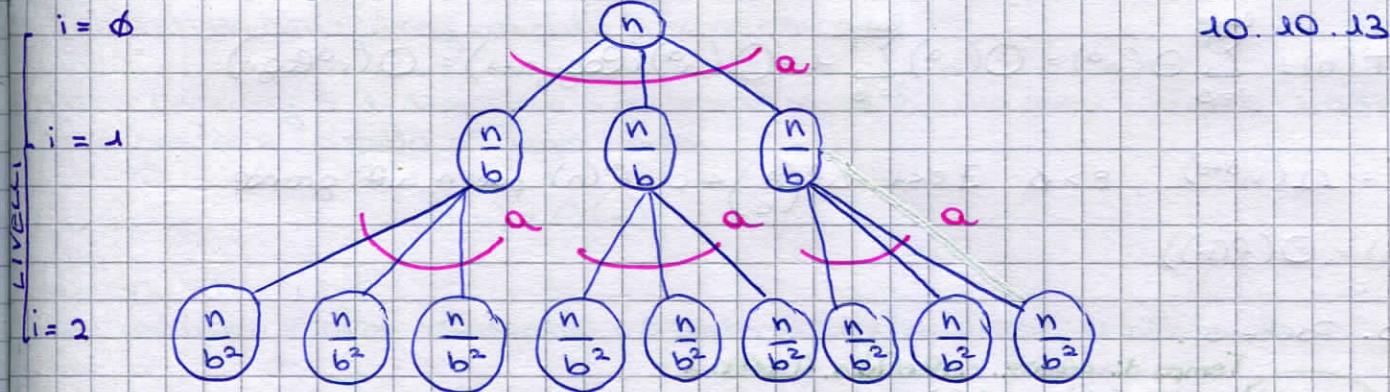
$$b = 8$$

$$d = \log_8 64 = 2$$

$$f(n) = \underline{\square} n^2 \log n$$

C<sub>o</sub> non posso applicare il teorema master.

## SPIEGAZIONE DEL DIVIDE ET IMP.



1) a livello  $i$ : la dimensione dei sottoproblemi è  $\frac{n}{b^i}$ ,  $i = \emptyset, 1, \dots$

2) il contributo di un nodo a livello  $i$ : sul tempo di esecuzione è  $f\left(\frac{n}{b^i}\right)$

3) il numero massimo di livelli dell'albero è  $\log_b n$

4) a livello  $i$  ci sono  $a^i$  nodi

$$a^i = \frac{\log_b n}{n}$$

5) il numero di foglie è  $n^d$  ( $d = \log_b a$ )

$$V(n) = \sum_{i=0}^{\log_b n} a^i$$

$$T(n) = \sum_{i=0}^{\log_b n} a^i f\left(\frac{n}{b^i}\right)$$

## DIMOSTRAZIONE DEL TEOREMA MASTER

$$1) f(n) = O(n^{d-\varepsilon}), \varepsilon > 0 \Rightarrow T(n) = \Theta(n^d)$$

dimostra.

$$\begin{aligned} a^i f\left(\frac{n}{b^i}\right) &= a^i O\left(\left(\frac{n}{b^i}\right)^{d-\varepsilon}\right) \\ &= O\left(a^i \left(\frac{n}{b^i}\right)^{d-\varepsilon}\right) \\ &= O\left(n^{d-\varepsilon} \cdot \frac{a^i}{(b^i)^d (b)^{-\varepsilon}}\right) = O\left(n^{d-\varepsilon} \cdot \frac{a^i \cdot (b^\varepsilon)^i}{a^i}\right) = O\left(n^{d-\varepsilon} \cdot (b^\varepsilon)^i\right) \end{aligned}$$

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_b n} O\left(n^{d-\varepsilon} \cdot (b^\varepsilon)^i\right) = O\left(\sum_{i=0}^{\log_b n} n^{d-\varepsilon} \cdot (b^\varepsilon)^i\right) = O\left(n^{(d-\varepsilon)} \sum_{i=0}^{\log_b n} (b^\varepsilon)^i\right) = \\ &= O\left(n^{d-\varepsilon} \cdot \frac{(b^\varepsilon)^{\log_b n + 1}}{b^\varepsilon - 1}\right) = O\left(n^{d-\varepsilon} \cdot \frac{(b^\varepsilon)^{\log_b n} \cdot b^{\varepsilon-1}}{b^\varepsilon - 1}\right) \end{aligned}$$

$$T(n) \geq n^d \Rightarrow T(n) = \Omega(n^d)$$

$$2) T(n) = \Theta(n^d) \Rightarrow T(n) = \Theta(n^d \log n)$$

dimostra.

$$a^i f\left(\frac{n}{b^i}\right) = \Theta\left(a^i \left(\frac{n}{b^i}\right)^d\right) = \Theta\left(n^d \frac{a^i}{(b^i)^d}\right) = \Theta\left(n^d \frac{a^i}{(b^d)^i}\right) = \Theta(n^d)$$

$$T(n) = \sum_{i=0}^{\log_b n} \Theta(n^d) = \Theta(n^d) \sum_{i=0}^{\log_b n} 1 = \Theta(n^d) (\log_b n + 1) = \Theta(n^d \log n)$$

$$3) f(n) = \Omega(n^{d+\varepsilon}), \varepsilon > 0 \quad \exists c < 1 \quad a^i f\left(\frac{n}{b^i}\right) \leq c f(n) \text{ per } n \text{ suff. grande}$$

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

dimostra. Parte a.

Tempo di esecuz. calcolata questo 1

$$a^i f\left(\frac{n}{b^i}\right) \leq c \cdot f(n) < f(n)$$

$$\forall i : a^i f\left(\frac{n}{b^i}\right) \leq c f(n)$$

induzione. suppongo vero fino a(i-1)

$$a^i f\left(\frac{n}{b^i}\right) = a \cdot a^{i-1} \cdot f\left(\frac{n/b}{b^{i-1}}\right) \leq a \cdot c^{i-1} f\left(\frac{n}{b^{i-1}}\right) = c^{i-1} a^i f\left(\frac{n}{b^i}\right) \leq c^{i-1} c f(n) = c^i f(n)$$

$$T(n) \leq \sum_{i=0}^{\log_b n} c^i f(n) = f(n) \sum_{i=0}^{\log_b n} c^i \xrightarrow{\text{c} < 1, \text{SERIE GEOMETRICA}} f(n) \frac{1}{1-c} = T(n) = \Theta(f(n))$$

Parte b.  $T(n) = \Omega(f(n))$   
 $T(n) \geq f(n)$

sono tutti  $> 0$

Ricerca Seq (Ricerca L, eletta x)

```

for each y ∈ L do
    if (y = x) then return TRUE
return FALSE

```

$T_{best}(n) = O(1) = \text{Costante}$  ← TEMPO MIGLIORE (NON DIPENDE DALLA DIM. DEL DATO IN INPUT)

$T_{worst}(n) = C \cdot n = \text{costante} \cdot n = O(n)$

$T_{medio}(n) = C \cdot \frac{n}{2} = O(n)$  → bisognerebbe conoscere la distribuzione

Ci poniamo sempre nel caso peggiore, con  $T_{worst}$ .

$T_{worst}(n) = \max \text{ istanze} I \dim(n) \quad T(I)$

$T_{best}(n) = \min \text{ istanze} I \dim(n)$

$T_{avg} = \sum_{\substack{\text{istanza} \\ \text{di dim } n}} \varphi(I) \cdot T(I)$

↳ Probabilità di istanza

## ALGORITMO

1) Operazione logico-aritmetica / assegnazione  $\equiv O(1)$  [costante]

2) if (+test) then Body<sub>1</sub> else Body<sub>2</sub>

Tempo di esecuzione di un if =  $T(\text{test}) + \max \{T(B_1) + T(B_2)\}$

↳ IP peggiore tra i tre di B1 e B2

3) for i = 1 to n

Body

$$\sum_{i=1}^n T_i \longrightarrow \text{tempo corpo}$$

4) while (test) do

Body

5) repeat

Body  
until (Test)

$$\sum_{i=\emptyset}^m T_i + \sum_{i=\emptyset}^{n-1} T_i$$

6) funz (x)

7) seq di istc. I<sub>1</sub>  
I<sub>2</sub>  
I<sub>3</sub>

## Fattoriale (n)

```

ris = 1
for i = 1 to n do
    ris = ris * i
return ris

```

$$T(n) = C_1 + \sum_{i=1}^n C_2 = C_1 + C_2 n = \Theta(n)$$

PER CASA

Scrivere un algoritmo in pseudocodice che prenda la lista  $\boxed{3 \ 5 \ 4 \ 6 \ 3 \ 2 \ 9}$  e determini il massimo locale (maggiorre di precedente e successivo) LOCAL MAX

Svolgimento

LocalMax (lista L)

```

L = 3, 5, 4, 6, 3, 2, 9
while (L != 'STOP')
    if (L > L-1 & L < L+1)
        max = L;
    else
        if (L-1 > L)
            max = L-1;
        else
            max = L+1;
    return max;

```

ALGORITMI RICORSIVI

RICORRENZA

$$T(n) = \begin{cases} 1 & n=1 \\ C + T\left(\frac{n}{2}\right) & n \geq 2 \end{cases}$$

