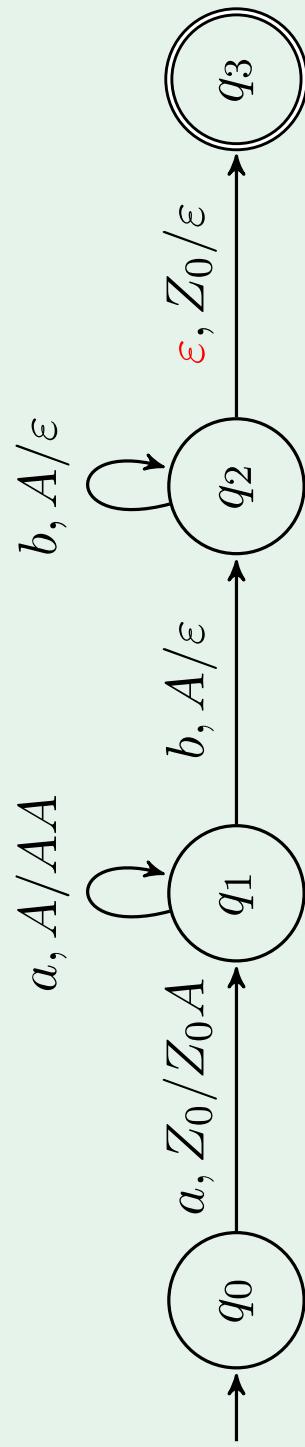


Esempio: Riconoscere  $\{a^n b^n \mid n > 0\}$

Un'alternativa



$\varepsilon - \text{mossa}$

- Questo automa effettua una mossa senza leggere dall'input
- Posso evitare di usare  $B$  come "marcatore della prima  $a$ "

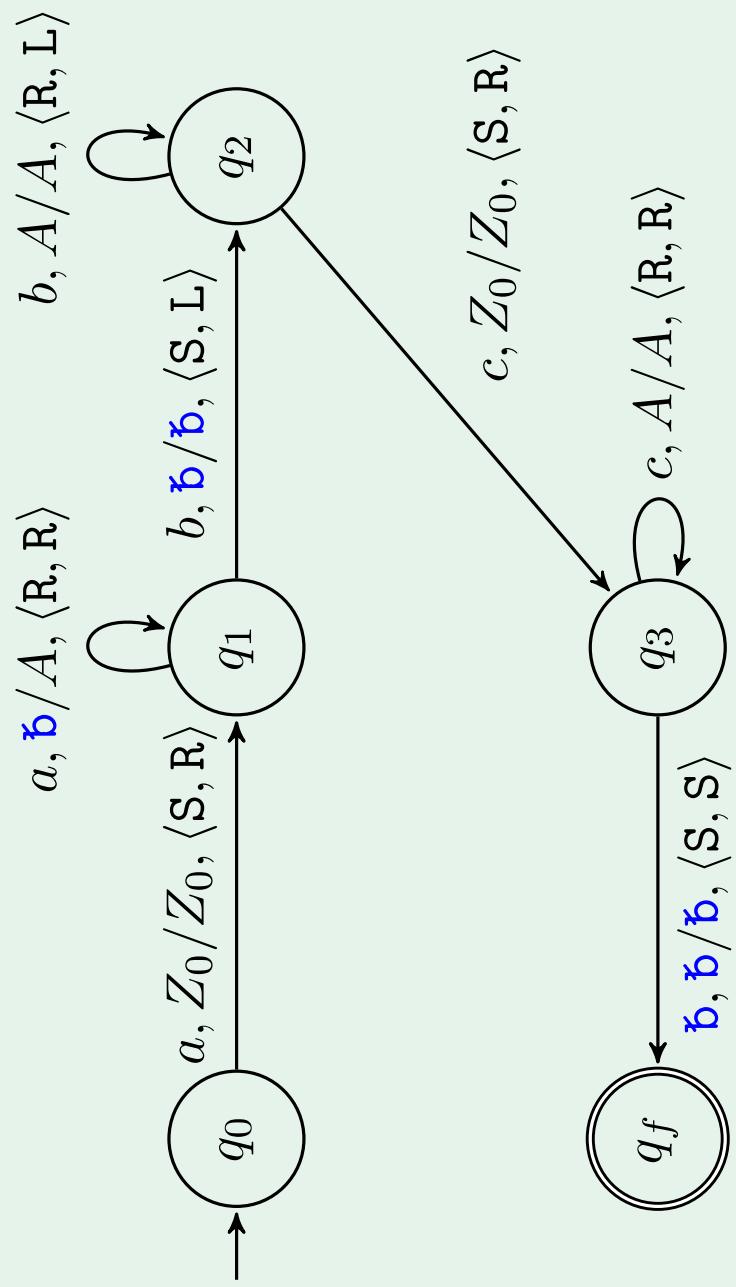
## Conseguenze delle proprietà

### La famiglia $L_{AP}$

- $L_{AP}$ : la famiglia di linguaggi riconosciuti dagli AP deterministici
- $L_{AP}$  non è chiusa rispetto all'unione per quanto detto
- $L_{AP}$  è chiusa rispetto al complemento? Sì.
  - Il principio della dimostrazione è lo stesso degli FSA, scambiare  $F$  con  $Q \setminus F$
- $L_{AP}$  non è chiusa rispetto all'intersezione (perché?)

## MT: esempi

Riconoscere  $L = \{a^n b^n c^n, n > 0\}$



## Esempio di traduzione: Incremento di un numero decimale

### Schema di funzionamento

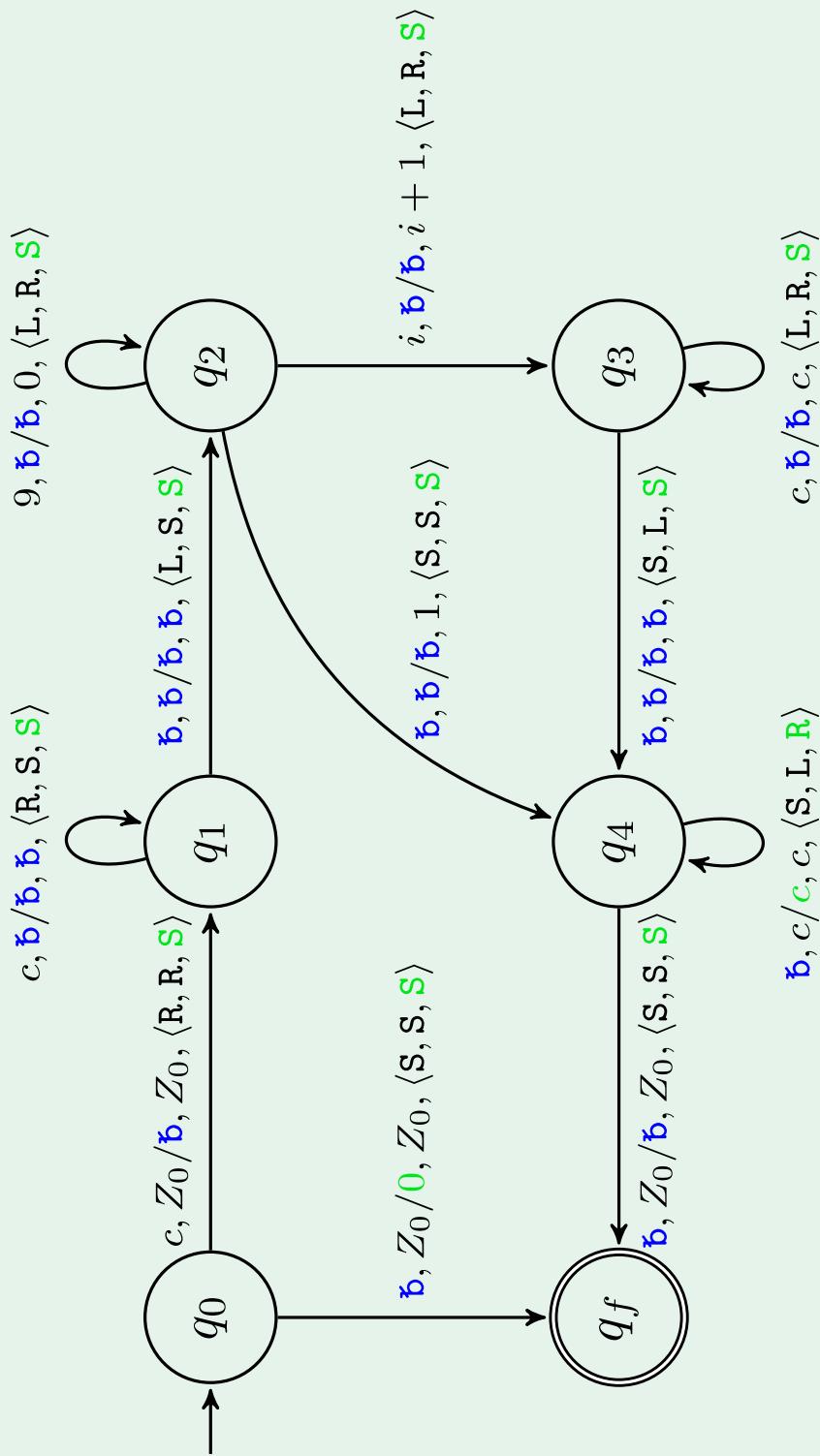
- ① Scorri l'input fino alla fine
- ② Scorrendo all'indietro l'input, scrivi 0 sul nastro di memoria fino alla prima cifra diversa da 9
- ③ Scrivi il successore della cifra corrente sul nastro di memoria, dopodichè copia le cifre rimanenti dall'ingresso su di esso
- ④ Scorrendo al contrario il nastro di memoria, copia le cifre in uscita

### Notazione sintetica

- ① i: una qualunque cifra tra 0 e 8
- ② c: una qualunque cifra decimale
- ③  $i+1$ : il successore della cifra decimale  $i$

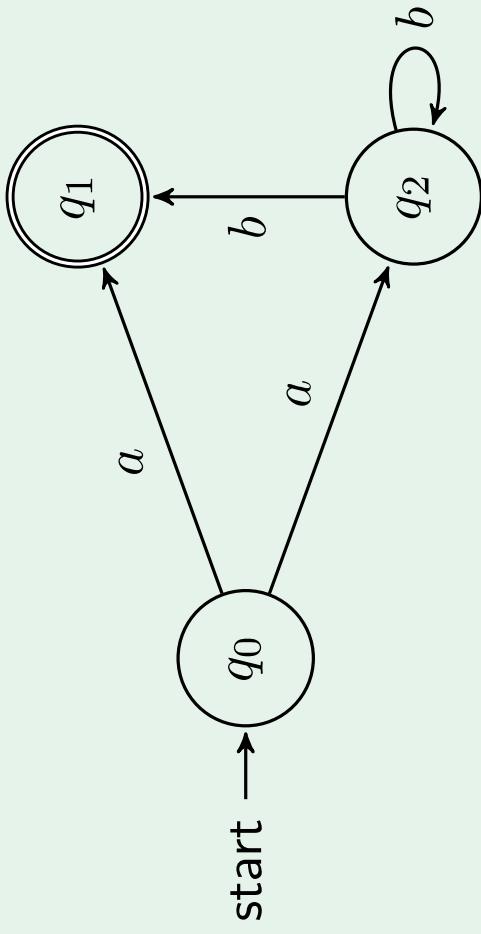
## MT: esempi

Incrementatore decimale (con  $\tau(\varepsilon) = 0$ )



## Versioni nondeterministiche (ND) di modelli noti

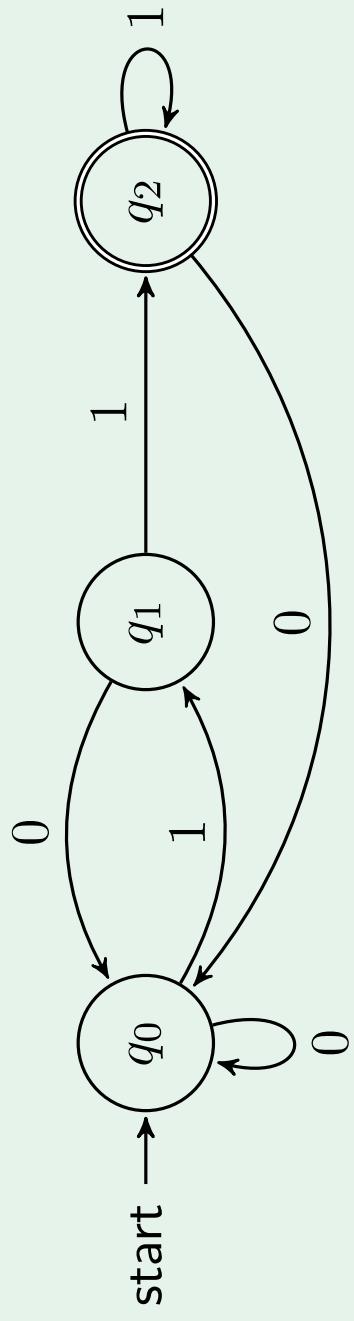
Automi a stati finiti ND (riconosce  $L = ab^*$ )



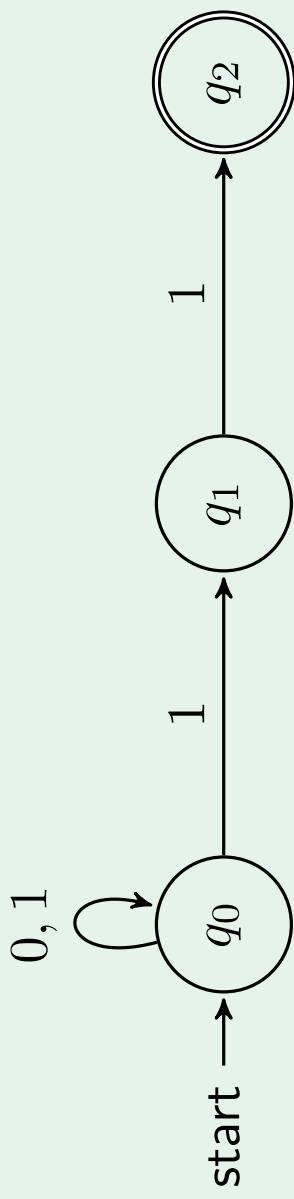
- La  $\delta$  diventa  $\delta : Q \times I \mapsto \wp(Q)$
- Nell'esempio abbiamo  $\delta(q_0, a) = \{q_1, q_2\}$ ;  $\delta(q_2, b) = \{q_1, q_2\}$

## Automi a stati finiti ND e D a confronto

FSA-D per  $L = \{0, 1\}^* 11$  (stringhe che terminano in 11)



FSA-ND per  $L = \{0, 1\}^* 11$  (stringhe che terminano in 11)



# Conclusioni sul nondeterminismo

## Un utile formalismo

- Utile per rappresentare problemi/algoritmi dove alcune scelte locali non sono fattibili al momento/importanti
- Aumenta la potenza dei soli AP (tra i formalismi visti)
- Può essere applicato praticamente a tutti i modelli di calcolo (estensione facile ai traduttori)
- N.B.: nondeterministico  $\neq$  probabilistico
  - La computazione procede sempre con certezza verso l'insieme di stati successivo
  - Esistono modelli di calcolo probabilistico, ma sono ben diversi (e.g., FSA-prob  $\approx$  catene di Markov)