METODO DELL'ITERAZIONE

$$T(n) = C \cdot T\left(\frac{n}{2}\right)$$

$$= C + C + T\left(\frac{n}{2^2}\right) = 2C + T\left(\frac{n}{2^2}\right) = 2C + \left[C + T\left(\frac{n}{2^3}\right)\right] = 3C + T\left(\frac{n}{2^3}\right)$$

$$C \cdot \log_{2} n + 1 = \Theta(\log n)$$

$$\hookrightarrow kC + T\left(\frac{n}{2^k}\right) \rightarrow k = \log_2 n$$

$$T(n) = \begin{cases} 9T\left(\frac{n}{3}\right) + n & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases}$$

$$T(n) = 9T\left(\frac{n}{3}\right) + n$$

$$= 9 \left[ 9T\left(\frac{n}{3^2}\right) + \frac{n}{3} \right] = 9^2 T\left(\frac{n}{3^2}\right) + (3n + n)$$

$$= 9^3 \cdot T\left(\frac{n}{3^3}\right) + (3^2 n + 3n + n) = 9^3 \cdot T\left(\frac{n}{3^3}\right) + (3^2 n + 3^1 n + 3^0 n)$$

$$T(n) = q^k T\left(\frac{n}{3^k}\right) + n \sum_{i=0}^{k-1} 3^i$$

progressione geometrica

$$= q^k T\left(\frac{n}{3^k}\right) + n \frac{3^{k-1}}{2}$$

Impongo  $\frac{n}{3^k} = 1$ ,  $k = \log_3 n$

$$\begin{aligned} T(n) &= q^{\log_3 n} + n \frac{3^{\log_3 n - 1}}{2} = \\ &= 3^{\log_3 n} + n \frac{n-1}{2} = 3^{\log_3 n} + \frac{n(n-1)}{2} = n^2 + \frac{n(n-1)}{2} = \Theta(n^2) \end{aligned}$$

9.10.13

### ABBIANO VISTO

- 1) Metodo delle iterazioni
- 2) Metodo di sostituzione

### ESEMPIO

$$T(n) = \begin{cases} 1 & \text{se } n=1 \\ T\left(\lfloor \frac{n}{2} \rfloor\right) + n & \text{se } n>1 \end{cases}$$

$$T(n) = O(n)? \quad \exists c > 0 \text{ tale che } T(n) \leq c \cdot n$$

$$\begin{aligned} T(n) &= T\left(\lfloor \frac{n}{2} \rfloor\right) + n \\ &\leq c \left\lfloor \frac{n}{2} \right\rfloor + n \leq c \cdot \frac{n}{2} + n = \left(\frac{c}{2} + 1\right) n \stackrel{?}{\leq} c \cdot n \end{aligned}$$

$$\text{dove } \frac{c}{2} + 1 \leq c \rightarrow c \geq 2$$

### DIVIDE ET IMPERA

$$T(n) = \underbrace{T_D(n)}_{\text{DIVIDE}} + \sum_{i=1}^a T(h_i) + \underbrace{T_C(n)}_{\text{COMBINE}}$$

$$T(n) = \begin{cases} a T\left(\frac{n}{b}\right) + f(n) & n > 1 \\ 1 & n = 1 \end{cases}$$

$\star$

$a \geq 1$   
 $b > 1$   
 $d = \log_b a$

### TEOREMA MASTER

Sia  $T(n)$  definita come in  $\star$ , con  $a \geq 1$ ,  $b > 1$ ,  $f(n)$  asintoticamente non negativa

1)  $T(n) = \Theta(n^a)$ , se  $f(n) = O(n^{a-\epsilon})$ , per  $\epsilon > 0$

2)  $T(n) = \Theta(n^a \log n)$ , se  $f(n) = \Theta(n^a)$

3)  $T(n) = \Theta(f(n))$ , se  $\begin{cases} 3.1 & f(n) = \Omega(n^{a+\epsilon}), \epsilon > 0 \\ 3.2 & \exists c < \epsilon \text{ c.c. } af\left(\frac{n}{b}\right) \leq cf(n) \text{ per } n \text{ suff. grande} \end{cases}$

### ESEMPIO 1

$$T(n) = 3T\left(\frac{n}{2}\right) + n^2$$

Siamo nelle condizioni di usare master?

$$\begin{aligned} a &= 3 \\ b &= 2 \\ d &= \log_2 3 \\ f(n) &= n^2 \end{aligned}$$

Si

Ora devo vedere se è  $\Theta - \Theta - \Omega$

$$n^2 = \Omega(n^{\underline{\log_2 3 + \varepsilon}}) \quad \varepsilon > \emptyset \quad \rightarrow \text{verificata in 3.1}$$

$$d < \varepsilon \leq 2 - \log_2 3 \quad \text{rende vero} \quad \heartsuit$$

$$\log_2 3 + \varepsilon \leq 2$$

$$n^2 = \Omega(n^{\underline{\log_2 3 + \varepsilon}})$$

$$\exists c < 1 \quad t.c. \quad 3\left(\frac{n}{2}\right)^2 \leq c \cdot n^2 \quad ? \quad \rightarrow \text{verificata in 3.2}$$

$$3\left(\frac{n}{2}\right)^2 \leq c \cdot n^2$$

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

$$\hookrightarrow \frac{3}{4}y^2 \leq cy^2 \quad \rightarrow \frac{3}{4} \leq c < 1$$

### ESEMPIO 2

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2$$

$$\begin{aligned} a &= 4 \\ b &= 2 \\ d &= \log_2 4 = 2 \\ f(n) &= n^2 \end{aligned}$$

$$n^d = n^2 \quad \rightarrow \text{caso 2} \quad \rightarrow T(n) = \Theta(n^2 \log n)$$

### ESEMPIO 3

$$T(n) = T\left(\frac{n}{2}\right) + 2^n$$

$$\begin{aligned} a &= 1 \\ b &= 2 \\ d &= \log_2 1 = \emptyset \\ f(n) &= 2^n \end{aligned}$$

devo confrontare  $2^n$  con  $n^d$

$$\varepsilon > \emptyset. \quad f(n) = 2^n = \Omega(n^\varepsilon)$$

$$2^n = \Omega(n) \text{ se scelgo } \varepsilon = 1$$

$$\exists c < 1 \quad t.c. \quad af\left(\frac{n}{b}\right) \leq c \cdot f(n) \quad ?$$

$\hookrightarrow$  ho che  $\Omega(n^{0+1}) = \Omega(n)$   
[in def. dice  $\Omega(n^{d+\varepsilon})$ ]

$$\frac{n}{2^2} \leq c \cdot 2^n = c \cdot 2^{\frac{n}{2}} \cdot 2^{\frac{n}{2}}$$

$$\hookrightarrow 1 \leq c \cdot 2^{\frac{n}{2}}$$

$$\hookrightarrow \frac{1}{2^{\frac{n}{2}}} \leq c < 1$$

concludo che è lo caso 3

### ESEMPIO

$$T(n) = 2^n T\left(\frac{n}{2}\right) + n^2$$

$a = 2^n$   
 $b = 2$   
 $d = \log_2 2^n = n$

→ NO TEOREMA MASTER PERCHÉ A NON È COSTANTE!

### ESEMPIO

$$T(n) = 16 T\left(\frac{n}{4}\right) + n$$

$a = 16$   
 $b = 4$   
 $d = \log_4 16 = 2$

$\varepsilon > \phi$  t.c.  $n = O(n^{2-\varepsilon})$  se prendo  $\varepsilon = 1$  ottengo:  $n = O(n^1) \rightarrow n = O(n)$

primo caso,  $T(n) = \Theta(n^2)$

### ESEMPIO

$$T(n) = \frac{1}{2} T\left(\frac{n}{2}\right) + \frac{1}{2}$$

$a = \frac{1}{2} \quad \rightsquigarrow a < 1 \text{ quindi } \underline{\text{NO MASTER}}$   
 $b = 2$   
 $d = \log_2 \frac{1}{2} = -1$

### ESEMPIO

$$T(n) = 2T\left(\frac{n}{2}\right) + n \log n$$

$a = 2$   
 $b = 2$   
 $d = \log_2 2 = 1$

Dimostra che non puoi usare master perché non si verifica nessuna delle opzioni.

1)  $n \log n = O(n^{1-\varepsilon}) \quad \varepsilon > \phi ?$

$c > \phi$  per  $n$  sufficientemente grande  $n \log n \leq c \cdot n^{1-\varepsilon} = \frac{n}{n^\varepsilon} \rightarrow n^\varepsilon \log n \leq c$

2)  $n \log n = \Theta(n) \rightarrow n \log n \in \Theta(n)$

$n \log n = O(n) ?$

$\lim_{n \rightarrow \infty} \frac{n \log n}{n} = +\infty$  Quindi non è mai  $O(n)$ , quindi non vale

3)  $n \log n = \Omega(n^{1+\varepsilon})$  per  $\varepsilon > \phi$

$$\lim_{n \rightarrow \infty} \frac{n \log n}{n^{1+\varepsilon}} = \lim_{n \rightarrow \infty} \frac{n \log n}{n^{1+\varepsilon}} = \infty$$

↓ prevalse

Quindi  $n \log n = o(n^{1+\varepsilon})$ .  $o$ -piccolo e  $\Omega$ -grande sono disgiunti. Quindi non vale.

### ESERCIZI PER CASA

①  $T(n) = 2T\left(\frac{n}{4}\right) + n^{0.51}$

②  $T(n) = 3T\left(\frac{n}{3}\right) + \sqrt{n}$

③  $T(n) = 64T\left(\frac{n}{8}\right) - n^2 \log n$

# RISOLVO ESERCIZI PER CASA

$$\textcircled{1} \quad T(n) = 2T\left(\frac{n}{4}\right) + n^{0.5}$$

$$a = 2 \quad d = \log_4 2 = \frac{1}{2}$$

$$b = 4 \quad f(n) = n^{0.5}$$

$$\textcircled{2} \quad T(n) = 3T\left(\frac{n}{3}\right) + \sqrt{n}$$

$$a = 3 \quad d = \log_3 3 = 1$$

$$b = 3 \quad f(n) = \sqrt{n}$$

$$\textcircled{3} \quad T(n) = 64T\left(\frac{n}{8}\right) + n^2 \log n$$

$$a = 64 \quad d = \log_8 64 = 2$$

$$b = 8 \quad f(n) = n^2 \log n$$

C $\rightarrow$  non posso applicare il teorema master.

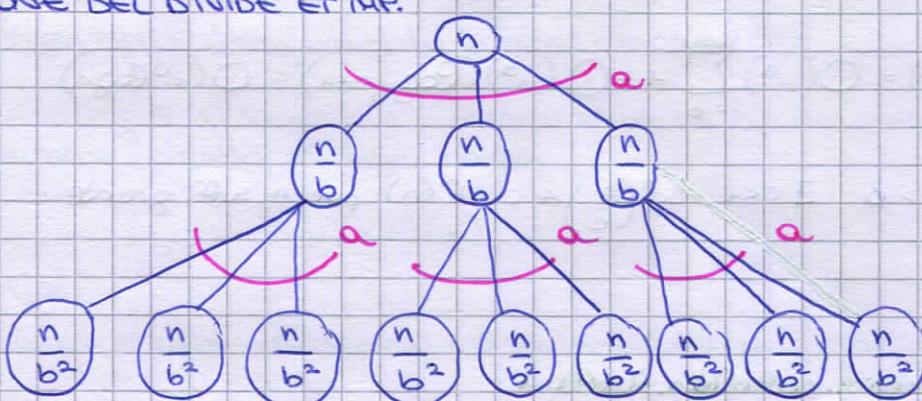
## SPIEGAZIONE DEL DIVIDE ET IMP.

$$i = \emptyset$$

$$i = 1$$

$$i = 2$$

$$10.10.13$$



1) a livello  $i$  la dimensione dei sottoproblemi è  $\frac{n}{b^i}$ ,  $i = \emptyset, 1, \dots$

2) il contributo di un nodo a livello  $i$  sul tempo di esecuzione è  $f\left(\frac{n}{b^i}\right)$

3) il numero massimo di livelli dell'albero è  $\log_b n$

4) a livello  $i$  ci sono  $a^i$  nodi

$$a^i = n^{\frac{\log_b n}{\log_b a}}$$

5) il numero di foglie è  $n^d$  ( $d = \log_b a$ )

$$V(n) = \sum_{i=0}^{\log_b n} a^i$$

$$T(n) = \sum_{i=0}^{\log_b n} a^i f\left(\frac{n}{b^i}\right)$$

## DIMOSTRAZIONE DEL TEOREMA MASTER

$$1) f(n) = O(n^{d-\varepsilon}), \varepsilon > 0 \Rightarrow T(n) = O(n^d)$$

dimostra.

$$\begin{aligned} a_i f\left(\frac{n}{b^i}\right) &= a_i O\left(\left(\frac{n}{b^i}\right)^{d-\varepsilon}\right) \\ &= O\left(a_i \left(\frac{n}{b^i}\right)^{d-\varepsilon}\right) \\ &= O\left(n^{d-\varepsilon} \cdot \frac{a_i}{(b^i)^{\varepsilon} (b^i)^{-\varepsilon}}\right) = O\left(n^{d-\varepsilon} \cdot \frac{a_i \cdot (b^\varepsilon)^i}{a_i}\right) = O\left(n^{d-\varepsilon} \cdot (b^\varepsilon)^i\right) \end{aligned}$$

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_b n} O\left(n^{d-\varepsilon} \cdot (b^\varepsilon)^i\right) = O\left(\sum_{i=0}^{\log_b n} n^{d-\varepsilon} \cdot (b^\varepsilon)^i\right) = O\left(n^{(d-\varepsilon)} \sum_{i=0}^{\log_b n} (b^\varepsilon)^i\right) = \\ &= O\left(n^{d-\varepsilon} \cdot \frac{(b^\varepsilon)^{\log_b n + 1 - 1}}{b^{\varepsilon-1}}\right) = O\left(n^{d-\varepsilon} \cdot \frac{(b^\varepsilon)^{\log_b n} \cdot b^{\varepsilon-1}}{b^{\varepsilon-1}}\right) \end{aligned}$$

$$T(n) \geq n^d \Rightarrow T(n) = \Omega(n^d)$$

$$2) T(n) = \Theta(n^d) \Rightarrow T(n) = \Theta(n^d \log n)$$

dimostra.

$$a_i f\left(\frac{n}{b^i}\right) = \Theta\left(a_i \left(\frac{n}{b^i}\right)^d\right) = \Theta\left(n^d \frac{a_i}{(b^i)^d}\right) = \Theta\left(n^d \frac{a_i}{(b^d)^i}\right) = \Theta(n^d)$$

$$T(n) = \sum_{i=0}^{\log_b n} \Theta(n^d) = \Theta(n^d) \sum_{i=0}^{\log_b n} 1 = \Theta(n^d)(\log_b n + 1) = \Theta(n^d \log n)$$

$$3) f(n) = \Omega(n^{d+\varepsilon}), \varepsilon > 0 \quad \exists c < 1 \quad a_i f\left(\frac{n}{b^i}\right) \leq c \quad f(n) \text{ per } n \text{ suff. grande}$$

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

dimostra. Parte a.

Tempo di esecuz. carica tabella questo 1

$$a_i f\left(\frac{n}{b^i}\right) \leq c \cdot f(n) < f(n)$$

$$\forall i : a_i f\left(\frac{n}{b^i}\right) \leq c f(n)$$

induzione. suppongo vero fino a(i-1)

$$a_i f\left(\frac{n}{b^i}\right) = a_i \cdot a^{i-1} \cdot f\left(\frac{n}{b^{i-1}}\right) \leq a_i \cdot c^{i-1} f\left(\frac{n}{b^{i-1}}\right) = c^{i-1} a_i f\left(\frac{n}{b^{i-1}}\right) \leq c^{i-1} \cdot c f(n) = c^i f(n)$$

$$T(n) \leq \sum_{i=0}^{\log_b n} c^i f(n) = f(n) \sum_{i=0}^{\log_b n} c^i \leq f(n) \sum_{i=0}^{\infty} c^i = f(n) \frac{1}{1-c} = T(n) = \Theta(f(n))$$

Sono tutti > 0

Parte b.  $T(n) = \Omega(f(n))$   
 $T(n) \geq f(n)$

ESEMPIO DI COLLEZIONE DI OGGETTI: dizionario (a ciascun elemento è associata una chiave di identificazione presa in un dominio totalmente ordinato)

Operazioni sulla collezione "Dizionario":

- (1) inserimento di elemento e di chiave a esso associata
- (2) cancellazione di un elemento data una chiave
- (3) ricerca dell'elemento associato a una data chiave

Tipo di dato: modello matematico che consiste in una coppia che ha un insieme di valori e quello che può fare

Tipo di dato "Dizionario": → Punteggiata

dati: un insieme  $S$  di coppie costituite dall'elemento che voglio memorizzare e dalla sua chiave (elmt, key)

Operazioni: insert (dizionario  $S$ , dim, chiave  $k$ )

delete (dizionario  $S$ , chiave  $k$ )

search (dizionario  $S$ , chiave  $k$ ) → elmt

Post: aggiunge a  $S$  una coppia (elmt, key)

Pre:  $k$  è presente in  $S$

Post: cancella da  $S$  le coppie con chiave  $k$

Post: Se la chiave  $k$  è presente in  $S$ , restituisce l'elemento ( $e, k$ ) ad essa associato, altrimenti restituisce null.

CHE COS'È UNA STRUTTURA DATI? È una particolare organizzazione delle informazioni che permette di supportare in modo efficiente le operazioni di un tipo di dato.

↳ Punteggiata

1<sup>a</sup> STRUTTURA DATI: Array ordinato in senso crescente

DATI: Un array  $S$  di dimensione  $n$  contenente record con due campi (info, key) e ordinato in senso crescente rispetto al campo chiave

$$S(n) = \Theta(n)$$

pre:  $S[p..r]$

post: restituisce l'indice dell'elemento in  $S$  che ha chiave  $k$  se esiste, -1 altrimenti.