# Grammatica

## Definizione formale

- Una grammatica è una quadrupla  $G = \langle \mathbf{V}_t, \mathbf{V}_n, \mathbf{P}, S \rangle$ 
  - $\mathbf{V}_t$ : alfabeto o vocabolario *terminale*
  - $\mathbf{V}_n$ : alfabeto o vocabolario *nonterminale*
  - $\mathbf{V} = \mathbf{V}_n \cup \mathbf{V}_t$ : alfabeto o vocabolario
  - $S \in \mathbf{V}_n$ : elemento di  $\mathbf{V}_n$  detto *assiomma* o *simbolo iniziale*
  - $\mathbf{P} \subseteq \mathbf{V}_n^+ \times \mathbf{V}^*$  insieme delle produzioni sintattiche o *regole di riscrittura*
- Per semplicità di notazione, indicheremo gli elementi  $p \in \mathbf{P}$ ,  $p = \langle \alpha, \beta \rangle$  come  
$$p = \alpha \rightarrow \beta$$

Alcuni esempi - 1

- Una prima grammatica semplice *G*<sub>1</sub>

- $V_t = \{a, b, c\}$ ,  $V_n = \{S, A, B, C\}$  con assioma  $S$
  - $P = \{S \rightarrow A, A \rightarrow aA, A \rightarrow B, B \rightarrow bB, B \rightarrow C, C \rightarrow cC, C \rightarrow \varepsilon\}$
  - Una possibile derivazione è  
$$S \Rightarrow A \Rightarrow aaA \Rightarrow aaB \Rightarrow accC \Rightarrow aaccC \Rightarrow aaccC \Rightarrow aacc$$
  - Un'altra possibile derivazione è  $S \Rightarrow A \Rightarrow B \Rightarrow bB \Rightarrow bC \Rightarrow b$
  - Linguaggio generato da  $G$ ,  $L(G) = \{a^*b^*c^*\}$

## Alcuni esempi - 2

Qualcosa di più sostanzioso  $G_2$

- $V_t = \{a, b\}$ ,  $V_n = \{S\}$  assioma  $S$
- $P = \{S \rightarrow aSbS, S \rightarrow \epsilon\}$
- Una possibile derivazione  $S \Rightarrow aSbS \Rightarrow aSb \Rightarrow aaSbSb \Rightarrow aaSbb \Rightarrow aabb$
- Una ulteriore possibile derivazione  
 $S \Rightarrow aSbS \Rightarrow aSbaSbS \Rightarrow abaSbS \Rightarrow ababS \Rightarrow abab$
- Linguaggio generato dalla grammatica? Coppie di  $a$  e  $b$  “ben parentetizzate”

## Espressività delle grammatiche

Quali linguaggi è possibile esprimere? La gerarchia di Chomsky

- 4 classi, a seconda delle limitazioni (crescenti) imposte sulle produzioni  $\alpha \rightarrow \beta$

$A \rightarrow B$	Tipo	Nome	Limitazione Produzioni
$\uparrow$	0	Non limitate	nessuna
$\in V_h^+$	1	Monotone (non-decreasing)	$ \alpha  \leq  \beta $
$v^*$	1	Dipendenti dal contesto	$\alpha = \gamma A \delta \quad \beta = \gamma \chi \delta, \chi \neq \varepsilon \circ S \rightarrow \varepsilon$
$\exists \text{ un riconoscitore CHC conosciute }$ $\text{SE UNA STRUTTURA }$ $\in AL \text{ LINGUAGGIO}$	2	Libere dal contesto	$ \alpha  = 1$
			di forma $A \rightarrow a, A \rightarrow aA$ , oppure $A \rightarrow a, A \rightarrow Aa$ con $a \in V_t, A \in V_n$

- Una grammatica più potente genera tutti i linguaggi di una meno potente.  
Domanda: l'inclusione è stretta?

# A quali automi corrispondono?

## Grammatiche Regolari e FSA

- Linguaggi generati da grammatiche regolari  $\equiv$  riconosciuti da FSA

## Dall'FSA $\mathcal{A}$ alla grammatica

- Poniamo  $\mathbf{V}_n = \mathbf{Q}, \mathbf{V}_t = \mathbf{I}, S = \langle q_0 \rangle$
- Per ogni  $\delta(q, i) = q'$  aggiungiamo  $\langle q \rangle \rightarrow i\langle q' \rangle$  all'insieme  $\mathbf{P}$
- Se  $q' \in \mathbf{F}$  per una data  $\delta(q, i) = q'$ , aggiungiamo anche  $\langle q \rangle \rightarrow i$  all'insieme  $\mathbf{P}$
- Facile mostrare per induzione che  $\delta^*(q_0, x) = q' \text{ sse } \langle q_0 \rangle \xrightarrow{*} x\langle q' \rangle$

## Dalla grammatica all'FSA (non deterministico)

- $\mathbf{Q} = \mathbf{V}_n \cup \{q_f\}, \mathbf{I} = \mathbf{V}_t, q_0 = S, \mathbf{F} = \{q_f\}$
- Se  $A \rightarrow bC \in \mathbf{P}, \delta(A, b) = C; \text{Se } A \rightarrow b \in \mathbf{P}, \delta(A, b) = q_f$

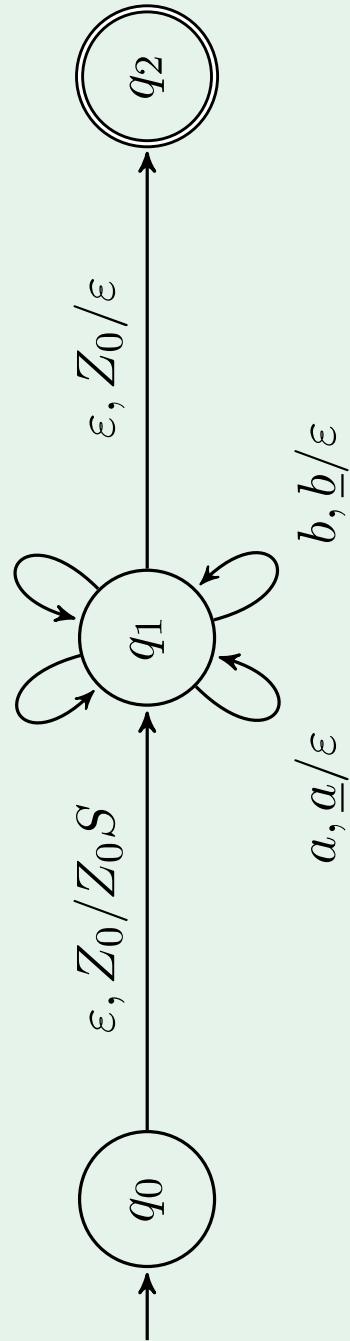
## A quali automi corrispondono?

### Grammatiche Libere dal Contesto e AP-ND

- I linguaggi generati dalle grammatiche libere dal contesto coincidono con i riconosciuti dagli AP non deterministici
- Dimostrazione non banale; intuizione: salvo in pila la coda della forma di frase

### Dalla grammatica all'AP ND: un esempio illustrativo

- Data  $\langle \{S\}, \{a, b\}, \{S \rightarrow aSb, S \rightarrow ab\}, S \rangle$
- $\varepsilon, S/ba$
- $\varepsilon, S/\underline{b}S\underline{a}$



# Rimanenti corrispondenze

## Automi a pila deterministici

- Esiste un sottoinsieme (proprio) delle grammatiche libere dal contesto che genera i linguaggi riconosciuti dagli AP deterministici
- Restrizione difficile da esprimere sulla forma delle produzioni
- Dettagliata nel corso di Formal Languages and Compilers

## Grammatiche dipendenti dal contesto

- Le grammatiche di tipo 1 corrispondono a un sottoinsieme delle MT di cui è certo che terminino sempre
- N.B. non si tratta delle uniche MT che terminano sempre
- Consentono sempre di sapere se una stringa  $x$  è generata da  $G$  (si può costruire l'insieme delle stringhe gen. da  $G$  lunghe quanto  $x$  e vedere se essa appare)

## Enumerazione algoritmica

### Enumerare un insieme

- Enumerazione  $\mathcal{E}$  di un insieme = corrispondenza biunivoca tra i suoi elementi e quelli di  $\mathbb{N}$
- Enumerazione *algoritmica*:  $\mathcal{E}$  è *effectively computable* (efficacemente calcolabile) se esiste un algoritmo (o una MT) che la calcola
- Enumerazione algoritmica di  $L = \{a^*b^*\}$ ,  $\mathcal{E} : L \rightarrow \mathbb{N}$ : “etichetta” le stringhe in ordine crescente di lunghezza, se hanno la stessa lunghezza, “etichetta” in ordine lessicografico
  - $\varepsilon \mapsto 0, a \mapsto 1, b \mapsto 2, aa \mapsto 3, ab \mapsto 4, bb \mapsto 5, aaa \mapsto 6, aab \mapsto 7 \dots$

### Primo fatto sulle MT

- Le MT sono *algoritmicamente enumerabili* (dimostriamolo)

## Convenzioni aggiuntive

### Convenzioni sul calcolo

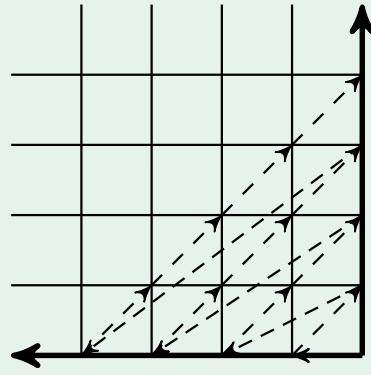
- D'ora in poi, problema = calcolo di una funzione  $f : \mathbb{N} \rightarrow \mathbb{N}$
- $f_i$  = funzione calcolata dalla  $i$ -esima MT (secondo enumerazione di Gödel)
- N.B.:  $f_i(x) = \perp$  se  $\mathcal{M}_i$  non si ferma quando riceve in ingresso  $x$
- Convenzione:  $f_i(x) = \perp$  se e solo se  $\mathcal{M}_i$  non si ferma quando riceve in ingresso  $x$ 
  - Data una generica  $\mathcal{M}_i$  basta fare in modo che proceda all'infinito (e.g. sposti all'infinito la testina verso sx) se non calcola un valore significativo per  $f_i(x)$
  - La convenzione consente di non dover trattare gli stati finali separatamente

# Macchina di Turing Universale

## Calcolare una generica MT con un'altra

- Esiste (almeno) una *Macchina di Turing universale (MTU)*
  - è la MT che calcola  $g(i, x) = f_i(x)$
- La MTU non sembra essere dello stesso tipo delle altre  $\mathcal{M}_i$  perchè  $f_i(\cdot)$  è funzione di una variabile,  $g(\cdot, \cdot)$  di due
- Proviamo il contrario ricordando che  $\mathbb{N} \times \mathbb{N}$  è enumerabile

Mappa di Cantor



$$d \text{ associa } (i, x) \text{ a } n \in \mathbb{N}$$
$$d(x, i) = \frac{(x+i)(x+i+1)}{2} + i$$

- N.B.  $d$  è invertibile: dato  $n$  ottengo una e una sola  $(x, y)$

## A confronto con la pratica

### MT, MTU, ASIC e Calcolatori programmabili

- Abbiamo visto che una MT è un modello molto semplice di calcolatore in qualche senso analogo ad una macchina con programma cablato (un ASIC)
- Una MTU è l'analogo di un calcolatore programmabile
  - Il numero di Gödel  $i$  agisce da “codice” del programma, l’ingresso  $x$  sono i dati
- Una MTU (e anche la sua implementazione<sup>a</sup>) può essere molto semplice: ne esiste una con 4 stati e 6 simboli
- Nulla vieta che  $i$  sia il numero di Gödel di un’altra MTU e che  $x$  contenga quindi il numero di Gödel e i dati da far girare nella MTU “emulata”: modello di una macchina virtuale

---

<sup>a</sup> [https://en.wikipedia.org/wiki/One\\_instruction\\_set\\_computer](https://en.wikipedia.org/wiki/One_instruction_set_computer)

## Quali sono i problemi risolvibili?

### Il problema della terminazione del calcolo

- Problema di carattere estremamente pratico:
  - Costruisco un programma
  - Lo eseguo su dei dati in ingresso
  - So che il programma potrebbe non terminare la sua esecuzione (in gergo, “va in loop”)
  - Posso determinare, con un metodo generale, se e quando questo accade?
- Rifrasando in termini equivalenti di MT:
  - Consideriamo la funzione  $g(i, x) = \begin{cases} 1 & \text{se } f_i(x) \neq \perp \\ 0 & \text{se } f_i(x) = \perp \end{cases}$
  - Esiste una MT che calcola  $g$ ?

## Il problema della terminazione del calcolo

... non può essere risolto

- **NON esiste** una MT che calcola la  $g$  appena definita!
- Di conseguenza, nella pratica:
  - Non esiste un compilatore che possa dirci che il nostro programma andrà in loop su un dato input
  - Non possiamo costruire l'antivirus definitivo che sia in grado di capire a priori se un programma è malevolo
    - Non possiamo "creare un programma per tentativi ciechi" controllando solo a posteriori che sia quello corretto
- Per contro, dire se ci siam dimenticati una parentesi in un linguaggio di programmazione (ben progettato) è fattibile: basta un AP per risolvere il problema

## Dimostrazione

### Dimostrazione - Seguito

- A questo punto cosa succede se calcolo  $h(x_h)$  ?
  - Caso  $h(x_h) = 1$ : Dato che  $f_{x_h} = h$ , si ha che  $f_{x_h}(x_h) = 1$ . Tuttavia, per definizione di  $h$  abbiamo che  $g(x_h, x_h) = 0$ , ma quindi, per definizione di  $g$ ,  $f_{x_h}(x_h) = \perp \neq 1$
  - Caso  $h(x_h) = \perp$ : Dato che  $f_{x_h} = h$ , si ha che  $f_{x_h}(x_h) = \perp$ . Tuttavia, per definizione di  $h$  abbiamo che  $g(x_h, x_h) = 1$ , ma quindi, per definizione di  $g$ ,  $f_{x_h}(x_h) \neq \perp \neq \perp$
- Otteniamo una contraddizione in entrambi i casi. ■

### Una visione intuitiva della dimostrazione

Se ho un programma  $g(i, x)$  in grado di dire se un generico altro programma  $f_i$  termina, posso usarlo per costruire un altro programma  $h$  che fa sempre sbagliare  $g(i, x)$  nel determinare se quest'ultimo (ovvero  $h$ ) termina.

## Generalizzazioni e specializzazioni

### Un corollario del teorema precedente

- $$h'(x) = \begin{cases} 1 & \text{se } g(x, x) = 1 \text{ (se } f_x(x) \neq \perp) \\ 0 & \text{se } g(x, x) = 0 \text{ (se } f_x(x) = \perp) \end{cases}$$
 non è calcolabile
- N.B. non può essere ricavato come conseguenza del teorema precedente (che copre un caso più generale, dove  $i$  e  $x$  possono esser diversi)
- Se un dato problema  $P$  non è calcolabile un suo caso particolare (= stesso problema su una restrizione del dominio) potrebbe esserlo
  - ma una sua generalizzazione (= stesso problema con dominio ampliato) non lo è mai
  - Se un dato problema  $P$  è calcolabile, una sua generalizzazione potrebbe non esserlo (ma una sua specializzazione è sempre calcolabile)

## Un ulteriore importante problema indecidibile

Calcolare se una funzione parziale calcolabile è una funzione (totale)

$$k(i) = \begin{cases} 1 & \text{se } \forall x \in \mathbb{N}, f_i(x) \neq \perp \text{ (}f_i(\cdot) \text{ totale)} \\ 0 & \text{se } \exists x \in \mathbb{N}, f_i(x) = \perp \text{ (}f_i(\cdot) \text{ non totale)} \end{cases}$$

- Problema simile ma diverso dal precedente: voglio sapere se il calcolo di  $f_i$  termina per tutti i suoi input

- Saper dire, per un  $x$  fissato, se  $f_i(x) \neq \perp$  non mi consente di dire sicuramente se  $f_i(\cdot)$  è totale: posso provare un po'di  $x$ , ma potrei non trovare quello critico per cui  $f_i(x) = \perp$

- Viceversa, potrei essere in grado di dire che una data  $f_i$  non è totale, anche se non so decidere se  $f_i(x) = \perp$  per un  $x$  fissato
- In pratica: questo è il problema di determinare se, dato un programma, termina su un qualsiasi dato in ingresso.