↳ Ricorsivo

Search\_index (dizionario  $S$ , chiave  $k$ , inoltre  $p$ , int  $r$ ) → int

if ( $p > r$ )

return -1

→ vettore vuoto, quindi non è presente

med =  $(p+r)/2$

→ per trovare l'elemento centrale del vettore

if [med], Key > K

return Search\_index ( $S, k, p, med-1$ )

else

return Search\_index ( $S, k, med+1, r$ )

Search (dizionario  $S$ , chiave  $k$ )

i = Search\_index ( $S, k, 1, S.length$ )

if  $i == -1$  return NULL

else return  $S[i].info$

## COMPLESSITÀ DI SEARCH\_INDEX:

$$T(n) = \begin{cases} \Theta(1) & n \neq \emptyset \\ T(\frac{n}{2}) + \Theta(1) & n > \emptyset \end{cases}$$

$$n^{\log_2 a} = n^{\log_2 1} = n^0 = 1 \quad \text{→ TEOREMA MASTER}$$

$f(n) = 1$  Stesso ordine di grandezza, caso 2 teorema master

$$\text{Complessità } \Theta(n^{\log_2 1} \log n) = \Theta(1 \cdot \log n) = \Theta(\log n)$$

## ESERCIZIO 2 - OPERAZIONE 1

insert(dizionario S, elem e, key k)

reallocate(S, S.length+1)  $\Theta(n)$  Ha costo n

i = 1

while i < S.length and S[i].key < k

do i = i + 1

for j = S.length downto i + 1

do S[j] = S[j - 1]

S[i].info = e

S[i].key = k

vengono eseguiti i volte

vengono eseguiti n-i volte

$\Theta(n)$  complessità lineare

## ESERCIZIO 3 - OPERAZIONE 2

delete(dizionario S, chiave k)

i = Search\_Index(S, k, 1, S.length)  $\Theta(\log(n))$

for j = i to S.length - 1

S[j] = S[j + 1]

reallocate(S, S.length - 1)  $\Theta(n)$

$n-i-1$  → al caso peggiore  $\Theta(n)$

LA SEARCH È QUELLA CHE FUNZIONA MEGLIO, PERCIÒ QUESTO SARÀ UTILE QUANDO CI SONO MOLTE SEARCH.

Tecnica raddoppio - dimensionamento array dimensione h dove memorizza n elementi soddisfa la seguente invariante

$$h \leq 4n$$

Inizialmente, quando  $n=0$ , poniamo  $h=1$ . Ogni volta che  $n$  supera  $h$ , l'array viene riallocato raddoppiando la dimensione

$$h = 2h$$

Cognitivamente  $n$  scende di  $h/2$ , l'array viene riallocato dimenticando la dimensione.

$$h \leftarrow h/2$$

$$S(h) = \Theta(h)$$

$n$  inserimenti costano

$$n = h \rightarrow 2h$$

tempo ammortizzato dato dal tempo totale richiesto dall'algoritmo nel caso pessimo per tutte le K-operazioni su istanze di dimensione  $n$

Esercizio. Realizzare le tipi del dizionario utilizzando vettore ordinato + tecnica del rdimensionamento.

I implementazione, lista doppia

Atti:  
 Una collezione L di n record contenenti (info, key, next, prev) dove next e prev sono puntatori al successivo e al precedente record della collezione.  
 L.head punta al primo elemento della lista.  
 L.head = NULL è la lista vuota

Operazioni:

insert (dizionario, elem e, chiave k)

Creare un nuovo record

p.info = e

p.key = k

p.next = L.head

if L.head ≠ NULL

L.head.prev = p

L.head = p

p.prev = NULL

delete (dizionario L, chiave k)

X = L.head

while X.key ≠ k

X = X.next

if X.next ≠ NULL

X.next.prev = X.prev

if X.prev ≠ NULL

X.prev.next = X.next

else

L.head = X.next

Rimuovi X

assegnamento = costante

}  $\Theta(n)$  nel caso pessimo

```

search (dizionario L, chiave k)
    x = L.head
    while x ≠ NULL AND x.key ≠ k
        x = x.next
    if x ≠ NULL
        return x.info
    else
        return NULL

```

## TECNICHE PER RAPPRESENTARE COLLEZIONI

### 1) Basata su strutture indirizzate (array)

Ipotesi: in un array di dimensione  $n$  gli indici possono essere compresi tra  $0 \dots n-1$  (in C) oppure tra  $1 \dots n$  (pseudo codice)

È possibile accedere in lettura e scrittura ad una qualsiasi cella in tempo costante

#### PROPRIETÀ BASILARE ARRAY

- (1) (forte) gli indici delle celle di un array sono numeri consecutivi
- (2) (debole) non è possibile aggiungere nuove celle ad un array

### 2) Basata su strutture collegate

#### PROPRIETÀ BASILARI

- (1) (forte) è sempre possibile aggiungere e togliere record da una struttura collegata
- (2) (debole) gli indirizzi dei record di una struttura collegata non sono necessariamente consecutivi (fondamentali per buone prestazioni)

## ESERCIZI SUL CALCOLO DELLA COMPLESSITÀ

MyAlgorithm (int n) → int

```

int a, i, j
if (n > 1) then
    a = 0
    for i = 1 to n-1
        for j = 1 to n-i
            a = a + (i+j) * (j+n)
        end for
    end for
    for i = 1 to 16
        a = a + MyAlgorithm (n/4)
    end for
    return a
else
    return n-1
end if

```

} costo:  $\Theta(n^2)$

$(n-1)(n-1) \rightarrow$  numero di volte che viene ripetuto

—> si ripete 16 volte  
 $\rightarrow T(n/4) \rightarrow T(n/4)$

## DETERMINARE IL COSTO COMPUTAZIONALE $T(n)$ DELL'ALGORITMO IN FUNZIONE DEL PARAMETRO $n \geq 0$

$$T(n) = \begin{cases} \Theta(1) & n \leq 1 \\ 16T(n/4) + c \cdot n^2 & n > 1 \end{cases}$$

$\Theta(n^2) \quad n \geq 0$  Teorema master

$$n^{\log_4 16} = n^{\log_4 16} = n^2$$

$$f(n) = c \cdot n^2 \rightarrow \text{caso 2, complessità } \Theta(n^2 \log n)$$

## ESERCIZIO

```

A (int n)
S = φ
for i=1 to n
    do S = S + B(i)
return S
B (int m)
S = φ
for j=1 to n
    do S = S + j
return S
  
```

$$\begin{aligned}
 T_A(n) &= \sum_{i=1}^n c \cdot i = c \sum_{i=1}^n i = \frac{n(n+1)}{2} = \Theta(n^2) \\
 O(n) &\rightarrow S = \sum_{k=0}^n k = \underline{\underline{\Theta(n^2)}} \\
 \rightarrow B(m) &= m \text{ funzione identità} \\
 \Theta(n) &
 \end{aligned}$$

## ESERCIZIO

```

Fun (A, n)
if n<3 return 4
t = Fun (A, n/2) → T(n/2)
if t > A[n]
    t = t + Fun (A, n/2) ↗ caso peggiore, va tenuto conto
for i=1 to n
    t = t + A[i] + Proc(n)
return t
  
```

$$T_{\text{fun}}(n) = T(n/2) + T(n/2) + \Theta(n\sqrt{n})$$

$$\left. \begin{array}{l} \\ \\ \end{array} \right\} n\sqrt{n}$$

Nell'ipotesi che  $\text{Proc}(m) = \Theta(\sqrt{m})$ , determinare la complessità asintotica (al caso peggiore) della funzione  $\text{Fun}(A, n)$  al crescere di  $n \in \mathbb{N}$

24.10.13

Si vuole e' ordine di grandezza della complessità della seg. funzione al crescere di  $n \in \mathbb{N}$

```

foo (int n)
if n≤5
    return 7
else
    if n≥75
        for i=1 to  $\lfloor n/2 \rfloor$ 
            k =  $\lfloor n/2 \rfloor$ 
        return k * foo(k) + foo( $n/2$ )
  
```

$$T(n) = 2T(n/2) + n/2$$

$$(\Rightarrow T(n) = T(n/2) + T(n/2) + n/2)$$

$$(\Rightarrow T(n) = 2T(n/4) + \Theta(n))$$

$$n^{\log_b a} = n^{\log_b 2} = n^{1/2} = \sqrt{n}$$

$$f(n) = n$$

Siamo nel caso 3?

$$f(n) = \Omega(n^{\log_b a + \epsilon}) \quad \text{con } \epsilon > 0$$

$$= \Omega(n^{1/2 + \epsilon}) \quad \text{fisso } \epsilon = 1/2$$

Verifico la condizione di regolarità:  $a f(n/b) \leq c f(n)$  con  $c < 1$

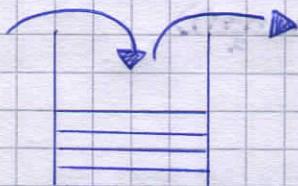
$$2 \frac{n}{\sqrt{n}} \leq c n \rightarrow c \leq \frac{1}{2}$$

pongo  $c = \frac{1}{2}$  → condiz. di regolarità soddisf.

La complessità è  $\Theta(n)$

## Tipi di dati: Pila

Una pila è una sequenza di elementi di un certo tipo in cui è possibile aggiungere o togliere elementi solo da un estremo (il top) della pila



### Tipi di dato:

Dati: una sequenza S di n elementi

operazioni: initstack()  $\rightarrow$  stack

Post: restituisce una stack vuota

op: stack-empty (stack s)  $\rightarrow$  bool

Post: restituisce true se s è vuota, false altrimenti

op: push (stack s, elem e)

Post: aggiunge e come ultimo elemento di s

op: pop (stack s)  $\rightarrow$  elem

Pre: s è non vuota

Post: toglie da s l'ultimo elemento e lo restituisce

op: top (stack s)  $\rightarrow$  elem

Pre: s è non vuota

Post: restituisce l'ultimo elemento di s

## PILA BASATA SUGLI ARRAY

Dati: uno stack s è un vettore di dimensione n  $s[1 \dots n]$  e' manteniamo un attributo s.top che è l'indice dell'ultimo elemento inserito.