## Problema risolvibile $\neq$ problema risolto

Saper che la soluzione esiste  $\neq$  essere in grado di determinarla

- Spesso è possibile dare una dimostrazione non costruttiva: dimostro che una soluzione esiste, ma non mostro come ricavarla in generale
  - E' il caso tipico di problemi che non hanno soluzione nota in forma chiusa
- Nel contesto della calcolabilità: posso sapere che *esiste* una MT che risolve il problema, anche se non sono in grado di dire *quale* sia tra le possibili
- Alcuni esempi con problemi a risposta binaria:
  - È vero che una partita a scacchi "perfetta" termina in parità?
  - È vero che ogni intero  $> 2$  è la somma di due primi?

# Problema risolvibile $\neq$ problema risolto

## Problemi a risposta chiusa

- In un problema con risposta binaria fissa per ogni input so a priori che la risposta è “sì” o “no” anche se non so quale sia
- Ricordando che per noi problema = funzione e risolvere un problema = calcolare una funzione, che funzioni associamo ai problemi precedenti?
- Codificando “sì” = 1, “no” = 0, i due problemi precedenti sono espressi dalle una tra le due funzioni  $f_1(x) = 1$ ,  $f_0(x) = 0$
- Dunque sono risolvibili problemi come:
  - Dire se  $g(10, 20) = 1$ , ossia se  $f_{10}(20) \neq \perp$
  - Dire se  $\forall x \in \{10, 11, 12\}, g(x, 20) = 1$
- Sono tutti problemi con risposta “sì” o “no”: possiamo non riuscire a determinare quale tra le due sia corretta, ma sicuramente sono calcolabili

## Un caso più interessante

### Calcolare le cifre di $\pi$

- $f(x) = x$ -esima cifra dell'espansione decimale di  $\pi$ 
  - $f$  è calcolabile, è noto più di un algoritmo (MT) che la calcola
- Date le capacità attuali di calcolare  $f$  ci domandiamo se  $g(x)$  sia calcolabile:  
 $g(x) = 1$  se nell'espansione di  $\pi$  sono presenti  $x$  cifre 5 consecutive, 0 altrimenti
- Calcolando la sequenza  $f(0) = 3, f(1) = 1, f(2) = 4,$   
 $f(3) = 1, f(4) = 5, f(5) = 9$  sappiamo che  $g(1) = 1$
- A priori, i valori di  $g(x)$  saranno distribuiti tra 0 e 1 in un modo deterministico,  
ma non predicibile (al meglio delle nostre conoscenze)

# $g(x)$ è calcolabile?

## Un approccio enumerativo

- Data  $x$ , se  $g(x) = 1$  lo scoprirei sempre (basta calcolare abbastanza cifre di  $\pi$ )
- Se  $g(x) = 0$  non posso esserne certo semplicemente calcolando un grande numero di cifre di  $\pi$ : la sequenza che cerco potrebbe essere appena dopo!
- Consideriamo la congettura: “Qualsiasi sia  $x$  esiste una sequenza di  $x$  cifre in nell’espansione di  $\pi$ ?”
- Se fosse vera,  $g(x)$  sarebbe calcolabile banalmente (sarebbe costante)
- In conclusione, date le conoscenze attuali, non sappiamo dimostrare né che  $g$  sia calcolabile, né che non lo sia

# Decidibilità e semidecidibilità

## Insiemi decidibili

- Concentriamoci su problemi con risposta binaria:
  - Essi richiedono di calcolare una certa  $f : \mathbb{N} \rightarrow \{0, 1\}$
  - Interpretiamo  $f$  come la funzione caratteristica di  $S$

$$1_S : \begin{cases} 1 & \text{se } x \in S \\ 0 & \text{se } x \notin S \end{cases}$$

- $1_S$  è nota anche come funzione indicatrice di  $S$ , notazione alternativa  $\chi_S$
- Problema = dato un insieme  $S \subseteq \mathbb{N}$  dire se  $x \in S$  o meno
- **S** è *ricorsivo* o *decidibile* se e solo se la sua funzione caratteristica è calcolabile
- N.B.:  $1_S$  è una funzione (non funzione parziale) dunque totale per definizione

## Decidibilità e semidecidibilità

### Insiemi semidecidibili

- Un insieme  $\mathbf{S}$  è *ricorsivamente enumerabile* (RE) o *semidecidibile* se e solo se
  - $\mathbf{S}$  è l'insieme vuoto
  - $\mathbf{S}$  è l'immagine di una funzione totale e calcolabile,  $g_{\mathbf{S}}$ :  
$$\mathbf{S} = \mathbf{I}_{g_{\mathbf{S}}} = \{y \mid y = g_{\mathbf{S}}(x), x \in \mathbb{N}\}$$
 ovvero  $\mathbf{S} = \{g_{\mathbf{S}}(0), g_{\mathbf{S}}(1), \dots\}$  da cui il nome *ricorsivamente (algoritmicamente) enumerabile*
- Il termine *semidecidibile* deriva dal fatto che:
  - Se  $y \in \mathbf{S}$  enumerando gli elementi di  $\mathbf{S}$ , ottenuti come  $g_{\mathbf{S}}(0), g_{\mathbf{S}}(1), \dots$  prima o poi trovo il valore di  $x \in \mathbb{N}$  tale per cui  $y = g_{\mathbf{S}}(x)$
  - Se  $y \notin \mathbf{S}$  non sono mai certo di poter rispondere “ $y$  non appartiene a  $\mathbf{S}$ ” enumerando, potrei non aver ancora trovato  $x \in \mathbb{N}$  tale per cui  $y = g_{\mathbf{S}}(x)$

## Legami tra decidibilità e semidecidibilità

Teorema (Decidibilità  $\Rightarrow$  semidecidibilità)

Se un insieme  $S$  è ricorsivo, esso è ricorsivamente enumerabile

Dimostrazione.

- Se  $S$  è vuoto, è RE per definizione.
- Assumiamo  $S \neq \emptyset$ , e costruiamo una funzione totale e calcolabile di cui  $S$  è immagine.  $\exists k \in S \Rightarrow 1_S(k) = 1$  Definiamo  $g_S$  come
$$g_S : \begin{cases} g_S(x) = x & \text{se } 1_S(x) = 1 \\ g_S(x) = k & \text{se } 1_S(x) = 0 \end{cases}$$
- $g_S$  è calcolabile, totale e  $I_{g_S} = S \Rightarrow S$  è RE

N.B. Dim. non costruttiva: non sappiamo se  $S \neq \emptyset$  né in generale calcolare  $g_S$  senza sapere  $1_S$

## Legami tra decidibilità e semidecidibilità

Teorema (semidecidibilità+semidecidibilità=decidibilità)

$S$  è ricorsivo se e solo se sono ric. enumerabili sia  $S$  che il suo complemento  $\bar{S} = \mathbb{N} \setminus S$

$S$  ricorsivo  $\Rightarrow S$  e  $\bar{S}$  RE.

- $S$  ricorsivo  $\Rightarrow S$  RE per teorema precedente
- $S$  ricorsivo  $\Rightarrow 1_S(x)$  calcolabile  $\Rightarrow$  calcolo  $1_{\bar{S}}(x)$  usando  $1_S(x)$  e rispondendo l' opposto del risultato ( $1_S(x)$  calcolabile e totale)  $\Rightarrow \bar{S}$  ricorsivo  $\Rightarrow \bar{S}$  RE ■

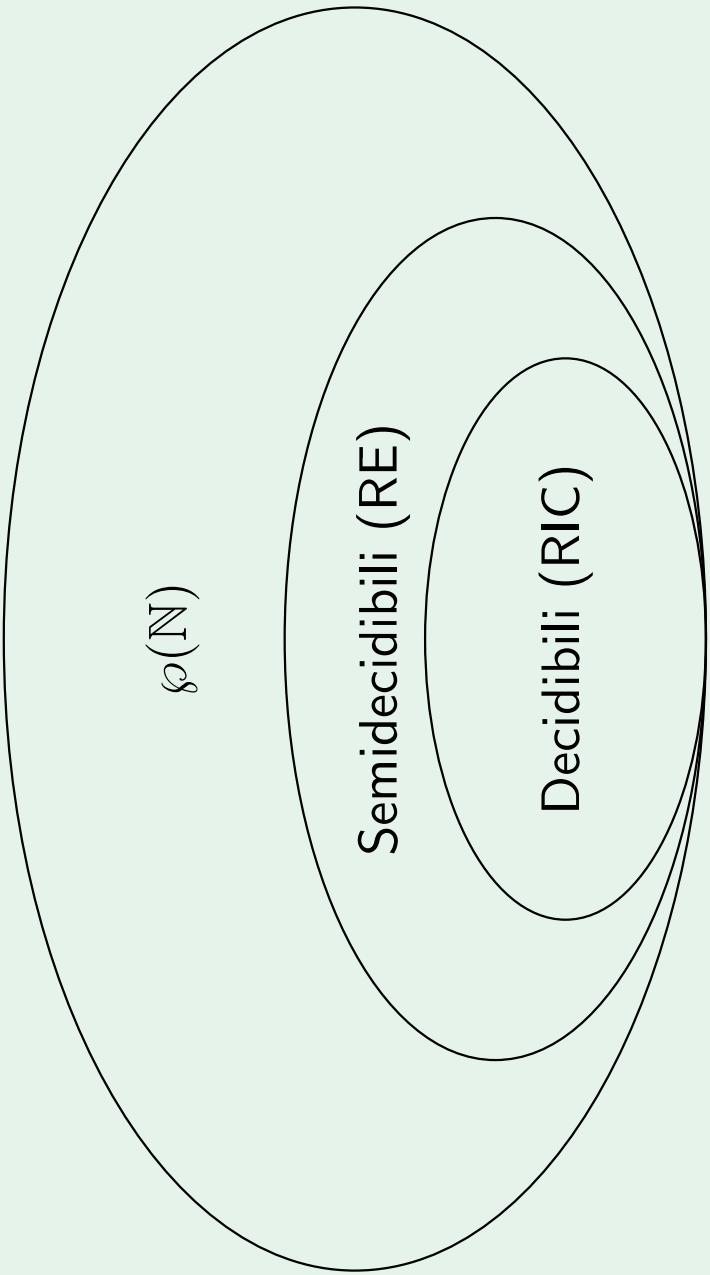
## Conseguenze pratiche e non

### Risvolti pratici

- Non è possibile, con un formalismo RE (MT, grammatiche, funzioni ricorsive) definire *l'insieme di tutte e sole le f calcolabili totali*
- Non posso in nessun modo descrivere come è fatto l'insieme di tutti e soli i programmi che terminano sempre:
  - Gli FSA e gli AP-Det definiscono solo funzioni totali, ma non tutte
  - Le MT definiscono tutte le funzioni calcolabili, ma anche quelle non totali
  - Il C mi permette di scrivere qualunque algoritmo, ma anche quelli che non terminano
  - Esiste un sottoinsieme del C che definisce tutti e soli i programmi che non terminano? **NO.**

## Riassumendo

Una gerarchia degli insiemi (di problemi, di linguaggi, di numeri interi)



- I contenimenti sono tutti stretti ( $\subset$ , non  $\subseteq$ )
- Gli insiemi semidecidibili non sono chiusi per complemento rispetto

## Verso il teorema di Rice

### Il teorema di Kleene del punto fisso

- Sia una funzione  $t(\cdot)$  totale e calcolabile. È sempre possibile trovare una  $f_p$ ,  $p \in \mathbb{N}$  tale che  $f_p = f_{t(p)}$ . La funzione  $f_p$  è detta punto fisso di  $t(\cdot)$ .

## Il teorema di Rice

### Teorema

**$F$  insieme di funzioni computabili, l'insieme  $S$  degli indici delle  $MT$  che calcolano le funzioni di  $F$ ,  $S = \{x | f_x \in F\}$ , è decidibile se e solo se  $F = \emptyset$  o  $F$  è l'insieme di tutte le funzioni computabili.**

### Dimostrazione - 1

- Per assurdo. Supponiamo  $S$  sia decidibile,  $F$  non vuoto e diverso dall'insieme di tutte le funzioni computabili
- Consideriamo  $\mathbf{1}_S(x) = \{1 \text{ se } f_x \in F, 0 \text{ altrimenti}\}$ ; essa è calcolabile per l'ipotesi appena fatta
- Possiamo quindi calcolare
  - 1 il più piccolo  $i \in \mathbb{N}$  tale che  $f_i \in F$ , ovvero  $i \in S$
  - 2 il più piccolo  $j \in \mathbb{N}$  tale che  $f_j \notin F$ , ovvero  $j \notin S$

## Il teorema di Rice

### Effetti pratici

- In tutti i casi non banali  $\mathbf{S}$ , l'insieme delle funzioni calcolabili con una data caratteristica desiderata, *non è decidibile!*
  - N.B. è quindi *semidecidibile o neppure semidecidibile*
- Posso dire se un programma  $P$  è corretto? Dire se risolve un dato problema? (La macchina  $\mathcal{M}_x$  calcola la sola funzione contenuta nell' insieme  $\mathbf{S} = \{f\}?$ )
- È possibile stabilire l'equivalenza tra due programmi? (La macchina  $\mathcal{M}_x$  calcola la sola funzione contenuta nell'insieme  $\mathbf{S} = \{f_y\}?$ )
- È possibile stabilire se un generico programma gode di una qualsiasi proprietà non banale riferita alla funzione che calcola? (e.g. non produce mai un risultato negativo?)

## Stabilire se un problema è (semi)decidibile

### Metodi pratici

- Dire se un generico problema è (semi)decidibile o meno è un problema indecidibile
- Dato uno specifico problema:
  - Se troviamo un algoritmo *che termina sempre* → decidibile
  - Se troviamo un algoritmo che termina sempre se la risposta è “sì”, ma può non terminare se è “no” → semidecidibile
  - Se riteniamo che il problema sia indecidibile come fare? Potremmo costruirci una dimostrazione diagonale ogni volta . . . fattibile, ma parecchio impegnativo!

## Dimostrazioni di non (semi)decidibilità

### La riduzione dei problemi

- Il teorema di Rice ci consente di mostrare che un problema non è decidibile
  - N.B. potrebbe comunque essere semidecidibile!
- Una tecnica alternativa, molto generale, è quella della *riduzione dei problemi*
- L'abbiamo già usata implicitamente, in maniera informale
- Consente di dimostrare in modo agevole che alcuni insiemi non sono neppure semidecidibili

## La tecnica di riduzione

### Una visione operativa

- Se ho un algoritmo per risolvere un problema  $P$ , posso riusarlo (modificandolo) per risolvere  $P'$  simile a  $P$ 
  - Se so risolvere il problema di ricerca di un elemento in un insieme, so calcolare l'intersezione tra due insiemi
  - Se so calcolare unione e complemento di due insiemi, so calcolarne l'intersezione
- In generale, se trovo un algoritmo che, dato un esemplare di  $P'$  ne costruisce la soluzione usando un solutore di  $P$  che so risolvere, ho *ridotto*  $P'$  a  $P$
- Aiuto mnemonico: se ho un programma che risolve  $P'$  chiamando una libreria che risolve  $P$ , ho ridotto il problema risolto dal chiamante a quello risolto dalla libreria

## La tecnica di riduzione

### Formalizzazione

- Sono in grado di risolvere  $y \in \mathbf{S}'$  ( $=$  calcolare  $1_{\mathbf{S}'}(\cdot)$ ) e voglio risolvere  $x \in \mathbf{S}$
- Se ho una funzione  $t$  calcolabile e totale tale per cui  $x \in \mathbf{S} \Leftrightarrow t(x) \in \mathbf{S}'$  riduco il calcolare  $x \in \mathbf{S}$  a  $x \in \mathbf{S}'$ !
  - Dato  $x$ , calcolo  $1_{\mathbf{S}'}(t(x))$  che, equivale a calcolare  $1_{\mathbf{S}}(x)$
- Utile anche in dimostrazioni per assurdo:
  - So di non saper risolvere  $y \in \mathbf{S}'$  ( $\mathbf{S}'$  non è decidibile)
  - **Voglio dimostrare** che neppure  $\mathbf{S}$  è decidibile. HP assurdo:  $\mathbf{S}$  è decidibile
  - Se riesco a ridurre il calcolo di  $y \in \mathbf{S}'$  a quello di  $x \in \mathbf{S}$  ho trovato una contraddizione: so che  $y \in \mathbf{S}'$  non è calcolabile, ma ho appena trovato un modo per calcolarlo, posto che sappia calcolare  $x \in \mathbf{S}$
  - Aiuto mnemonico: devo trovare una riduzione del problema che *so* non essere calcolabile a *quello* che voglio dimostrare non calcolabile

## La tecnica di riduzione

### Esempi di utilizzo

- Dalla non calcolabilità del problema dell'arresto della MT deduciamo la non calcolabilità del problema della terminazione del calcolo in generale
- Siano dati una MT  $\mathcal{M}_i$ , un numero  $x \in \mathbb{N}$ ,
- Costruisco un programma P (e.g., in C) che simula  $\mathcal{M}_i$  e memorizzo il numero  $x$  su un file f
- Il programma P termina la computazione su f se e solo se  $g(i, x) \neq \perp$
- Se sapessi decidere se P termina, sarei in grado di risolvere il problema dell'arresto della MT
- N.B. avremmo potuto dimostrare in modo diretto l'indecidibilità di un programma C enumerando i possibili programmi e applicando la consueta tecnica diagonale... con un po' più di fatica

## La tecnica di riduzione

### Esempi di utilizzo

- Una grande varietà di proprietà tipiche (e che spesso vorremmo eliminare) possono essere dimostrate non decidibili come visto sopra:
  - Il programma effettua un accesso ad un array fuori dai limiti?
  - Il programma effettua una divisione per zero?
  - Questi tipi saranno compatibili a run-time?
  - Questo programma solleverà un'eccezione?

## Considerazioni pratiche

Non decidibili, ma...

- Le proprietà di cui sopra, così come l'arresto della MT non sono decidibili, ma sono semidecidibili:
  - Se la MT si ferma, prima o poi lo scopro, se un programma va in segfault, anche ...
  - Con quale metodo operativo sono in grado di scoprire questo fatto?
  - Se inizio ad eseguire P con ingresso  $y$  e P non si ferma, come faccio a scoprire che P con ingresso  $x$  effettua una divisione per zero?

# Riassumendo

## Una dimostrazione di indecidibilità

- Scoprire molti dei comportamenti indesiderati a runtime di un programma non è decidibile, ma solamente semidecidibile
- N.B. Attenzione a qual è esattamente il problema semidecidibile: la *presenza* del comportamento indesiderato!
- Ma il complemento di un problema (solo) semidecidibile non può essere neppure semidecidibile
  - Se fosse semidecidibile, sarebbero entrambi decidibili!
  - L'assenza di errori a run-time di un programma è quindi un problema indecidibile!
  - Come “consolazione” otteniamo un metodo di dimostrare che un problema è non è neppure semidecidibile: dimostrare che il suo complemento è solamente semidecidibile, ma non decidibile

# Logica del prim'ordine su parole finite

## Sintassi ed interpretazione

- Consideriamo la logica del prim'ordine con predicati su una sola variabile. Consente di descrivere parole su un alfabeto **I**
- Con  $a \in I$  una lettera predicativa per ogni simbolo di **I**, una formula  $\varphi$  è :
  - $a(x)$ : il predicato  $a$  applicato ad una variabile
  - $x < y$
  - $\neg\varphi$ : negazione logica della formula
  - $\varphi \wedge \psi$ : congiunzione (and Booleano) di due formule
  - $\forall x (\varphi)$ : quantificatore universale
- Interpretazione:
  - il dominio delle variabili è un sottoinsieme finito di  $\mathbb{N}$  da pensare come posizioni
  - $<$  corrisponde alla relazione di minore tra le posizioni

POSITION DELL'ATTRAVERSAMENTA  
NELLA PAROLA



# Alcune abbreviazioni

## Concetti ricorrenti

- Come sempre:

- $\varphi_1 \vee \varphi_2 \stackrel{\Delta}{=} \neg(\neg\varphi_1 \wedge \neg\varphi_2)$ : legge di De Morgan
- $\varphi_1 \Rightarrow \varphi_2 \stackrel{\Delta}{=} \neg\varphi_1 \vee \varphi_2$ : definizione di implicazione semplice
- $\exists x(\varphi) \stackrel{\Delta}{=} \neg\forall x(\neg\varphi)$ : proprietà dei quantificatori
- $x = y \stackrel{\Delta}{=} \neg(x < y) \wedge \neg(x > y)$ : definizione di uguaglianza
- $x \leq y \stackrel{\Delta}{=} \neg(y < x)$

- In aggiunta:

- La costante 0:  $x = 0 \stackrel{\Delta}{=} \forall y (\neg(y < x))$
- La funzione successore  $S(x)$ :  $S(x) = y \stackrel{\Delta}{=} (x < y) \wedge \neg\exists z(x < z \wedge z < y)$
- Le costanti 1, 2, 3, ... come  $S(0), S(S(0)), S(S(S(0))), \dots$ .

# Interpretazione come parole su $\Sigma$

## Interpretazione di $a(x)$

- $a(x)$  è vero  $\Leftrightarrow$  l' $x$ -esimo simbolo di una parola  $w \in \Sigma^*$  è  $a$ 
  - gli indici di  $w$  partono da 0

## Esempi

- Formula vera su tutte e sole le parole non vuote che iniziano per  $a$ :  
$$\exists x (x = 0 \wedge a(x))$$
- Formula vera su tutte e sole le parole in cui ogni  $a$  è seguita da una  $b$ :  
$$\forall x (a(x) \Rightarrow \exists y (y = S(x) \wedge b(y)))$$
- Formula vera per la sola stringa vuota:  $\forall x (a(x) \wedge \neg a(x))$ 
  - n.b. se problematico considerare  $\varepsilon$ , la formula non è mai vera

## Altre abbreviazioni convenienti

### Abbreviazioni per indici comodi

- $y = x + 1$  indica  $y = S(x)$  (non esiste un'operatore di somma tra variabili)
- generalizzando, se  $k \in \mathbb{N}, k > 1$  indichiamo con  $y = x + k$   
 $\exists z_1, z_2, \dots, z_{k-1} (z_1 = x + 1, z_2 = z_1 + 1, \dots, y = z_{k-1} + 1)$
- $y = x - 1$  indica  $x = S(y)$ , ovvero  $x = y + 1$ , così come  $y = x - k$  indica  
 $x = y + k$
- $last(x)$  indica  $\neg \exists y (y > x)$

### Esempi

- Parole non vuote terminanti per  $a$ :  $\exists x (last(x) \wedge a(x))$
- Parole con almeno 3 simboli di cui il terzultimo è  $a$   
 $\exists x (a(x) \wedge \exists y (y = x + 2 \wedge last(y)))$  Abbreviando:  $[\exists x (a(x) \wedge last(x + 2))]$

## Proprietà della MFO

### Chiusura rispetto ad operazioni

- I linguaggi esprimibili con MFO sono chiusi per unione, intersezione, complemento
  - Basta combinare le formule con  $\wedge$ ,  $\vee$ ,  $\neg$
- In MFO non posso esprimere  $L = \{a^{2n}, n \in \mathbb{N}\}$  su  $I = \{a\}$
- MFO è strettamente *meno* potente degli FSA
  - Da una formula in MFO  $\varphi$  posso sempre costruire un FSA che riconosce  $L(\varphi)$

## Proprietà della MFO

### Chiusura rispetto alla $*$ di Kleene

- I linguaggi definiti da una formula MFO non sono chiusi rispetto alla  $*$  di Kleene
  - la formula  $a(0) \wedge a(1) \wedge \text{last}(1)$  definisce  $L = \{aa\}$
  - la  $*$ -chiusura di  $L$  è il linguaggio delle stringhe di  $a$  pari
- MFO è in grado di definire i linguaggi *star-free*: sono i linguaggi ottenuti per unione, intersezione, concatenazione e complemento di linguaggi finiti
  - Come definire tutti i REG?

# Logica Monadica del Secondo Ordine (MSO)

## Quantificare insiemi di posizioni

- Per avere lo stesso potere espressivo degli FSA basta “solo” permettere di quantificare sui predici monadici
  - In pratica, quantificare su *insiemi* di posizioni
  - Quantificazione su predici del prim'ordine  $\rightarrow$  logica del secondo ordine
- Ammettiamo formule come  $\exists X (\varphi)$  con  $X$  appartenente all' insieme dei predici monadici (insiemi di posizioni)
- Convenzione: usamo maiuscole e minuscole
  - Maiuscole per indicare variabili con dominio l'insieme dei predici monadici
  - Minuscole per indicare variabili  $\in \mathbb{N}$

# Semantica

## Assegnamento delle variabili

- L'assegnamento di variabili del  $2^{\Omega}$  ordine (insieme  $\mathbf{V}_2$ ) è una funzione  
 $v_2 : \mathbf{V}_2 \rightarrow \wp(\{0, 1, \dots, |w| - 1\})$ 
  - $w, v_1, v_2 \models X(x)$  se e solo se  $v_1(x) \in v_2(X)$
  - $w, v_1, v_2 \models \forall X(\varphi)$  se e solo se  $w, v'_1 \models \varphi$  per ogni  $v'_2$  con  $v'_2(Y) = v_2(Y)$ , con  $Y$  diversa da  $X$

## Esempio

- Possiamo descrivere il linguaggio  $L = \{a^{2n}, n \in \mathbb{N} \setminus \{0\}\}$

$$\exists P(\forall x( \begin{array}{l} a(x) \wedge \\ \neg P(0) \wedge \\ \forall y(y = x + 1 \Rightarrow (\neg P(x) \Leftrightarrow P(y))) \wedge \\ (last(x) \Rightarrow P(x)) \end{array}))$$

## Da MSO a FSA

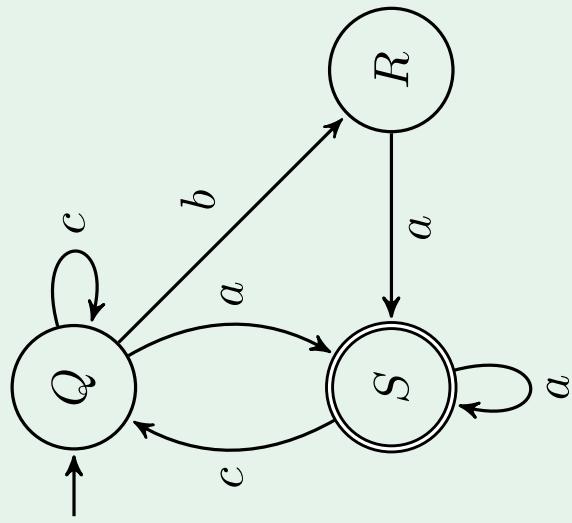
### Un approccio costruttivo

- Per ogni  $q \in Q$  dell'FSA, definiamo una variabile  $\mathbf{X}_q$  che rappresenta l'insieme di posizioni in una stringa accettata dove l'automa si trova in  $q$
- L'automa non è in due stati contemporaneamente: per ogni coppia  $\mathbf{X}_i, \mathbf{X}_j$ , abbiamo  $\neg \exists y (y \in \mathbf{X}_i \wedge y \in \mathbf{X}_j)$
- L'FSA parte da  $q_0$  equivale a dire che  $\forall x (x = 0 \Rightarrow x \in X_{q_0})$  delle posizioni dei caratteri letti dall'FSA partendo da  $q$
- Ogni transizione  $\delta(q_i, a) = q_j$  diventa:  
 $\forall x, y (y = x + 1 \Rightarrow (x \in \mathbf{X}_i \wedge a(x) \wedge y \in \mathbf{X}_j))$
- L'accettazione via  $\delta(q_i, a) \in F$  diventa:  $\forall x (last(x) \Rightarrow \bigvee_{\delta(q_i, a) \in F} (x \in \mathbf{X}_i \wedge a(x)))$

## Da FSA a MSO

Un esempio pratico

$$\exists Q, R, S ($$
  
$$\neg \exists z (\neg(Q(z) \wedge R(z)) \wedge$$
  
$$\quad \neg(Q(z) \wedge S(z)) \wedge$$
  
$$\quad \neg(R(z) \wedge S(z))) \wedge$$
  
$$\forall x (x = 0 \Rightarrow x \in Q) \wedge$$
  
$$\forall x, y (y = x + 1 \Rightarrow$$
  
$$\quad (Q(x) \wedge c(x) \wedge Q(y)) \vee$$
  
$$\quad (Q(x) \wedge b(x) \wedge R(y)) \vee$$
  
$$\quad (Q(x) \wedge a(x) \wedge S(y)) \vee$$
  
$$\quad (R(x) \wedge a(x) \wedge S(y)) \vee$$
  
$$\quad (S(x) \wedge a(x) \wedge S(y)) \vee$$
  
$$\quad (S(x) \wedge c(x) \wedge Q(y))) \wedge$$
  
$$\forall x (last(x) \Rightarrow Q(x) \wedge a(x) \vee R(x) \wedge a(x) \vee$$
  
$$S(x) \wedge a(x)))$$



## Comportamento asintotico e notazione

### Indicare i tassi di crescita

- Introduciamo una notazione per indicare il comportamento asintotico di una funzione:
  - La notazione  $\mathcal{O}$ -grande: limite asintotico superiore
  - La notazione  $\Omega$ -grande: limite asintotico inferiore
  - La notazione  $\Theta$ -grande: limite asintotico sia sup. che inf.
- Caveat pratico 1: comportamento asintotico  $\rightarrow$  per valori piccoli di  $n$  potrebbe non essere un buon modello
- Caveat pratico 2: in qualche (raro) caso valore “piccolo” per  $n$  non corrisponde alla nostra intuizione
  - Un algoritmo con complessità asintotica minore può essere più lento di uno a complessità maggiore per tutti i valori di  $n$  che ci interessano
  - Tutto funziona ... fin quando non crescono i dati in ingresso

## Proprietà di $\mathcal{O}, \Omega, \Theta$

### Proprietà di $\Theta$

- $\Theta$  è una relazione di equivalenza sull'insieme di funzioni
  - Riflessiva:  $f(n) \in \Theta(f(n))$
  - Simmetrica:  $f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$
  - Transitiva:  $f(n) \in \Theta(g(n)) \wedge h(n) \in \Theta(g(n)) \Rightarrow f(n) \in \Theta(h(n))$

### Proprietà di $\mathcal{O}, \Omega$

- Esse sono riflessive:  $f(n) \in \mathcal{O}(f(n)), f(n) \in \Omega(f(n))$
- $f(n) \in \Theta(g(n))$  se e solo se  $f(n) \in \mathcal{O}(g(n)) \wedge f(n) \in \Omega(g(n))$
- $\mathcal{O}$  e  $\Omega$  sono relazioni d'ordine parziale

# Modelli di calcolo

## Altri modelli deterministici di calcolo

- FSA: hanno sempre  $S_{\text{FSA}}(n) \in \Theta(1)$ ,  $T_{\text{FSA}}(n) \in \Theta(n)$ 
  - Tecnicamente,  $T_{\text{FSA}}(n) = n$ , leggono un carattere per mossa
- APD: hanno sempre  $S_{\text{APD}}(n) \in \mathcal{O}(n)$ ,  $T_{\text{APD}}(n) \in \Theta(n)$
- MT a nastro singolo?
  - È facile trovare una soluzione  $T_{\mathcal{M}}(n) \in \Theta(n^2)$  per il riconoscimento di  $L = \{wcw^r\}$
  - $S_{\mathcal{M}}(n)$  non potrà mai essere minore di  $\Theta(n)$
  - Non esiste un algoritmo più efficiente di  $T_{\mathcal{M}}(n) \in \Theta(n^2)$  per MT a nastro singolo  
(Intuizione: a ogni controllo di ognuna delle  $\frac{n-1}{2}$  copie di lettere, la TM scandisce una porzione di nastro che passa su  $c$ )
- In generale: MT a nastro singolo sono più potenti degli APD, ma ciò che eseguono può avere qualunque complessità spazio/temporiale!