Lo stack è vuoto se s.top =  $\emptyset$

```

init_stack()
  S = allocare (n)
  S.top =  $\emptyset$ 
  return S

stack-empty (stack s)
  return s.top ==  $\emptyset$ 

push (stack s, elem e)
  S.top = S.top + 1
  S[S.top] = e

pop (stack s)
  S.top = S.top - 1
  return S[S.top + 1]

top (stack s)
  return S[S.top]

```

QUESTE FUNZIONI  
HANNO COMPLESSITÀ  
COSTANTE

## Tipo di dato: Coda

Una coda è una sequenza di elementi di un certo tipo in cui è possibile aggiungere elementi ad un estremo (la coda) e togliere elementi dall'altro (la testa) FIFO

### Tipi di dato:

Dati: una sequenza Q di n elementi

op: initqueue ()  $\rightarrow$  Queue

Post: restituisce una coda vuota

op: queue-empty (Queue q)  $\rightarrow$  bool

Post: restituisce true se q è vuota, false altrimenti

op: enqueue (Queue q, elem e)

Post: aggiunge e come ultimo elemento di q

op: dequeue (Queue q)  $\rightarrow$  elem

Pre: q non è vuota

Post: toglie da q il primo elemento e lo restituisce

op: first (Queue q) -> elem

Pre: q è non vuota

Post: restituisce il primo elemento in q

## RAPPRESENTAZIONE VETTORE CIRCOLARE



Q è un vettore di dim n con 2 attributi

- (1) Q.head è l'indice della posizione da cui estrarre un elemento
- (2) Q.tail da cui inserire un elemento

Il successore dell'ultimo elemento (n) è la prima posizione

$$Q.\text{head} = Q.\text{tail} = 2$$

In ogni istante, gli elementi della coda si trovano nel segmento  $Q.\text{head}, Q.\text{head}+1, \dots, Q.\text{tail}-1$

In ogni istante, si garantisce che la coda abbia capacità massima di  $n-1$  elementi.

coda vuota  $\Rightarrow Q.\text{head} == Q.\text{tail}$

coda piena  $\Rightarrow Q.\text{head} = (Q.\text{tail} \bmod n) + 1$

enqueue()

Q = allocate(n)

Q.tail = 1

Q.head = -1

return Q

COMPLESSITÀ COSTANTE ( $\Theta(1)$ )

queue\_empty (Queue Q)

x = Q[Q.head]

if Q.head = Q.tail

Q.head = 1

else

Q.head = Q.head + 1

return x

first (Queue Q)

return Q[Q.head]

## REALIZZARE UNA CODA Q UTILIZZANDO DUE FILE p1 e p2

enqueue (Queue Q, elem e)

push (Q.p1, e)

dequeue (Queue Q)

if stack\_empty (q.p2)

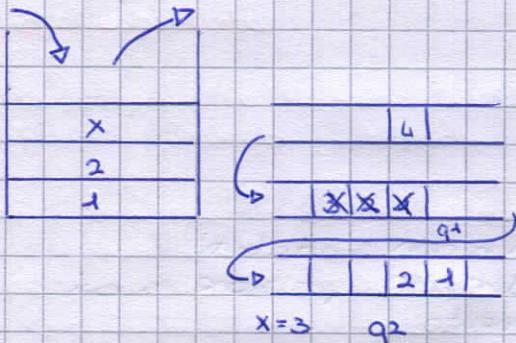
while not stack\_empty . p2

x = pop (q.p2)

push (q.p2, x)

return pop (q.p2)

IDEA



N.B! Tempo ammortizzato:  
diviso per il n° di operazioni.

### ESERCIZIO DELL'ALTRA VOLTA

push (Stack s, Element x)

enqueue (s.q1, x)

pop (Stack s)

x = dequeue (s.q1)

while not queue-empty (s.q1) }

enqueue (s.q2, x)

x = dequeue (s.q1)

}  $\Theta(n)$  nel caso peggiore

↳ risulta molto costosa perché per estrarre  
devo ribaltare tutto

scambio (s.q1, s.q2)

:

return x

→ Scambia i contenuti delle  
due pile usando i puntatori

### INSERISCI I NUMERI DA 1 A 5 IN UNA PILA E POI STAMPALLA

```
*include <stdio.h>
*include <stdlib.h>

/* dichiara le type pile */
typedef struct nodo{
    int info;
    struct nodo *next;
} nodo;
typedef nodo * List;

struct stack {
    List contents;
    int size;
}

typedef struct stack * stack;

/* definisce le funzioni */
Stack initstack (){
    stack s;
    s = (stack) malloc (sizeof (struct stack));
    s->contents = NULL;
    s->size = 0;
    return s;
}

int stack_empty (stack s){ ... }

void push (stack s, int elem){ ... }

int pop (stack s){ ... }

int top (stack s){ ... }

int size (stack s){ ... }
```



```

int main () {
    stack s;
    int i;
    s = initstack();
    for (i=1, i <= 5, i++)
        push (s, i);
    while (!stackempty(s))
        printf ("%d\n", pop(s));
    return EXIT_SUCCESS;
}

```

### PROGRAMMA CUENTE

Aggiungi `#include "stack.h"` così da includere l'interfaccia.

File `usePila.c`

### SVANTAGGI:

- (1) Se ho un errore devo ricompilare
- (2) push / pop legate e non esportabili
- (3) accesso elementi che non sono gestito
- (4) per utilizzare un array al posto di una lista devo cambiare il programma

### SOLUZIONE: uso più file

#### PROGRAMMA CUENTE: usa le tipi di dato

IMPLEMENTAZIONE: definizione tipo di dato + realizzazione operazioni sul tipo di dato

INTERFACCIA: [rappresenta i prototipi che possiamo usare] insieme dei prototipi delle funzioni che operano sul tipo di dato. `stack.h`

`stack.h` FILE

```

typedef struct stack* stack;
stack initstack();
int stack_empty (stack s);
void push (stack s, int elem);
int pop (stack s);
int top (stack s);
int size (stack s);

```

ben definita perché è un indirizzo

} Esempio di dichiarazioni in cui ho un nome, stack, che non so cosa sia. Quindi ASSUNGO LA DEFINIZIONE DEL TIPO STACK (CHE È NEL FILE stack.h)

### Implementazione - Imp. P. lista.c

`#include "stack.h"`



### item.h

```

typedef struct item * Item;
Item oggi();
void stampa (Item x);
int compare (Item x, Item y)
/* ImpItemInt.c */
#include <stdlib.h>
#include <stdio.h>
#include "item.h"
struct item {
    int info;
};
/* usePila.c */
#include "item.h"
#include "stack.h"

```

`gcc -c modolo.c -> restituisce file oggetto.o  
gcc -c impPilaLista.c ] 2 file oggetto  
gcc -c usePila.c`

↳ gcc -c impPilaLista.c  
usePila.c

`gcc impPilaLista.o usePila.o -o output`

↳ crea un eseguibile output

Una lista è una sequenza di elementi di un certo tipo in cui è possibile aggiungere o togliere elementi in posizioni arbitrarie.

dati: una sequenza  $\ell$  di elementi

op:  $\text{createList}(\ell) \rightarrow \text{list}$

Post: restituisce una lista vuota

op:  $\text{isEmpty}(\ell) \rightarrow \text{Bool}$

Post: restituisce true se  $\ell$  è vuota, false altrimenti

op:  $\text{search}(\ell, k) \rightarrow \text{Node}$

Post: restituisce il primo elemento con chiave  $k$ , se esiste, altrimenti null.

op:  $\text{insert}(\ell, k, x) \rightarrow \text{list}$

Post: inserisce un nodo  $x$  (la chiave key è già stata impostata) nella lista

op:  $\text{delete}(\ell, k) \rightarrow \text{list}$

Pre:  $x \in \ell$

Post: cancella il nodo  $x$  dalla lista  $\ell$

op:  $\text{size}(\ell) \rightarrow \text{int}$

Post: restituisce il numero di elementi della lista  $\ell$

Realizzo la lista con strutture collegate

### 1) Lista semplice



Un nodo della lista ha i seguenti attributi:

$x.\text{key}$  contiene la chiave (l'info che vogliamo memorizzare)

$x.\text{next}$  punta al successore di  $x$  nella nostra lista

$L.\text{head}$  denota la testa della lista

$L.\text{head} = \text{null}$  lista vuota

### 2) Lista doppia



N.B. elemento = nodo

$x.\text{key}$

$x.\text{next}$  punta al successore

$x.\text{prev}$  punta al predecessore di  $x$  nella lista

$L.\text{head}$

Una lista può anche avere:

1) un puntatore all'ultimo elemento



2) può essere circolare



3) può avere un nodo sentinella per meglio gestire i casi limite (inserimento in testa e in coda)



I dati memorizzati possono

- essere ordinati oppure non ordinati

- ammettere duplicati o tutti i valori delle chiavi sono distinti

ESERCIZIO : Ho ora questa semplice e voglio determinare se è circolare oppure no

int circolare (List e)

```
    primo = e;
    if (e == NULL || e->next == NULL)
        return 0;
    fineLista = 0;
    e = e->next;
    while & fineLista AND primo != e
        if e->next == NULL
            fineLista = 1
        else
            e = e->next;
    return primo == e;
```

ESERCIZIO : Voglio scorrere una sola volta una lista e dividerla a metà

L<sup>d</sup> uso due puntatori, uno a velocità snella e uno a velocità doppia, ovvero uno che scorre ogni elemento e uno che ne salta unoogni volta

typedef struct node{

```
    int key;
    struct node *next;
```

} node;

typedef Node \*list;

void split (list e, list \*e1, list \*e2) {

```
    list primo, secondo;
```

```
    primo = e;
```

```
    secondo = NULL;
```

```
    if (!e || !e->next)
```

```
        *e1 = e;
```

```
        *e2 = segundo;
```

```
    else {
```

```
        secondo = primo->next;
```

```
        while (secondo != NULL){
```

```
            if (secondo->next == NULL)
```

```
                secondo = secondo->next;
```

```
            else
```

```
                primo = primo->next;
```

```
                secondo = secondo->next->next;
```

```
}
```

```
*e2 = primo->next;
```

```
primo->next = NULL; → così DIVIDO LE 2 LISTE
```

```
*e1 = e;
```

COMPLESSITÀ:  $O(n)$



}

INVARIANTE = asserzione vera prima, dopo e ad ogni iterazione del ciclo.