# I teoremi di accelerazione lineare

## Teorema (Compressione dello spazio)

Se  $L$  è accettato da una MT  $\mathcal{M}$  a  $k$  nastri in  $S_{\mathcal{M}}(n)$ , per ogni  $c \in \mathbb{Q}$ ,  $c > 0$  posso costruire una MT  $\mathcal{M}'$  a  $k$  nastri con  $S_{\mathcal{M}'}(n) < c \cdot S_{\mathcal{M}}(n)$

## Schema della dimostrazione

- Scelgo un fattore di compressione  $r$  tale che  $r \cdot c > 1$
- Per ognuno degli  $i$  nastri di  $\mathcal{M}$ , considero l'alfabeto  $\Gamma_i$  e costruisco  $\Gamma'_i$  di  $\mathcal{M}'$  creando un elemento in  $\Gamma'_i$  per ogni  $s \in \Gamma_i^r$
- Costruisco l'OC di  $\mathcal{M}'$  in modo tale per cui:
  - Calcoli con i nuovi simboli sui nastri emulando le mosse di  $\mathcal{M}$  spostando le testine sui nastri di  $\mathcal{M}'$  ogni  $r$  movimenti di  $\mathcal{M}$
  - Memorizzi la posizione della testina "all'interno" dei nuovi simboli degli alfabeti di nastro  $\Gamma_i$  usando gli stati

## I teoremi di accelerazione lineare - 2

**Teorema** (Da  $k$  a 1 nastro di memoria)

Se  $L$  è accettato da una MT  $\mathcal{M}$  a  $k$  nastri in  $S_{\mathcal{M}}(n)$ , posso costruire una MT  $\mathcal{M}'$  a 1 nastro (non nastro singolo) con  $S_{\mathcal{M}'}(n) = S_{\mathcal{M}}(n)$  (concateno i contenuti dei  $k$  nastri su uno solo)

**Teorema**

Se  $L$  è accettato da una MT  $\mathcal{M}$  a  $k$  nastri in  $S_{\mathcal{M}}(n)$ , per ogni  $c \in \mathbb{Q}, c > 0$  posso costruire una MT  $\mathcal{M}'$  a  $k = 1$  nastri con  $S_{\mathcal{M}'}(n) < c \cdot S_{\mathcal{M}}(n)$

## I teoremi di accelerazione lineare - 3

### Teorema

Se  $L$  è accettato da una MT  $\mathcal{M}$  a  $k$  nastri in  $T_{\mathcal{M}}(n)$ , per ogni  $c \in \mathbb{Q}, c > 0$  posso costruire una MT  $\mathcal{M}'$  a  $k+1$  nastri con  $T_{\mathcal{M}'}(n) = \max(n+1, c \cdot T_{\mathcal{M}}(n))$

### Schema di dimostrazione

- Approccio simile alla complessità spaziale: codifichiamo in modo compresso i simboli dell'alfabeto di  $\mathcal{M}$  e calcoliamo sui simboli compressi
- Dobbiamo considerare che la compressione è fatta a runtime: il minimo costo per effettuare la compressione è lineare nel numero di simboli da comprimere
- Comprimendo  $r$  simboli in uno, nel caso pessimo, possono servirmi 3 mosse di  $\mathcal{M}'$  per emularne  $r+1$  di  $\mathcal{M}$

## Conseguenze pratiche

L'unico "pasto gratis" è lo speedup lineare

- Lo schema di dimostrazione usato vale anche per i calcolatori a-la Von Neumann
  - Equivale a cambiare la dimensione della word della macchina ( $32b \rightarrow 64b$ ) o, equivalentemente, ad usare operazioni vettoriali (SSE/AVX, Neon, AltiVec)
- Possiamo avere speedup *lineari* arbitrariamente grandi, aumentando il parallelismo fisico (stanti i limiti della termodinamica/trasmissione dei segnali)
- Miglioramenti più che lineari nel tempo di calcolo possono essere ottenuti solo *cambiando algoritmo*
  - Concepire/utilizzare algoritmi efficienti è di gran lunga più efficace della forza bruta

# Criterio di costo logaritmico

## Contare i singoli bit

- Copiare/spostare/scrivere/leggere un intero  $i$  costa tanto quanto il suo numero di cifre in base  $b$ :  $\log_b(i) = \Theta(\log(i))$ 
  - Con  $b = 2$  il costo è il numero di bit usati per rappresentare  $i$
- Il costo delle operazioni aritmetico/logiche elementari dipende dall'operazione (definiamo  $d = \log_2(i)$ )
  - Addizioni, sottrazioni, op. al bit  $\Theta(d)$
  - Moltiplicazioni: metodo scolastico  $\Theta(d^2)$ 
    - Primo miglioramento:  $\Theta(d^{\log_2(3)}) \approx \Theta(d^{1.58})$
    - Ulteriore miglioramento:  $\Theta(d \log(d) \log(\log(d)))$
    - Miglior algoritmo attuale:  $\Theta(d \log(d))$
  - Divisioni: metodo scolastico  $\Theta(d^2)$ , o al costo dell'algoritmo di moltiplicazione scelto per  $\log^2(d)$
  - JUMP e HALT sono a costo costante, così come le Jcc (il valore del codice di condizione è nella PSW, già calcolato)

# Rapporti tra criteri di costo

## Ri-analizzando

- Riconoscere  $L = \{wcw^R\}$  è  $\Theta(n \log(n))$  (colpa del contatore)
- Ricerca binaria:  $\Theta(\log(n)^2)$  (colpa degli indici)

## Quale criterio scegliere?

- Se la dimensione di ogni singolo elemento in ingresso non varia significativamente nell'esecuzione dell'algoritmo (= stesso numero di cifre per tutta l'esecuzione): costo costante
- Nel caso in cui ci sia un significativo cambio nella dimensione dei singoli elementi in ingresso (= il numero di cifre cresce in modo significativo): costo logaritmico

# Rapporti tra complessità con diversi modelli di calcolo

Modelli di calcolo diversi → diversa efficienza

- Risolvere lo stesso problema con macchine diverse può dare luogo a complessità diverse
- Non esiste un modello migliore in assoluto

Tesi di correlazione polinomiale

- Sotto “ragionevoli” ipotesi di criteri di costo, se un problema è risolvibile da  $\mathcal{M}$  in  $T_{\mathcal{M}}(n)$ , allora è risolvibile da un qualsiasi altro modello (Turing-completo)  $\mathcal{M}'$  in  $T_{\mathcal{M}'}(n) = \pi(T_{\mathcal{M}}(n))$  dove  $\pi(\cdot)$  è un opportuno polinomio
- Dimostriamo il teorema di correlazione (temporale) polinomiale tra MT e RAM

# Correlazione temporale tra MT a $k$ nastri e RAM

RAM simula MT a  $k$  nastri: lettura

- Lettura del il blocco 0 e dello stato ( $\Theta(k)$ ) mosse)
- Lettura dei valori sui nastri in corrispondenza delle testine ( $\Theta(k)$ ) accessi indiretti)

RAM simula MT a  $k$  nastri: Scrittura

- Scrittura dello stato ( $\Theta(1)$ )
- Scrittura delle celle dei nastri ( $\Theta(k)$ ) accessi indiretti)
- Scrittura nel blocco 0 per aggiornare le posizioni delle  $k$  testine ( $\Theta(k)$ )

Complessità dell' emulazione

- RAM emula una mossa della MT con un  $k$  di mosse:  $T_{RAM}(n) = \Theta(T_M(n))$   
 $(= \Theta(T_M(n) \log(T_M(n))))$  costo log.)

## Correlazione temporale tra MT a $k$ nastri e RAM

### Concludendo

- La MT impiega al più  $\Theta(T_{RAM}(n))$  per simulare una mossa della RAM
- Se la RAM ha complessità  $T_{RAM}(n)$  essa effettua al più  $T_{RAM}(n)$  mosse (ogni mossa costa almeno 1)
- La simulazione completa della RAM da parte della MT costa al più  $\Theta((T_{RAM}(n))^2)$  il legame tra  $T_{RAM}(n)$  e  $T_{MT}(n)$  tra è polinomiale

## Pseudocodice - Sintassi

### Procedure, assegnamenti, costrutti di controllo

- Ogni algoritmo è rappresentato con una procedura (= funzione che modifica i dati in input, non ritorna nulla)
- Operatori: Aritmetica a singola precisione come in C, assegnamento ( $\leftarrow$ ), e confronti ( $<$ ,  $\leq$ ,  $=$ ,  $\geq$ ,  $>$ ,  $\neq$ )
- Commenti mono-riga con  $\triangleright$ , blocchi dati dall'indentazione
- Costrutti di controllo disponibili: `while`, `for`, `if-else`
- Tutte le variabili sono locali alla procedura descritta
- Il tipo delle variabili non è esplicito, va inferito dal loro uso

# Pseudocodice

## Tipi di dato aggregato

- Ci sono gli array, notazione identica al C, indici iniziano da 1
- Sono disponibili anche i sotto-array (slices) come in Fortran, Matlab, Python
  - $A[i..j]$  è la porzione di array che inizia dall' $i$ -esimo elemento e termina al  $j$ -esimo
- Sono presenti aggregati eterogenei (= strutture C)
  - L'accesso a un campo è effettuato tramite l'operatore .  $A.campo1$  è il campo di nome campo1 della struttura A
  - Diversamente dal C, una variabile di tipo aggregato è un puntatore alla struttura
  - Un puntatore non riferito ad alcuna struttura ha valore NIL

## Attenzione all'aliasing

- 1  $y \leftarrow x$
- 2  $x.f \leftarrow 3 //$  dopo questa riga anche  $y.f$  vale 3

## Pseudocodice - Convenzioni

### Passaggio parametri

- Il passaggio di parametri ad una procedura viene effettuato:
  - Nel caso di tipi non aggregati: per copia
  - Nel caso di tipi aggregati: per riferimento
- Comportamento identico al C per tipi non aggregati ed array
  - Diverso per le strutture (in C sono passate per copia, uguale a quello di Java)

### Modello di esecuzione

- Lo pseudocodice è eseguito dalla macchina RAM
- Assunzione fondamentale: un singolo statement di assegnamento tra tipi base è tradotto in un numero costante  $k$  di istruzioni dell'assembly RAM

# Ricorsione e complessità

Come calcolare la complessità di algoritmi ricorsivi?

- Ci sono algoritmi con complessità non immediatamente esprimibile in forma chiusa
- Il caso tipico sono algoritmi *divide et impera*:
  - Divido il problema in  $a$  sottoproblemi con dimensione dell'input pari a una frazione  $\frac{1}{b}$  dell'originale,  $n$
  - Quando  $n$  è piccolo a sufficienza, risolvo in tempo costante (caso limite  $n = 0$ )
  - Ricombino le soluzioni dei sottoproblemi
  - Indichiamo con  $D(n)$  il costo del suddividere il problema e con  $C(n)$  il costo di combinare le soluzioni
- Esprimiamo il costo totale  $T(n)$  con un'*equazione di ricorrenza* (o *ricorrenza*):

$$T(n) = \begin{cases} \Theta(1) & \text{se } n < c \\ D(n) + aT\left(\frac{n}{b}\right) + C(n) & \text{altrimenti} \end{cases}$$

# Ricorsione e complessità

Come risolvere le ricorrenze?

- Sono possibili 3 tecniche principali:
  - Sostituzione
  - Esame dell'albero di ricorsione
  - Teorema dell'esperto (master theorem)
- Usiamo come caso di studio la ricerca binaria:
  - Formuliamo il problema di cercare in un vettore lungo  $n$  come quello di cercare nelle sue metà superiori e inferiori
  - Costo di suddivisione (calcolo indici) costante  $D(n) = \Theta(1)$
  - Costo di ricombinazione costante: sappiamo che una delle due metà non contiene per certo l'elemento cercato  $C(n) = \Theta(1)$
  - Complessità espressa come  $T(n) = \Theta(1) + T(\frac{n}{2}) + \Theta(1)$

# Metodo di sostituzione

## Ipotesi e dimostrazione

- Il metodo di sostituzione si articola in tre fasi:
  - 1 Intuire una possibile soluzione
  - 2 Sostituire la presunta soluzione nella ricorrenza
  - 3 Dimostrare per induzione che la presunta soluzione è tale per l'equazione/disequazione alle ricorrenze
- Ad esempio, con la complessità della ricerca binaria:  $T(n) = \Theta(1) + T\left(\frac{n}{2}\right) + \Theta(1)$ 
  - 1 Intuizione: penso sia  $T(n) = \mathcal{O}(\log(n))$  ovvero  $T(n) \leq c \log(n)$
  - 2 Devo dimostrare:  $T(n) = \Theta(1) + T\left(\frac{n}{2}\right) + \Theta(1) \leq c \cdot \log(n)$
  - 3 Considero vero per ipotesi di induzione  $T\left(\frac{n}{2}\right) \leq c \cdot \log\left(\frac{n}{2}\right)$  in quanto  $\frac{n}{2} < n$  e sostituisco nella (2) ottenendo :  
$$T(n) \leq c \cdot \log\left(\frac{n}{2}\right) + \Theta(k) = c \cdot \log(n) - c \log(2) + \Theta(k) \leq c \log(n)$$

# Metodo dell'albero di ricorsione

## Espandere le chiamate ricorsive

- L'albero di ricorsione fornisce un aiuto per avere una congettura da verificare con il metodo di sostituzione, o un appiglio per calcolare la complessità esatta
- È una rappresentazione delle chiamate ricorsive, con la loro complessità
- Ogni chiamata costituisce un nodo in una sorta di albero genealogico: i chiamati appaiono come figli del chiamante
- Ogni nodo contiene il costo della chiamata, senza contare quello dei discendenti
- Rappresentiamo l'albero di  $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$

## Metodo dell'albero di ricorsione

### Esplendendo completamente

- L'albero ha la ramificazione a profondità massima posta all'estrema destra del disegno precedente
- Sappiamo che essa ha profondità  $k$  che ricaviamo ponendo  $\frac{2^k}{3^k}n = 1$  (il  $k$ -esimo pronipote a dx)  
$$\rightarrow 2^k n = 3^k \rightarrow \log_3(2^k n) = k = \log_3(2^k) + \log_3(n) = \log_3(n) + \frac{\log_2(2^k)}{\log_2(3)}$$
 da cui  
abbiamo che  $(\log_2(3) - 1)k = \log_3(n) \rightarrow k = c \log_3(n)$
- Il costo pessimo per il contributo di un dato livello
- Congetturiamo che  $T(n) = \Theta(n \log(n))$
- Dimostriamolo mostrando che  $T(n) = \mathcal{O}(n \log(n))$  e  $T(n) = \Omega(n \log(n))$

## Metodo dell'albero di ricorsione

$$T(n) = \mathcal{O}(n \log(n))$$

- Per hp. di induzione abbiamo sia che  $T\left(\frac{n}{3}\right) \leq c_1\left(\frac{n}{3} \log\left(\frac{n}{3}\right)\right)$  sia che  $T\left(\frac{2n}{3}\right) \leq c_2\left(\frac{2n}{3} \log\left(\frac{2n}{3}\right)\right)$  (dato che  $\frac{2}{3}n < n$  e  $\frac{1}{3}n < n$ )
- Sostituendo abbiamo  
$$T(n) \leq c_1\left(\frac{n}{3} \log\left(\frac{n}{3}\right)\right) + c_2\left(\frac{2n}{3} \log\left(\frac{2n}{3}\right)\right) + n = c_1\left(\frac{n}{3}(\log(n) - \log(3)) + c_2\left(\frac{2n}{3}(\log(n) - \log(3) + \log(2))\right) + c_3n = c_4n \log(n) - c_5n + c_3n \leq c_4n \log(n)\right)$$
 per una scelta opportuna delle costanti  $c_4, c_5, c_6$

$$T(n) = \Omega(n \log(n))$$

- Hp ind.  $T\left(\frac{n}{3}\right) \geq c_1\left(\frac{n}{3} \log\left(\frac{n}{3}\right)\right)$ ,  $T\left(\frac{2n}{3}\right) \geq c_2\left(\frac{2n}{3} \log\left(\frac{2n}{3}\right)\right)$
- Sostituendo  $T(n) \geq c_4n \log(n) - c_5n + c_6n \geq c_4n \log(n)$

# Teorema dell'esperto (Master theorem)

## Uno strumento efficace per le ricorsioni

- Il teorema dell'esperto è uno strumento per risolvere buona parte delle equazioni alle ricorrenze.
- Affinchè sia applicabile, la ricorrenza deve avere la seguente forma:  
$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \text{ con } a \geq 1, b > 1$$
- L'idea di fondo è quella di confrontare  $a^{\log_b(n)} = a^{\frac{\log_a(n)}{\log_a(b)}} = n^{\log_b(a)}$  (costo totale delle foglie dell'AdR) con  $f(n)$  (il costo della sola radice dell'AdR)
- Le ipotesi del teorema dell'esperto sono le seguenti:
  - $a$  deve essere costante e  $a \geq 1$  (almeno 1 sotto-problema per chiamata ricorsiva)
  - $f(n)$  deve essere sommata, non sottratta o altro a  $aT\left(\frac{n}{b}\right)$
  - Il legame tra  $n^{\log_b(a)}$  e  $f(n)$  deve essere polinomiale
- Se queste ipotesi sono valide, è possibile ricavare informazione sulla complessità a seconda del caso in cui ci si trova

## Master Theorem

### Caso 1

- Nel primo caso  $f(n) = \mathcal{O}(n^{\log_b(a)-\epsilon})$  per un qualche  $\epsilon > 0$
- La complessità risultante è  $T(n) = \Theta(n^{\log_b(a)})$
- Intuitivamente: il costo della ricorsione “domina” quello della singola chiamata
- Esempio:  $T(n) = 9T(\frac{n}{3}) + n$
- Confrontiamo:  $n^1 = n^{\log_3(9)-\epsilon} \Rightarrow \epsilon = 1 \quad \checkmark$
- Otteniamo che la complessità è:  $\Theta(n^{\log_3(9)}) = \Theta(n^2)$

# Master Theorem

## Caso 2

- Nel secondo caso abbiamo che  $f(n) = \Theta(n^{\log_b(a)})$
- La complessità risultante della ricorrenza è  $T(n) = \Theta(n^{\log_b(a)} \log(n))$
- Intuitivamente: il contributo della ricorsione e quello della singola chiamata differiscono per meno di un termine polinomiale
- Esempio:  $T(n) = T\left(\frac{n}{3}\right) + \Theta(1)$
- Confrontiamo:  $\Theta(1) = \Theta(n^{\lfloor \log_3(1) \rfloor})$  è vero ?
  - Sì:  $\Theta(1) = \Theta(n^0)$  ✓
- La complessità risultante è  $\Theta(n^{\log_3(1)} \log(n)) = \Theta(\log(n))$

# Master Theorem

## Caso 3

- In questo caso abbiamo che  $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ ,  $\epsilon > 0$
- Cond. Necessaria: vale che:  $af(\frac{n}{b}) < cf(n)$  per un qualche valore di  $c < 1$
- Se le ipotesi sono rispettate,abbiamo che  $T(n) = \Theta(f(n))$
- Intuitivamente: il costo della singola chiamata è più rilevante della ricorsione
- Esempio:  $T(n) = 8T(\frac{n}{3}) + n^3$
- Confrontiamo  $n^3 = \Omega(n^{\log_3(8)+\epsilon}) \Rightarrow \epsilon = 3 - \log_3(8) > 0$  ✓
- Controlliamo se  $8f(\frac{n}{3}) = \frac{8}{3^3}n^3 < cn^3$  per un qualche  $c < 1$ ?
  - Sì, basta prendere  $c$  in  $(1 - (\frac{8}{3^3}); 1)$  ✓
- La complessità dell'esempio è:  $\Theta(n^3)$

## Riassumendo

Un confronto tra ordinamenti per confronto

Algoritmo	Stabile?	$T(n)$ (caso pessimo)	$T(n)$ (caso ottimo)	$S(n)$
Insertion	✓	$\Theta(n^2)$	$\Theta(n)$	$O(1)$
Merge	✓	$\Theta(n \log(n))$	$\Theta(n \log(n))$	$\Theta(n)$
Quick	✗	$\mathcal{O}(n^2)$	$\Omega(n \log(n))$	$O(1)$

- Non è possibile essere più veloci usando algoritmi di ordinamento *per confronto*
- C'è modo di fare meglio ordinando senza confrontare tra elementi?

# Analisi critica di strutture note

## Vettori

- Un vettore è una struttura dati compatta in memoria in cui si accede direttamente ad ogni elemento, data la sua posizione
- Se il vettore di lunghezza  $n$  non è ordinato:
  - ricerca, minimo, massimo, successore sono  $\mathcal{O}(n)$
  - inserimento e cancellazione costano  $\mathcal{O}(n)$  (la cancellazione può essere ridotta a  $\mathcal{O}(1)$  usando dei simboli di "cella vuota")
- Se il vettore di lunghezza  $n$  è ordinato:
  - minimo e massimo:  $\Theta(1)$ , ricerca e successore  $\Theta(\log(n))$
  - inserimento e cancellazione costano  $\mathcal{O}(n)$
- Inserimenti in vettore pieno:
  - sono rifiutati (tenendo un conteggio degli elementi):  $\mathcal{O}(1)$
  - causano una riallocazione  $\mathcal{O}(n)$ : (causa copia degli elementi esistenti)

## Analisi critica di strutture note

### Liste semplicemente connesse

- Una lista semplice stocca gli elementi sparsi in memoria: ogni elemento ha un riferimento al successivo (i.e., puntatore)
- Se la lista di lunghezza  $n$  non è ordinata:
  - ricerca, minimo, massimo, successore sono  $\mathcal{O}(n)$
  - inserimento:  $\mathcal{O}(1)$ , cancellazione:  $\mathcal{O}(n)$  se l'elemento va trovato,  $\mathcal{O}(1)$  se si ha un riferimento alla posizione di inserimento
- Se la lista di lunghezza  $n$  è ordinata:
  - uno dei due tra minimo e massimo è  $\Theta(1)$  l'altro  $\Theta(n)$
  - Con puntatore accessorio all'ultimo elemento: entrambi  $\Theta(1)$
  - ricerca e successore sono  $\mathcal{O}(n)$
  - inserimento:  $\mathcal{O}(n)$ , cancellazione:  $\mathcal{O}(n)$

# Operazioni tipiche su strutture dati

## Interrogare la struttura

- **Search( $S, k$ ):** restituisce il riferimento a  $e$  con  $e.key=k$  in  $S$ , NIL se  $k$  non è contenuto in  $S$
- **Minimum( $S$ ):** se le chiavi sono ordinate, restituisce  $e$  con la chiave minima
- **Maximum( $S$ ):** come sopra, ma la chiave ha valore massimo
- **Successor( $S, k$ ):** restituisce  $e$  t.c.  $e.key$  è la più piccola tra le chiavi  $>k$
- **Predecessor( $S, k$ ):** come sopra, ma considerando il predecessore

## Modificare la struttura

- **Insert( $S, e$ ):** Inserisce un oggetto  $e$  nella struttura
- **Delete( $S, e$ ):** Cancella un oggetto  $e$  dalla struttura

# Pile (Stack)

## Una struttura dati familiare

- Una pila è una struttura dati con le seguenti operazioni:
  - Push(S,e): aggiunge l'elemento in cima alla pila
  - Pop(S): restituisce l'elemento in cima alla pila cancellandolo
  - Empty(S): restituisce true se la pila è vuota
- Questa struttura dati astratta può essere realizzata usando una lista semplicemente connessa o un vettore

## Realizzazione con lista

- Lo stoccaaggio dati è nella lista, le operazioni diventano:
  - Push(S,e): inserisci in testa alla lista  $\mathcal{O}(1)$
  - Pop(S): restituisci il primo elemento della lista, cancellandolo dalla stessa  $\mathcal{O}(1)$
  - Empty(S): controlla se il successore della testa è NIL:  $\mathcal{O}(1)$

## Code (Queues)

### Struttura ed operazioni

- Una coda è una struttura dati con le seguenti operazioni:
  - Enqueue (Q, e) : aggiunge e alla fine della coda
  - Dequeue (Q) : restituisce l'elemento all'inizio della coda, cancellandolo dalla stessa
  - Empty (Q) : restituisce true se la coda è vuota
- Come nel caso della pila, è possibile realizzare una coda sia con una lista che con un vettore

## Mazzo o coda a due fini (Deque)

### Struttura dati

- La struttura dati si comporta come un mazzo di carte, di cui ognuna contiene un elemento
- E' possibile aggiungere sia in testa che in coda alla struttura:
  - PushFront(Q, e): inserisci l'elemento e in testa al mazzo
  - PushBack(Q, e): inserisci l'elemento e in coda al mazzo
  - PopFront(Q): restituisci l'elemento in testa, cancellandolo
  - PopBack(Q): restituisci l'elemento in coda, cancellandolo
  - Empty(S): Restituisci true se il mazzo è vuoto

# Dizionari

## Un primo approccio

- Nel caso in cui le *possibili* chiavi siano un numero limitato un'implementazione di un dizionario è un vettore di puntatori
- Le chiavi vengono usate come indice del vettore
- Le operazioni sul dizionario sono implementate come:
  - `Insert(D,e): D[e.key] ← e`
  - `Delete(D,e): D[e.key] ← NIL`
  - `Search(D,e.key): return D[e.key]`
- Complessità computazionale:  $\Theta(1)$  per tutte le azioni
- Complessità spaziale:  $\mathcal{O}(|D|)$ , con  $D$  il dominio delle chiavi
  - Estremamente oneroso se il dominio è molto ampio

# Tabelle Hash

## Efficienza

- Se il calcolo di  $h(\cdot)$  è  $\mathcal{O}(1)$  la tabella di hash ideale ha la stessa efficienza temporale del dizionario realizzato con un vettore di  $|\mathbf{D}|$  puntatori

## Il problema delle collisioni

- Idealmente,  $h$  mappa ogni chiave su di un distinto elemento del suo codominio
  - Impossibile! Per costruzione  $|\mathbf{D}| \gg m$  (specie se  $|\mathbf{D}| = \infty$ )
- Chiamiamo *collusione* i casi in cui, date due chiavi  $k_1, k_2$ ; con  $k_1 \neq k_2$  abbiamo che  $h(k_1) = h(k_2)$ 
  - Dobbiamo gestire le collisioni, oppure due elementi con chiavi distinte finiranno allo stesso indirizzo

## Efficienza computazionale

### Stime di costo

- Caso pessimo: tutti gli elementi collidono dando origine ad una lista (open hashing) lunga  $m$  elementi: Insert/Delete/Search in  $\mathcal{O}(m)$
- Chiamiamo fattore di carico  $\alpha = \frac{n}{m}$ ,  $0 \leq \alpha \leq \frac{|\mathbf{D}|}{m}$ 
  - $\alpha$  può essere più grande di 1  $\rightarrow$  tutti i bucket sono pieni

### Ipotesi di Hashing Uniforme Semplice (IHUS)

- Una opportuna scelta di  $h$  fa sì che ogni chiave abbia la stessa probabilità  $\frac{1}{m}$  di finire in una qualsiasi delle  $m$  celle
- Come fare “la scelta opportuna”? Dipende dalla distribuzione delle chiavi

## Gestione delle collisioni - 2

### Indirizzamento aperto (a.k.a., open addressing, closed hashing)

- In caso di collisione si seleziona secondo una regola deterministica l'indirizzo di un altro bucket di destinazione (procedimento di ispezione)
- Nel caso non si trovino bucket vuoti:
  - L'inserimento semplicemente fallisce  $\Theta(m)$
  - Si rialloca una tabella più grande, vuota, e si ri-inseriscono tutti gli elementi della vecchia nella nuova (ricalcolando la loro hash, re-hashing), incluso il nuovo  $\Theta(n)$
- Si modifica la procedura di ricerca, affinchè, se l'elemento non viene trovato nel suo bucket, essa effettui la stessa ispezione
- La cancellazione è effettuata inserendo un opportuno valore (tombstone) che non corrisponde ad alcuna chiave

## Procedure di ispezione

### Ispezione lineare (Linear probing) e clustering

- Il metodo di ispezione più semplice è l'*ispezione lineare*
  - Dato  $h(k, 0) = a$  il bucket dove avviene la collisione al primo ( $i = 0$ ) tentativo di inserimento, si sceglie  $h(k, i) = a + c \cdot i \bmod m$  come bucket candidato per l' $i$ -esimo inserimento
  - Problema: se ci sono molte collisioni su un dato bucket, peggiorerà la probabilità di collisione in tutte le vicinanze
    - Il fenomeno è detto di *clustering primario* delle collisioni
    - Per alcune scelte di  $h$ , il peggiorare delle prestazioni dovuto al clustering dell'ispezione lineare è molto forte
    - È possibile avere clustering di dimensione logaritmica nella dimensione della tabella, effettuando rehashing molto prima che sia piena

## Procedure di ispezione

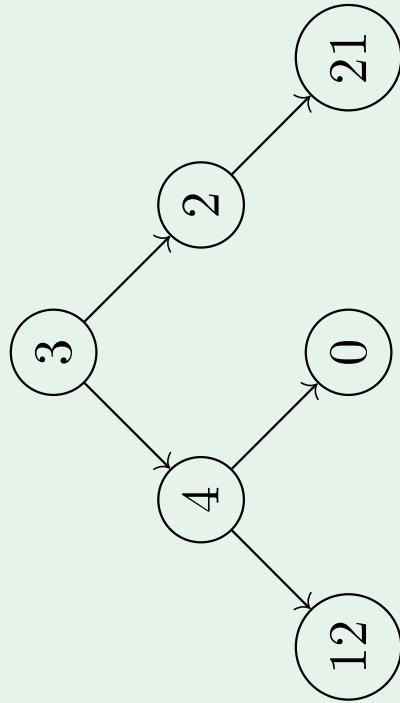
### Ispezione quadratica (Quadratic probing)