Per dimostrare che è invariante, dimostro che:

(1) Inizializzazione: è vera prima della prima iterazione del ciclo

(2) Conservazione: se è vera prima di un'iterazione del ciclo, rimane vera prima della successiva  
Inv ∧ Guardia → Inv [dopo l'esec. del corpo del ciclo]

(3) Condizione: quando il ciclo termina l'invariante fornisce un'altra proprietà che il rapporto è corretto

Inv ∧ Guardia → asserzione finale

## Liste doppie

search(list L, item k)

x = L.head

while x ≠ NULL AND x.key ≠ k

x = x.next

return x

funzione di terminazione: funzione a valori naturali che decrese strettamente ad ogni iterazione dell'ut

$f(n) = \# \text{ elementi della lista}$   
non ancora visitati

Inv = gli elementi da L.head a x non compreso sono diversi da k

## INIZIAZZAZIONE

Inv = gli elementi compresi tra x e x escluso hanno valori diversi da k

Inv [L.head]

non ci sono elementi  $\rightarrow$  proprietà vacuamente vera

## CONSERVAZIONE

Inv  $\wedge$  x ≠ NULL  $\wedge$  x.key ≠ k

Inv [x.next / x]

Ipotesi

(1) Inv = gli ee da L.head .... ]  $\star$

(2) x ≠ NULL  $\wedge$  x.key ≠ k

Dico dimostrare

Inv [x.next / x] =  $\star$

RIPRESA DALLA VOLTA SCORSA  


search (List L, Element k)

```

x = L.head
while x ≠ NULL AND x.key ≠ k
    x = x.next
return x
  
```

Inv = gli elementi da L.head a x non compreso hanno chiavi ≠ k

- (1) inizializzazioni
- (2) conservazione
- (3) conclusione

Inv ∧ Guardia → Inv (dopo l'esecuzione del corpo del ciclo)  
 Inv ∧ ¬Guardia → (asserzione finale)

N.B! Il ciclo termina per due ragioni:

- (1)  $x = \text{NULL}$  in tal caso l'invariante assicura che k non è presente in tutta la lista
- (2)  $x.key = k$  l'invariante assicura che k non è presente prima di x dunque x è la prima occorrenza dell'elemento con chiave k



insert (List L, Node x)

```

x.next = L.head
if L.head ≠ NULL
    L.head.prev = x
L.head = x
x.prev = NULL
  
```

delete (List L, Node x)

```

/* Pre: x ∈ L */
if x.prev ≠ NULL
    x.prev.next = x.next
else
    L.head = x.next
if x.next ≠ NULL
    x.next.prev = x.prev
  
```

rimuovi x

→ Distingue tre casi

- 1) è testa
- 2) è coda
- 3) è in mezzo

## REALIZZAZIONE CON LISTE DOBBIE E SENTINELLA

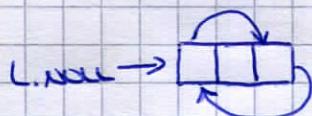
Sentinella: oggetto fittizio che ha tutti i campi degli altri elementi della lista

L.NULL rappresenta NULL

L.head è eliminata e sostituita con L.NULL.next, e L.NULL.prev punta alla coda

Una lista vuota è formata solo dalla sentinella.

La sentinella rende il codice più compatto.



```

search (List L, Element k)
    x = L.NULL.next
    while x != L.NULL AND x.key != k
        x = x.next
    return x

```

$L.\text{NULL}.\text{key} = k$



Semplifica una delle condizioni  
con l'elemento fittizio  $L.\text{NULL}$   
che esiste sempre

```

insert (List L, Node x)
    x.next = L.NULL.next
    L.NULL.next.prev = x
    L.NULL.next = x
    x.prev = L.NULL

```

```

delete (List L, Node x)
    x.prev.next = x.next
    x.next.prev = x.prev
remove (x)

```

```

struct node {
    int info;
    struct node *next;
    struct node *prev;
}
typedef struct node *List;

```

```

List createList () {
    List e;
    e = malloc (sizeof (struct node));
    e->next = e;
    e->prev = e;
    return e;
}

```

UNION per gli insiemi dinamici richiede due insiemi DISGIUNTI  $s_1$  e  $s_2$  come input  
restituiscere un insieme  $s = s_1 \cup s_2$ , che è formato da tutti gli elementi di  $s_1$  e  $s_2$

↳ COSTANTE, usa un puntatore alla testa e un puntatore  
alla coda (ex.)

```

Union (List s1, List s2)
    if s1.head == NULL
        return s2
    else
        if s2.head == NULL
            return s1
        else
            s1.tail.next = s2.head
            s2.tail = s1.tail

```

Ho concatenato le due liste



Rappresentazione con più array

indice non presente nel nostro vettore

next	0	0	3	6	8	4
key		13	12	11		
prev	0	6	8			

1 2 3 4 5 6 7 8



La costante NULL viene rappresentata con l'indice  $\emptyset$  (per es pseudocodice) - 1 (in C).

freeList  $\rightarrow$  tutte le parti vuote

free+list concatenate singolarmente in cui mantengo gli oggetti liberi. È una variabile globale.

Alocate - Object()

if free = NULL

errore "spazio terminato"

else

x = free

free = next [x]

return x

Free - Object(x)

next [x] = free

free = x

$\Theta(1)$  perché sto facendo degli assegnamenti

Albero (RADICATO) è una coppia  $T = (N, A)$ .  $N$  è un insieme finito di nodi fra cui si distingue un nodo  $r$  detto RADICE

$A \subseteq N \times N$  è un insieme di coppie di nodi, dette ARCHI

In un albero, ogni nodo  $v$  (eccetto la radice) ha esattamente un genitore (o padre) in  $A$  detto  $(u, v) \in A$

Albero  $\Rightarrow$  GRAFO CONNESSO ACQUICO <sup>NON</sup> ORIENTATO

Un nodo  $v$  può avere 0 o più figli in  $A$  detto  $(v, u) \in A$

il numero di figli di un nodo è detto grado di un nodo. Un nodo senza figli è detto foglia e un nodo non foglia (cioè che ha dei figli) è un nodo interno.

Se due nodi hanno lo stesso padre sono fratelli.

Cammino da un ~~qualsiasi~~ nodo  $u$  a un nodo  $u'$  in  $T$  è una sequenza di nodi  $\langle n_0, n_1, \dots, n_k \rangle$  tali che soddisfa le seg. proprietà

- 1)  $n_0 = u$
- 2)  $n_k = u'$
- 3)  $\langle n_{i-1}, n_i \rangle \in A$  per  $i = 1, \dots, k$

La lunghezza di un cammino è il numero di archi oppure il numero di nodi che formano il cammino - 1

Sia  $x$  un nodo in un albero radicato  $T$  con radice  $r$ . Un qualsiasi  $y$  in un cammino da  $r$  a  $x$  è detto antenato di  $x$ . [anche lo nodo stesso è antenato]

Se  $y$  è antenato di  $x$ , allora  $x$  è discendente di  $y$ .

N.B. Ogni nodo è antenato e discendente di se stesso.

Se  $y$  è un antenato di  $x$  e  $x \neq y$  allora  $y$  è un antenato proprio di  $x$  e  $x$  è un discendente proprio di  $y$ .

Le sottoalberi con radice in  $x$  è l'albero indotto dai discendenti di  $x$  con radice in  $x$

La PROFONDITÀ di un nodo  $x$  è la lunghezza del cammino dalla radice a  $x$

Un livello di un albero è costituito da tutti i nodi che stanno alla stessa profondità

L'altezza di un nodo  $x$  è la lunghezza del più lungo cammino che scende da  $x$  a una foglia

L'altezza di un albero è l'altezza della sua radice —  $\rightarrow$  Profondità massima di un nodo qualsiasi dell'albero

Albero binario  $\rightarrow$  di natura ricorsiva

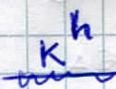
- un albero vuoto è un albero binario
- un albero costituito da un nodo radice, @ un albero binario detto sottoalbero sinistro della radice, e @ da un albero ancora binario detto sottoalbero destro della radice, è un albero binario

Un albero  $k$ -ario è un albero in cui i figli di un nodo sono etichettati con interi positivi distinti e le etichette maggiori di  $k$  sono assenti.

Un albero binario è un albero  $k$ -ario con  $k=2$ .

Un albero  $k$ -ario completo è un albero  $k$ -ario in cui tutte le foglie hanno la stessa profondità e tutti i nodi interni hanno grado  $k$ .

ESERCIZIO: Trovare il numero di foglie e di nodi interni di un albero  $k$ -ario completo di altezza  $h$ .



Perché la radice ha  $k$  figli che hanno  $k$  figli tutto per  $h$  volte

$$\# \text{foglie}(h) = k^h \quad h = \phi$$

$k^0 = 1$  l'albero è costituito dalla sola radice che è l'unica foglia

Assumiamo che per un albero di altezza  $h$  sia vero  $\# \text{foglie}(h) = k^h$  e lo dimostri per ~~K~~ altri alberi di altezza  $h+1$ .