- Per mitigare il fenomeno del clustering è possibile utilizzare il criterio di ispezione quadratica:  $h(k, i) = a + c_1 i + c_2 i^2 \bmod m$ 
  - Viene evitato il clustering banale nell'intorno di alcuni elementi
  - Non è più garantito a priori che la sequenza di ispezione tocchi tutte le celle: potrei dover fare rehashing a tabella non piena
- Rimuove il clustering primario
- Chiavi con la stessa posizione iniziale generano ancora clustering: hanno la stessa sequenza di ispezione (clustering secondario)

# Visita di un albero

## Visita *in-order* (*in-order*)

- Nella visita in ordine si visita prima il sottoalbero sx, quindi la radice, infine il sottoalbero dx



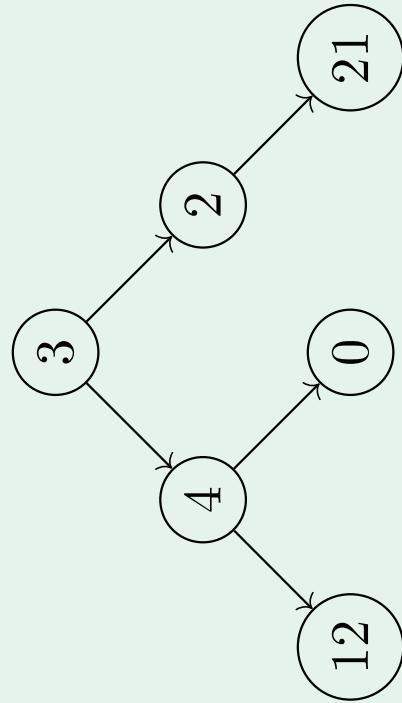
```
INORDER( $T$ )
1 INORDER( $T.left$ )
2 PRINT( $T.key$ )
3 INORDER( $T.right$ )
4 return
```

- INORDER dell'esempio stampa: 12, 4, 0, 3, 2, 21
- Complessità:  $\Theta(n)$ , tocca una sola volta ogni nodo

# Visita di un albero

## Visita anticipata (*preorder*)

- Nella visita in ordine si visita prima la radice, quindi il sottoalbero sx, infine il sottoalbero dx



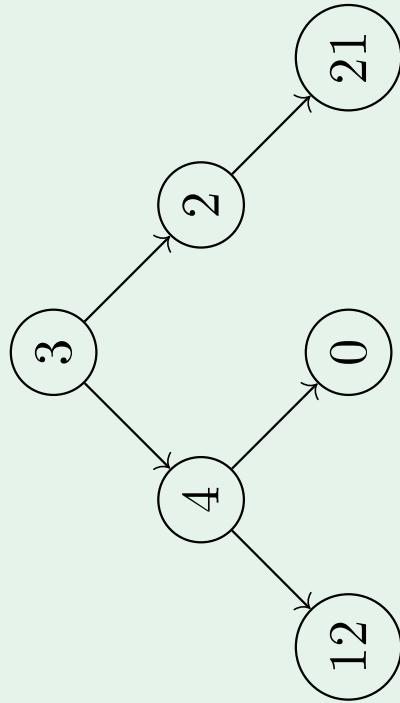
```
PREORDER( $T$ )
1 PRINT( $T.key$ )
2 PREORDER( $T.left$ )
3 PREORDER( $T.right$ )
4 return
```

- PREORDER dell'esempio stampa: 3, 4, 12, 0, 2, 21
- Complessità:  $\Theta(n)$ , tocca una sola volta ogni nodo

# Visita di un albero

## Visita posticipata (*postorder*)

- Nella visita in ordine si visita prima il sottoalbero sx, poi il sottoalbero dx e infine la radice



```
POSTORDER( $T$ )
1 POSTORDER( $T.left$ )
2 POSTORDER( $T.right$ )
3 PRINT( $T.key$ )
4 return
```

- POSTORDER dell'esempio stampa: 12, 0, 4, 21, 2, 3
- Complessità:  $\Theta(n)$ , tocca una sola volta ogni nodo

# Alberi binari di ricerca (BST)

## Definizione

- Uno degli usi più comuni degli alberi binari è utilizzare quelli per cui è valida una data relazione tra le chiavi
- Un albero binario è un *albero binario di ricerca* se per ogni suo nodo  $x$  valgono:
  - Se  $y$  è contenuto nel sottoalbero sinistro di  $x$ ,  $y.key \leq x.key$
  - Se  $y$  è contenuto nel sottoalbero destro di  $x$ ,  $y.key \geq x.key$
- Inserimenti e cancellazioni devono preservare la proprietà
- Una visita in-ordine del BST stampa le chiavi in ordine

# Operazioni su BST

## Ricerca

- La struttura dei BST li rende naturali candidati per una ricerca efficace degli elementi per chiave

RICERCA( $T, x$ )

```
1  if  $T = \text{NIL}$  or  $T.key = x.key$ 
2    return  $T$ 
3  if  $T.key < x.key$ 
4    return RICERCA( $T.right, x$ )
5  else return RICERCA( $T.left, x$ )
6
```

- Complessità:  $\mathcal{O}(h)$  con  $h$  l'altezza dell'albero
- Nel caso ottimo (albero "ben bilanciato") diventa  $\mathcal{O}(\log(n))$
- Nel caso pessimo (albero degenere in lista) è  $\mathcal{O}(n)$

## Operazioni su BST

### Minimo e massimo

- L'elemento con chiave minima (massima) è quello più a sinistra (destra) del BST

MIN( $T$ )

```
1   cur ←  $T$ 
2   while  $cur.left \neq \text{NIL}$ 
3      $cur \leftarrow cur.left$ 
4   return  $cur$ 
```

MAX( $T$ )

```
1   cur ←  $T$ 
2   while  $cur.right \neq \text{NIL}$ 
3      $cur \leftarrow cur.right$ 
4   return  $cur$ 
```

- Complessità:  $\mathcal{O}(h)$  con  $h$  l'altezza dell'albero

# Operazioni su BST

## Successore

- Lo pseudocodice per la ricerca del successore è il seguente:

```
SUCCESSORE(x)
1 if x.right ≠ NIL
2   return MIN(x.right)
3 y ← x.p
4 while y ≠ NIL and y.right = x
5   x ← y
6   y ← y.p
7 return y
```

- Complessità nel caso 1: la stessa del calcolo del minimo:  
 $\mathcal{O}(h)$
- Complessità nel caso 2: caso pessimo, *x* è la foglia più distante dalla radice,  $\mathcal{O}(h)$

# Operazioni su BST

## Inserimento – Pseudocodice

INSERISCI( $T, x$ )

```
1   pre  $\leftarrow$  NIL
2   cur  $\leftarrow T.root$ 
3   while cur  $\neq$  NIL
4       pre  $\leftarrow$  cur
5       if x.key  $<$  cur.key
6           cur  $\leftarrow cur.left$ 
7       else cur  $\leftarrow cur.right$ 
8       x.p  $\leftarrow pre$ 
9       if pre = NIL
10      T.root  $\leftarrow x$ 
11      elseif x.key  $<$  pre.key
12          pre.left  $\leftarrow x$ 
13      else pre.right  $\leftarrow x$ 
```

- Le righe 3–7 effettuano la ricerca della posizione di inserimento nell'albero
- Le righe 8–13 effettuano l'inserimento vero e proprio
- Complessità: la stessa della ricerca  $\mathcal{O}(h)$  più una porzione a tempo costante (inserimento)

# Analisi di complessità

## Sommario

- Tutte le operazioni sono  $\mathcal{O}(h)$  con  $h$  l'altezza del BST
- Nel migliore dei casi  $h = \log(n)$  (albero completo o quasi completo), nel peggiore  $h = n$  (lista)
- È critico per avere buone prestazioni mantenere il BST il più possibile vicino al caso ottimo
- Si può dimostrare che l'altezza attesa di un BST è  $\mathcal{O}(\log(n))$  se le chiavi inserite hanno distribuzione uniforme
- Volendo un metodo deterministico ci serve una definizione di albero *ben bilanciato*

# Alberi rosso-neri

## Struttura e definizione

- Un albero rosso-nero è un BST i cui nodi sono dotati di un attributo aggiuntivo, detto *colore*  $\in \{\text{rosso}, \text{nero}\}$ , e soddisfacente le seguenti 5 proprietà:
  - 1 Ogni nodo è rosso o nero
  - 2 La radice è nera
  - 3 Le foglie sono nere
  - 4 I figli di un nodo rosso sono entrambi neri
  - 5 Per ogni nodo dell'albero, tutti i cammini dai suoi discendenti alle foglie contenute nei suoi sottoalberi hanno lo stesso numero di nodi neri
- Chiamiamo, per comodità, altezza nera (*black height*) di un nodo  $x$  il valore  $bh(x)$  pari al numero di nodi neri, escluso  $x$  se è il caso, nel percorso che va da  $x$  alle foglie

## Alberi rosso-neri

### Azioni sugli alberi rosso-neri

- Tutte le operazioni che non vanno a modificare la struttura dell'albero sono identiche ai BST: RICERCA, MIN, MAX, SUCCESSORE, PREDECESSORE
- Le operazioni di INSERISCI e CANCELLA hanno necessità di mantenere le proprietà degli alberi rosso-neri
  - Idea di massima: opero come se si trattasse di un BST generico, dopodichè compenso le eventuali violazioni
- È necessario essere in grado di ri-bilanciare l'albero con modifiche solamente locali (no ricostruzione dell'albero)

## Alberi rosso-neri

**Teorema** (Proprietà di buon bilanciamento)

*Un albero RB con  $n$  nodi interni ha altezza massima  $2 \log(n + 1)$*

**Dimostrazione - 1**

- Dim. che un sottoalbero con radice  $x$  ha almeno  $2^{bh(x)} - 1$  nodi interni, per induzione sull'altezza del sottoalbero
  - Caso base (altezza 0):  $x$  è una foglia, il sottoalbero contiene almeno  $2^{bh(x)} - 1 = 2^0 - 1 = 0$  nodi interni
  - Passo: dato  $x$ , entrambi i suoi figli hanno altezza nera  $bh(x)$  o  $bh(x) - 1$ . Dato che l'altezza dei figli è minore di quella di  $x$  (per hp. ind.) i loro sottoalberi hanno almeno  $2^{bh(x)-1} - 1$  nodi interni. L'albero radicato in  $x$  contiene quindi almeno  $2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 2 = 2^{bh(x)}$  nodi

## Alberi rosso-neri

### Dimostrazione.

- Per la proprietà 4, almeno metà dei nodi su un qualunque percorso radice-foglia sono neri
- L'altezza nera della radice è almeno  $\frac{h}{2}$ ; per quanto detto prima il sottoalbero radicato in essa contiene almeno  $2^{\frac{h}{2}} - 1$  nodi
- $n \geq 2^{\frac{h}{2}} - 1$ , risolvendo per  $h \rightarrow h \leq 2 \log(n + 1)$



### Conseguenze

- Le operazioni che restano invariate tra RB-trees e BST (RICERCA, MAX, MIN, SUCCESSORE) sono  $\mathcal{O}(2 \log(n + 1))$
- Se riesco a riparare le violazioni in maniera efficiente ho anche INSERISCI e CANCELLA) in  $\mathcal{O}(2 \log(n + 1))$

## Alberi rosso-neri - Inserimento

### Analisi di complessità

- Nei casi 2 e 3 la procedura RIPARARBINSERISCI( $z$ ) deve solo effettuare un cambio di colori locale e 2 o 1 rotazioni
  - Tutte queste operazioni sono  $\mathcal{O}(k)$
- Nel caso 1 continua analizzando il nonno del nodo corrente
  - Caso pessimo: il ciclo itera numero di volte pari a metà dell'altezza dell'intero albero
- L'intera riparazione prende al più  $\mathcal{O}(\log(n)) \rightarrow$  l'inserimento, comprensivo di riparazione, in alberi RB è  $\mathcal{O}(\log(n))$

## Operazioni su alberi RB

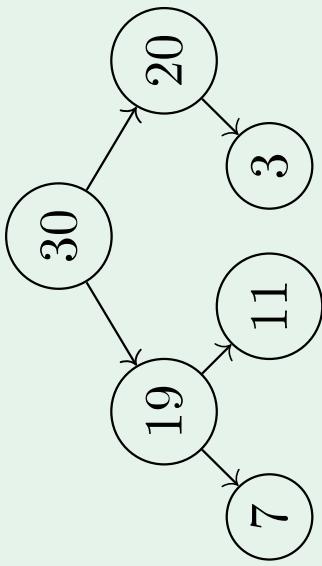
### RIPARARBCANCELLA – Analisi di complessità

- I casi 0,1,3 e 4 di RIPARARBCANCELLA effettuano un numero costante di rotazioni e scambi di colore → sono  $\mathcal{O}(k)$
- L'unica chiamata ricorsiva è quella che avviene nel caso 2 sul padre di  $x$ 
  - Nel caso il padre ricada nei casi 0,1,3 o 4, la nuova chiamata termina in  $\mathcal{O}(k)$
  - In caso contrario, viene ri-invocata RIPARARBCANCELLA
- Ad ogni chiamata ricorsiva si risale di un livello verso la radice → al massimo effettuiamo  $\mathcal{O}(\log(n))$  chiamate
- La procedura complessiva di cancellazione da alberi RB è quindi  $\mathcal{O}(\log(n))$  come tutte le altre azioni

# Mucchi (Heaps)

## Una struttura parzialmente ordinata

- Un *mucchio* (*heap*) è una struttura dati ad albero **la chiave del nodo padre è sempre maggiore (max-heap) di quella dei figli**
  - Nessuna relazione sussiste tra le chiavi di due fratelli
  - È possibile definirne una variante in cui la chiave del padre è sempre minore di quella dei figli (min-heap)
- Se l'albero è binario, parliamo di mucchi binari (binary heaps)
  - Manteniamo lo heap come un albero quasi-completo
- Esempio di max-heap:



# Mucchi (Heaps)

## Proprietà e usi pratici

- Gli heap, in particolare gli heap binari, trovano uso per:
  - Implementare code con priorità
  - Ordinare vettori (proposti originariamente per questo)
- Per tutti gli usi più comuni, è conveniente materializzare lo heap sempre come struttura dati implicita
  - È un albero binario quasi-completo → le foglie mancanti sono quelle che occupano la parte finale dell'array in cui è stoccatato
  - Avremo un attributo  $A.heapsize$  che indica il numero di elementi dello heap e  $A.length$  che contiene la lunghezza dell'array di supporto:  $A.heapsize \leq A.length$
- Le operazioni su un max-heap sono: MAX, INSERISCI, CANCELLA-MAX, COSTRUISCI-MAX-HEAP, MAX-HEAPIFY
  - In un max-heap l'elemento con chiave più grande è la radice

## Code con priorità

### Riassunto delle complessità

- Una coda con priorità implementata con uno heap binario ha un costo, sia per l'accodamento che per l'estrazione, pari a  $\mathcal{O}(\log(n))$
- Inserendo gli elementi uno alla volta, si ha un costo complessivo di  $\mathcal{O}(n \log(n))$  per la costruzione dell'intera coda
- Apparentemente, il costo è identico a quello della COSTRUISCI-MAX-HEAP
- È in realtà possibile dimostrare che COSTRUISCI-MAX-HEAP risulta essere in grado di costruire lo heap in  $\mathcal{O}(n)$

# HeapSort

## Considerazioni

- HEAPSORT ha complessità  $\mathcal{O}(n \log(n))$  nel caso pessimo: è la migliore possibile
- Necessita solamente di  $\mathcal{O}(1)$  in spazio ausiliario (ordina sul posto) a differenza di MERGESORT
- Nelle implementazioni pratiche, nel caso medio, è più lento di QUICKSORT:  
HEAPSORT ha un costo lineare sommato a  $\mathcal{O}(n \log(n))$  che viene sempre pagato
- Resta il vantaggio della complessità di caso pessimo garantita
- Come la versione di MERGESORT out-of-place HEAPSORT non è stabile