Il numero di nodi di profondità  $h$  sono  $k^h$  per ip. induttiva in quanto foglie di un albero di altezza  $h$ .

Poiché l'albero è completo, ognuno di questi nodi ha esattamente  $k$  figli

$$k^h \cdot k = k^{h+1} \quad \text{numero di foglie} \quad \text{C.V.D}$$

$$\# \text{foglie}(h+1) = k^{h+1}$$

- Numero nodi interni  $h$

$$\sum_{i=0}^{h-1} k^i = \frac{k^{h+1}-1}{k-1} = \frac{k^h-1}{k-1}$$

ESERCIZIO: Trovare l'altezza di un albero  $k$ -ario completo con  $n$  foglie  
Le foglie di un albero completo di altezza  $h$

$$n = k^h \quad h = \log_k n$$

$$n = \frac{k^{h+1}-1}{k-1} \quad \Rightarrow \text{così posso ricavare } h$$

Tipo: Albero

dati: un insieme di nodi (di tipo Node) e un insieme di archi

operaz:

newtree()

Post: restituisce un albero vuoto

treeEmpty(Tree t) → Bool

Post: restituisce true se è vuoto, false altrimenti

gradgrado(Tree t, Node v) → int

Pre: v in t

Post: restituisce il numero di figli del nodo v

numNodi(Tree t) → int

Post: restituisce il numero di nodi dell'albero in t

padre(Tree t, Node v)  $\rightarrow$  Node

Pre: v in t

Post: restituisce il padre del nodo v nell'albero t oppure NULL se v è la radice

figli(Tree t, Node v)  $\rightarrow$  List of Node

Pre: v in t

Post: restituisce una lista contenente i figli di v

aggiungiNodo(Tree t, Node u, Element k)  $\rightarrow$  Node

Pre: se il nodo u  $\neq$  NULL allora u in t, altrimenti t è vuoto

Post: inserisce un nuovo nodo v con chiave k come figlio di u se u  $\neq$  NULL, altrimenti v è la radice dell'albero e restituisce v

aggiungiSottoAlbero(Tree t, Node u, Tree tu)

Pre: u in t

Post: inserisce nell'albero il sottoalbero tu in modo che la radice di tu diventi

figlio di u

rimuoviSottoAlbero(Tree t, Node u)  $\rightarrow$  Tree

Pre: u in t

Post: cancella e restituisce quattro albero radicati in v. L'operazione cancella dell'albero t è nodo v e tutti i suoi discendenti

```

figli (Tree P, Node v)
e = creaListA()
iter = v.left-child
while iter != null
    inserisci iter in e
    iter = iter.right-sibling
return e

```

$$T(n) = \Theta(\text{grado}(v))$$

14.11.13

## ALGORITMI DI VISITA DEGLI ALBERI

```

visitaGenerica (Node r)
S = {r}
while S ≠ ∅
    estrai un nodo u da S
    visita il nodo u
    S = S ∪ {figli di u}

```

PROTOTIPO DI ALGORITMO DI VISITA.  
Con "visita" si intende "esegue qualche operazione"

Ogni volta estraggo un nodo e prendo i figli; S rappresenta i nodi non ancora visitati.

**TEOREMA.** L'algoritmo di visita applicato alla radice di un albero con n nodi termina in  $O(n)$  iterazioni. *fowza: lo spazio usato è  $O(n)$*

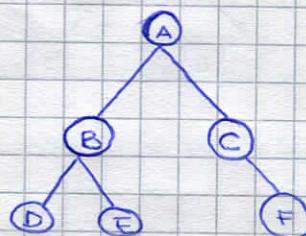
DIMOSTRAZIONE.

Ipotesi: estrazioni e inserimento avvengono in tempo costante [ $\Theta(1)$ ]

Ogni nodo verrà inserito ed estratto da S UNA VOLTA perché in un albero non si può tornare da un nodo a partire dai suoi figli procedendo di figlio in figlio.

Quindi le iterazioni del ciclo while saranno al più  $O(n)$  [ $\Rightarrow$  quanti sono i nodi dell'albero]

Poiché ogni nodo compare al più una volta in S, lo spazio richiesto è proprio  $O(n)$ .



Un algoritmo che si basa sulla struttura descritta ha costo lineare.

### TIPI DI VISITA

(1) DFS (Depth-First Search): Scendo nell'albero finché non ho visitato tutto il sottosegno sinistro e poi faccio lo stesso col destro.

```

visitaGenerica(Node r)
Stack S
push(S, r)
while not stack_empty(S)
    u = pop(S)
    if u ≠ null
        visita nodo u
        push(S, u.right)
        push(S, u.left)

```

Uso una pila. Inserisco prima il destro, poi il sinistro perché, dato che inserisco e tolgo, se facessi il contrario non riuscirei a visitare l'albero in profondità.



ORDINE DI VISITA:  
ABDECF

## VERSIONE RICORSIVA DI DFS

```

VisitaDFS (Node u)
if u ≠ null
    visita il nodo u
    VisitaDFS (u.left)
    VisitaDFS (u.right)

```

**TEOREMA.** Se  $x$  è la radice di un sottoalbero di  $n$  nodi, la chiamata di  $\text{VisitaDFS}(x)$  richiede al tempo  $\Theta(n)$

### DIMOSTRAZIONE

Per dimostrare che qualcosa è  $\Theta(n)$ , devo dimostrare che è sia  $\Omega(n)$  sia  $O(n)$ .

(1) Visto che DEVE visitare tutti i nodi del sottoalbero radicato in  $x$

$$T(n) = \Omega(n) \quad [\text{limite inferiore}]$$

(2)

$$\textcircled{*} \quad T(n) = \begin{cases} c > \phi & n = \phi \\ T(k) + T(n-k-1) + d & n > \phi \end{cases}$$

con  $d$  costante che rappresenta tutto quello che l'algoritmo fa eccetto la chiamata ricorsiva.

Si suppone che la funzione  $\text{VisitaDFS}$  sia chiamata per un nodo  $x$  per il quale il sottoalbero sinistro ha  $k$  nodi e il sottoalbero destro ha  $(n-k-1)$  nodi.

Per dimostrare che  $T(n) = O(n)$  non posso usare il teorema master: uso il metodo di sostituzione.

$$T(n) \leq (c+d)n + c$$

→ devo dimostrare che questa funz. è limite superiormente  $T(n)$

dimostra per induzione su  $n$ . Caso base:  $n = \phi$

$$T(\phi) \leq (\phi + d)\phi + c$$

$$c \leq T(\phi) \leq \phi + c$$

$$c \leq T(\phi) \leq c$$

$$c \leq c \leq c ? \quad \text{sì}$$

Abbiamo detto in  $\textcircled{*}$  che  $T(\phi) = c$

Utilizzo l'induzione completa. Per  $n > \phi$

$$T(n) \leq T(k) + T(n-k-1) + d \quad \text{per } \textcircled{*} \quad \text{con } k < n$$

Per ipotesi induttiva

$$T(n) \leq ((c+d)k + c) + ((c+d)(n-k-1) + c) + d$$

$$\hookrightarrow (c+d)(k+n-k-1) + 2c + d =$$

$$= (c+d)n - c - d + 2c + d =$$

$$= (c+d)n + c$$

#OK

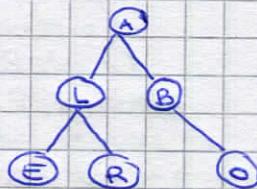
IN BASE ALL'OPERAZIONE DI VISITA DEL NODO SI OTTIENE:

(1) **VISITA IN PREORDINE**: si visita prima la radice poi si effettua le chiamate ricorsive sui figli sx e dx

(2) **VISITA SIMMETRICA**: si effettua prima la chiamata ricorsiva sul figlio sx, poi si visita la radice, poi si effettua la chiamata ricorsiva sul figlio destro

(3) **VISITA IN POSTORDINE**: si effettuano prima le chiamate ricorsive sui figli sx e dx e infine si visita la radice

## ESEMPIO.



- (1) Preordine: ALERBO
- (2) Simmetrico: ELRABO
- (3) Postordine: ELOBA

## VISITA IN AMPIEZZA (BREATH-FIRST SEARCH, BFS)

Visita BFS (Node u)

```

Queue Q
enqueue(Q, u)
while not queue_empty(Q)
  u = dequeue(Q)
  if u ≠ NULL
    visita ie nodo u
    enqueue(Q, u.left)
    enqueue(Q, u.right)
  
```

Utilizzo la coda perché mi permette di mantenere i livelli senza perdere.