# Grafi

## Rappresentazione in memoria

- Sono possibili due strategie per rappresentare un grafo: liste di adiacenza e matrice di adiacenza
- Liste di adiacenza:
  - Un vettore di liste lungo  $|V|$ , indicizzato dai nomi dei nodi
  - Ogni lista contiene i nodi adiacenti all'indice della sua testa
- Matrice di adiacenza:
  - Una matrice di valori booleani  $|V| \times |V|$ , con righe e colonne indicizzate dai nomi dei nodi
    - la cella alla riga  $i$ , colonna  $j$  contiene 1 se l'arco  $(v_i, v_j)$  è presente nel grafo (0 altrimenti)

# Visita in ampiezza

## Pseudocodice - VISITA AMPIEZZA( $G, s$ )

```
VISITAAMPIEZZA( $G, s$ )
1   for each  $n \in V \setminus \{s\}$ 
2      $n.color \leftarrow white$ 
3      $n.dist \leftarrow \infty$ 
4      $s.color \leftarrow grey$ 
5      $s.dist \leftarrow 0$ 
6      $Q \leftarrow \emptyset$ 
7     ENQUEUE( $Q, s$ )
8   while  $\neg$ ISEMPTY( $Q$ )
9      $curr \leftarrow$  DEQUEUE( $Q$ )
10    for each  $v \in curr.adjacenti$ 
11      if  $v.color = white$ 
12         $v.color \leftarrow grey$ 
13         $v.dist \leftarrow curr.dist + 1$ 
14        ENQUEUE( $Q, v$ )
15     $curr.color \leftarrow black$ 
```

- Linee 1–7 : inizializzano tutti i nodi come bianchi
- Linee 8–15: effettuano la visita del grafo
- N.B. Ogni arco è visitato una sola volta (a partire dal nodo sorgente)
- Complessità totale:  
 $\mathcal{O}(|V| + |E|)$

# Dijkstra Ottimizzato

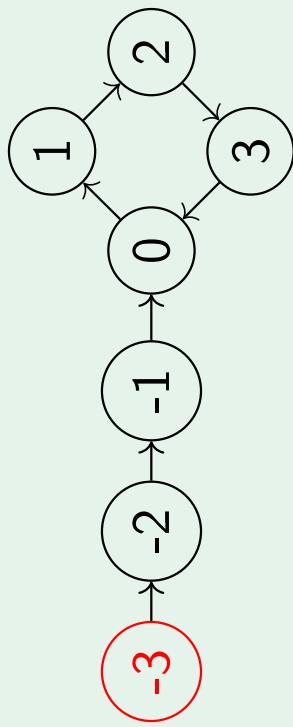
DIJKSTRAQUEUE( $G, s$ )

```
1    $Q \leftarrow \emptyset$ 
2    $s.dist \leftarrow 0$ 
3   for each  $v \in V$ 
4     if  $v \neq s$ 
5        $v.dist \leftarrow \infty$ 
6        $v.pred \leftarrow NIL$ 
7       ACCODAPRI( $Q, v, v.dist$ )
8   while  $Q \neq \emptyset$ 
9      $u \leftarrow \text{CANCELLAMIN}(Q)$ 
10    for each  $v \in u.succ$ 
11       $ndis \leftarrow u.dist + peso(u, v)$ 
12      if  $v.dist > ndis$ 
13         $v.dist \leftarrow ndis$ 
14         $v.prev \leftarrow u$ 
15    DECREMENTAPRI( $Q, v, ndis$ )
```

- Le righe 3–7 inizializzano la coda: costo  $\mathcal{O}(|V| \log(|V|))$
- Le righe 8–15 visitano ogni arco una volta (grafo come lista di adiacenze): Costo  $\mathcal{O}(|E| \log(|V|))$
- Compl. totale  $\mathcal{O}((|E| + |V|) \log(|V|))$

# Algoritmo di Floyd

## La lepre e la tartaruga - Idea



- Immaginiamo che il cammino su cui vogliamo individuare il ciclo sia una pista
- Usiamo due riferimenti  $t$  e  $l$  che spostiamo a ogni passo:
  - Nel caso di  $t$ , dal nodo a cui punta al successore (1 “passo”)
  - Nel caso di  $l$ , dal nodo a cui punta al successore del successore (2 “passi”)
- Entrambi partono dal nodo iniziale ( $-3$  in figura)
- Se esiste un ciclo, essi sono destinati a “incontrarsi” ( $l$  doppierebbe prima o poi  $t$ )

# Riconoscimento di cicli

FLOYDLT( $G, x$ )

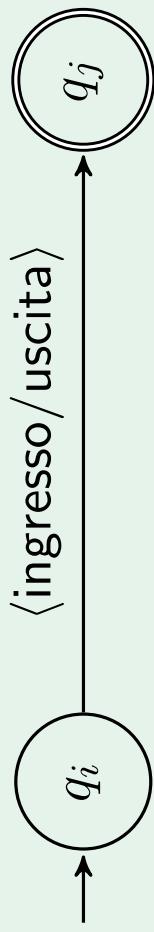
```
1    $t \leftarrow x.succ$ 
2    $l \leftarrow x.succ.succ$ 
3   while  $l \neq t$ 
4        $t \leftarrow t.succ$ 
5        $l \leftarrow l.succ.succ$ 
6    $T \leftarrow 0$ 
7    $t \leftarrow x$ 
8   while  $l \neq t$ 
9        $t \leftarrow t.succ$ 
10       $l \leftarrow l.succ$ 
11       $T \leftarrow T + 1$ 
12       $l \leftarrow t$ 
13       $C \leftarrow 0$ 
14      while  $l \neq t$ 
15           $l \leftarrow l.succ$ 
16           $C \leftarrow C + 1$ 
17      return  $T, C$ 
```

- Le righe 1–5 trovano il ciclo quando  $l$  riprende  $t$
- Le righe 6–11 calcolano  $T$  facendo ripartire  $t$
- Le righe 14–16 calcolano  $C$  tenendo  $t$  ferma come segnaposto per  $l$
- Complessità temporale:  
 $\Theta(T + C - r + T + C) = \Theta(2(T + C) - r)$
- Complessità spaziale:  $\Theta(1)$

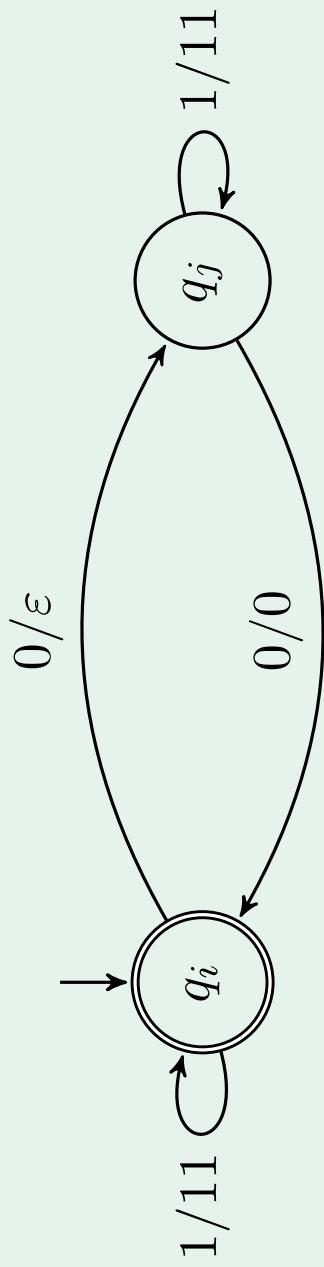
# Traduttori

## FSA traduttore

- Un FSA traduttore da  $L_1$  a  $L_2$  associa un simbolo letto e uno scritto a ogni transizione



$L_1 \subset \{0,1\}^*$  stringhe con numero di "0" pari,  $\tau$  : dimezza gli "0", raddoppia gli "1"



## Chiusura nei linguaggi

### Chiusura di famiglie di linguaggi

- Famiglia di linguaggi: un insieme  $\mathbb{L}$  i cui elementi sono linguaggi,  $\mathbb{L} = \{L_i\}$
- $\mathbb{L}$  è chiusa rispetto a un'operazione (binaria)  $\star$  se  $\forall L_1, L_2 \in \mathbb{L}$  vale  $L_1 \star L_2 \in \mathbb{L}$

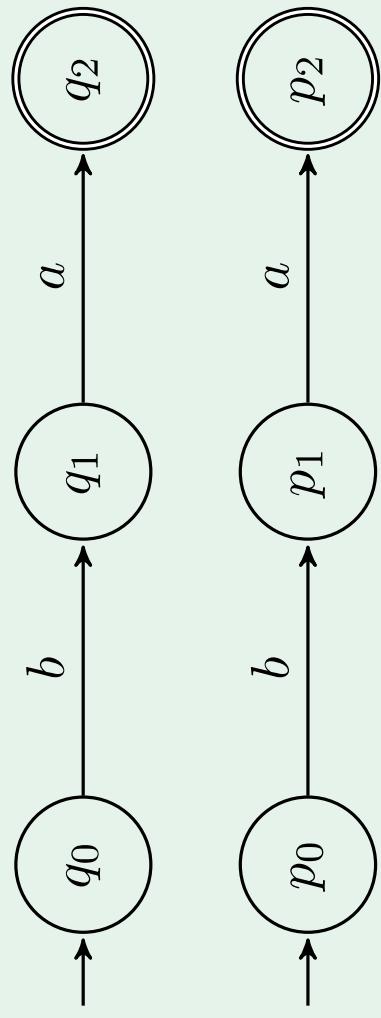
### Linguaggi regolari

- La famiglia di linguaggi riconoscibili con un FSA è la famiglia dei linguaggi *regolari*, **R** o REG
- **R** è chiusa rispetto a  $\cup, \cap, \neg, \backslash,$  alla concatenazione,  $a * e +$

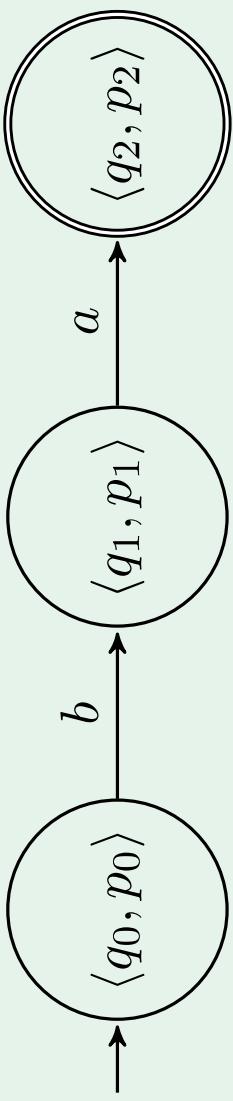
## Intersezione tra linguaggi

Combinare gli FSA

Dati due FSA riconoscitori

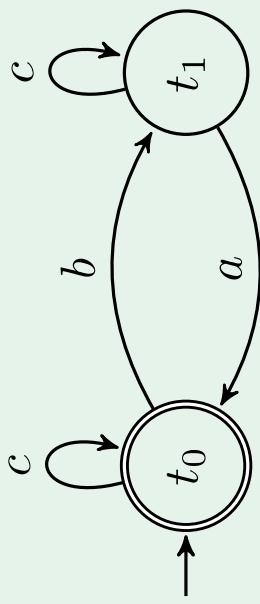
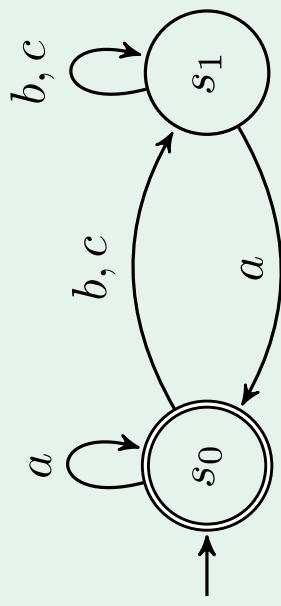


Ottengo il riconoscitore del linguaggio intersezione facendoli funzionare “insieme”: è possibile una transizione solo se c’è in entrambi

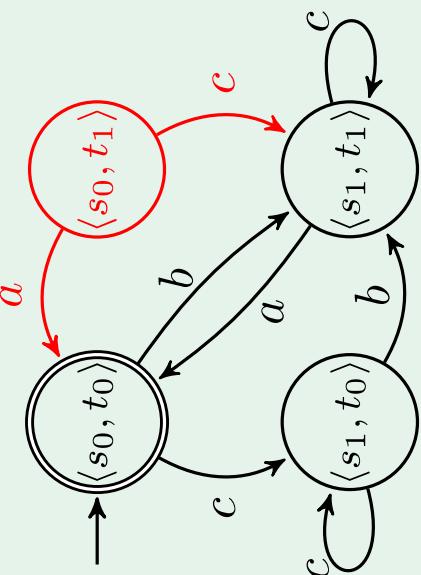


## Intersezione: esempio

Automi “sorgente”



Automa intersezione



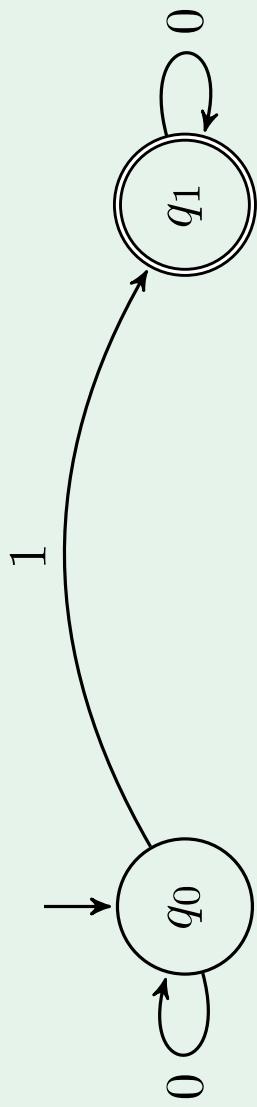
- Lo stato in **rosso** non è raggiungibile dallo stato iniziale  $\rightarrow$  può essere eliminato
- Con la costruzione a punto fisso a partire da  $\langle s_0, t_0 \rangle$  non viene neppure aggiunto

## Complemento

Il complemento si fa rispetto ad  $A^*$  (con  $A$  l'alfabeto)

Costruito operativamente

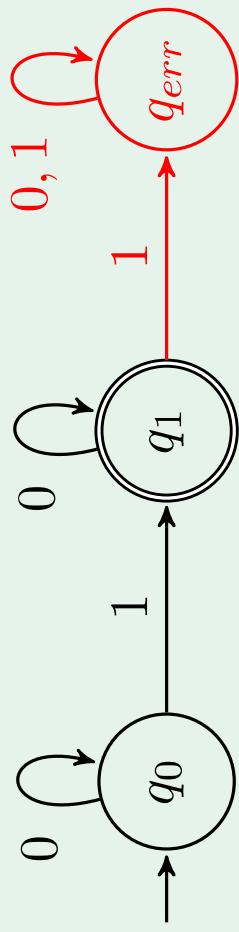
- Idea generale: "rendere finali gli stati non finali e viceversa" ( $F_{compl} = Q \setminus F$ )
- È sufficiente scambiare i ruoli degli stati?



# Complemento

## Gestire $\delta$ parziali

- No.  $\delta$  è una funzione parziale, negli FSA uno stato non accettante è “implicito”: lo stato di errore



- Aggiunto  $q_{err}$  nell’FSA, “scambiare  $F$  e  $Q \setminus F$ ” funziona
- Problema non così facile da risolvere con altri modelli di calcolo
- In generale: calcolare la risposta negativa a un quesito non è equivalente a quella positiva.