## VISITA PER LIVELLI: ALBERO

### CALCOLO DELL'ALTEZZA DI UN ALBERO

height (Node u)

```

if u == NULL
  return -1
else
  return 1 + max(height(u.left), height(u.right))
  
```

→ complessità  $T(n) = \begin{cases} C & n=0 \\ T(k) + T(n-k-1) + d & n>0 \end{cases}$   
che è quello di prima (caso  $\Theta(n)$ )

## GENERALIZZAZIONE

Albero generale  
left-child      right-sibling

heightgen (Node u)

```

if u == NULL
  return -1
else
  return max{1 + heightgen(u.left-child), heightgen(u.right-sibling)}
  
```

Stessa complessità di prima,  $\Theta(n)$

↳ fratelli allo stesso altezza

Un albero binario è bifasciato se  $h = O(\log n)$  dove  $n = \text{nº nodi dell'albero}$

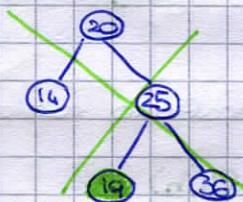
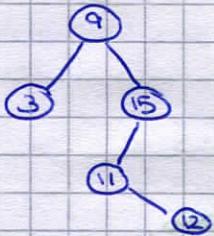
Albero completo  $\xrightarrow{4X}$  bifasciato

### ALBERO BINARIO DI RICERCA (ABR)

È un albero binario che soddisfa le seguenti proprietà:

Sia  $x$  un nodo in un albero binario di ricerca. Se  $y$  è un nodo nel sottoalbero sinistro di  $x$ , allora  $y.\text{key} \leq x.\text{key}$

Se  $y$  è un nodo nel sottoalbero di destra di  $x$ , allora  $y.\text{key} \geq x.\text{key}$



$\rightarrow$  Sbagliato: non è un ABR

La proprietà di ricerca consente di estrarre in ordine non decrescente le chiavi di un ABR visitando l'albero in ordine simmetrico.

- Dimostro per induzione sul nº di nodi di  $n$  che si visita in ordine cresc.  $\rightarrow$  x LASA

### ABR

T con un attributo root

T.root = nodo radice

x = nodo dell'albero

x.key

x.left

x.right

x.p

TreeSearch(Node x, Elm k)

```

if x == NULL or x.key == k
    return x
  
```

else

```

    if k < x.key
      return TreeSearch(x.left, k)
  
```

else

```

      return TreeSearch(x.right, k)
  
```

TreeSearch (Node x, Elm k)  $\rightarrow$  Node

/\* Post: restituisce un nodo con chiave k se esiste, altrimenti doi NULL \*/

O(h)

I nodi incontrati durante la ricorsione formano un cammino verso le basse dalla radice dell'albero e quindi il tempo di esecuzione è  $O(h)$  dove  $h$  è l'altezza dell'albero.

## CORRETTEZZA

Se  $x == \text{NULL}$  oppure  $x.\text{key} == k$  OK!

restituisce correttamente NULL o il nodo  $x$  che contiene la chiave  $k$ .

Se  $k < x.\text{key}$ , la proprietà degli alberi binari di ricerca assicura che nel sottoalbero di destra non ci possono essere nodi con chiave  $< k$ .

```
Iterative_Tree_Search( Node x, E& k )
    while x ≠ NULL AND x.key ≠ k
        if k < x.key
            x = x.left
        else
            x = x.right
    return x
```

$O(h)$

È più efficiente di quella ricorsiva

/\* Pre :  $x \in T */$

Post : restituisce il minimo del sottoalbero radicato in  $x */$

```
Tree_minimum( Node x ) → Node
    while x.left ≠ NULL
        x = x.left
    return x
```

$O(h)$

a ogni ciclo scendo di livello,  
sto costruendo un cammino  
dalla radice alle foglie

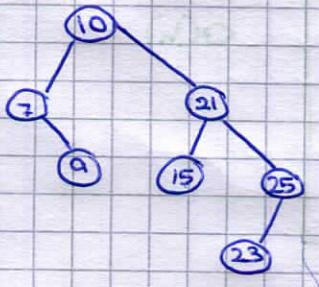
## CORRETTEZZA

Se  $x$  è un nodo che non ha sottoalbero sinistro allora poiché ogni chiave nel sottoalbero destro è almeno grande quanto  $x.\text{key}$ ; la chiave minima nel sottoalbero con radice  $x$  è  $x.\text{key}$ .

Se  $x$  ha un sottoalbero sinistro allora NELL'ORDINE nessuna chiave nel sottoalbero destro è minore di  $x.\text{key}$  e ogni chiave nel sottoalbero sinistro non è maggiore di  $x.\text{key}$  dunque la chiave minima si troverà nel sottoalbero sinistro di  $x$ .

## PREDECESSORE E SUCCESSORE

Dato un nodo in un ABR, le sue successori e le sue predecessori sono i nodi che lo seguono o lo precedono immediatamente nell'ordine stabilito da una visita simmetrica.



$$\text{succ}(10) = 15$$

II)  $x$  ha un figlio destro; il successore è il minimo del sottoalbero destro di  $x$

III)  $x$  non ha un figlio destro;  $\text{succ}(x)$ , se esiste, è l'antenato più prossimo di  $x$  verso la radice fino ad incontrare la prima svolta a destra.

/\* Pre: x è int \*/  
 /\* Post: restituisce il successore di x in una visita simmetrica se esiste, altrimenti null \*/

Tree\_Successore (Node x) -> Node

```

if x.right ≠ NULL
    return Tree_minimum (x.right)
else
    y = x.p
    while y ≠ NULL AND x = y.right
        x = y
        y = x.right
    y = x.p
    return y
    
```

O(h)

/\* Post: inserisce il nodo z in T rispettando le proprietà degli ABR  
 z è un nodo t.c.

```

z.key = k
z.left = NULL
z.right = NULL
*/
```

Tree\_insert (Tree T, Node z)

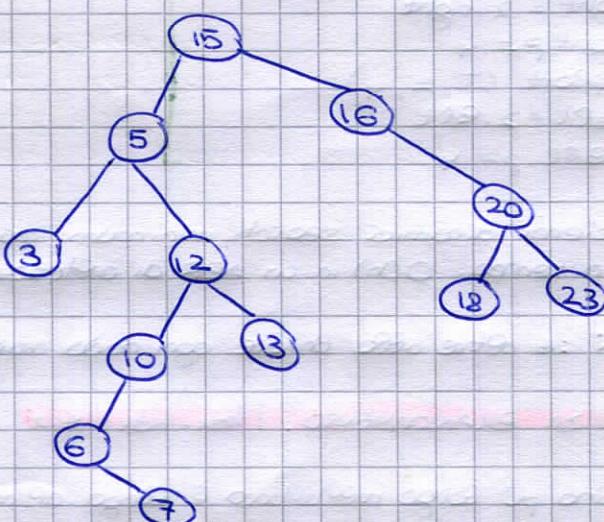
```

y = NULL      /* padre di x */
x = T.root
while x ≠ NULL
    y = x
    if z.key < x.key
        x = x.left
    else
        x = x.right
    z.p = y
    if y == NULL
        T.root = z
    else
        if z.key < y.key
            y.left = z
        else
            y.right = z
    
```

O(h)

28.11.2013

Proposizione: Se un nodo in un ABR ha due figli, allora il suo successore non ha un figlio sx e il suo predecessore non ha un figlio dx



DIMOSTRAZIONE ( $\Rightarrow$ ) Sia  $x$  un nodo con 2 figli. In una visita simmetrica, i nodi nel sottoalbero sx precedono  $x$ , quelli nel sottoalbero dx seguono  $x$ . Così  $\text{pred}(x)$  è nel sottoalbero sx e  $y$  è successore nel dx.

Sia  $s$  il successore di  $x$ . Assumiamo che  $s$  abbia un figlio sx, che chiamiamo  $y$ .  $y$  segue  $x$  nella visita simmetrica perché è nel sottoalbero dx di  $x$ , ma precede  $s$  perché è nel suo sottoalbero sx.

Dunque abbiamo  $x \rightsquigarrow y \rightsquigarrow s$  ASSURDO perché  $s$  non sarebbe più successore.

### NODI DA CANCELLARE

- 1)  $z$  non ha figli, modifichiamo il padre
- 2)  $z$  ha un figlio unico, stacchiamo  $z$  creando un collegamento tra suo figlio e suo padre
- 3) se il nodo  $z$  ha due figli, troviamo il successore  $y$  di  $z$  che deve trovarsi nel sottoalbero dx di  $z$  e facciamo in modo che  $y$  assuma la posizione di  $z$  nell'albero.

N.B!  $y$  non ha figlio sx (cfr. sopra)

TRANSPLANT: sostituisce il sottoalbero con radice nel nodo  $u$  con il sottoalbero con radice in  $v$

/\*Pre:  $u$  è un nodo dell'albero\*/

Transplant (Tree T, Node u, Node v)

```
if  $u.p == \text{NULL}$ 
    T.root = v
else
    if  $u == u.p.\text{left}$ 
         $u.p.\text{left} = v$ 
    else
         $u.p.\text{right} = v$ 
        if  $v \neq \text{NULL}$ 
             $v.p = u.p$ 
```

Tree\_Delete (Tree T, Node z)

```
if  $z.\text{left} == \text{NULL}$ 
    Transplant (T, z, z.right)
else
    if  $z.\text{right} == \text{NULL}$ 
        Transplant (T, z, z.right)
    else
        y = Tree_Minimum (z.right)
        if  $y.p \neq z$ 
            Transplant (T, y, y.right)
            y.right = z.right
            z.right.p = y
        Transplant (T, z, y)
        y.left = z.left
        y.left.p = y
```

/\*Pre:  $z$  è un nodo in  $T$ \*/

$O(h)$

Teorema: Gli operazioni sugli insiemi dinamici search, minimum, maximum, succ, pred, insert, delete, possono essere eseguite in tempo  $O(h)$  in un ABB di altezza  $h$ .

1) Alberi AVL / ABR bilanciati: oltre alle chiavi ogni nodo mantiene un'informazione sul bilanciamento

FATTORE DI BILANCIMENTO:  $|h_{\text{left}} - h_{\text{right}}| \leq 1$

2) Alberi rosso-neri: il cammino più lungo nell'albero è lungo al massimo il doppio di quello più breve

3) B-Alberi: alberi di ricerca bilanciati non binari