Complessità computazionale

— ○ —

Un modello della computazione semplice ma esaustivo

Piero A. Bonatti

Università di Napoli Federico II

Laurea Magistrale in Informatica

Tema della lezione

- Per dimostrare proprietà degli algoritmi e dei problemi dobbiamo darne una *formalizzazione matematica*
  - la descrizione a parole dei problemi non è adeguata (il linguaggio naturale è ambiguo)
  - i linguaggi C, Java, ecc. non sono adatti
    - ne hanno formalizzato solo la *grammatica*
    - non la *semantica*
    - che nel caso del C dipende fortemente dall'hardware
  - A noi occorrono descrizioni di problemi e algoritmi che prescindano dal linguaggio di programmazione usato e dall'hardware
    - aspiriamo a una teoria *universale*, indipendente da quei dettagli
    - una teoria di *rigore matematico*

Tema della lezione

- A questo scopo formalizzeremo
  - i *problemi* mediante *linguaggi*
  - gli *algoritmi* mediante *Macchine di Turing* (MdT)
    - che modellano software e hardware insieme
    - permettendo di definire le *risorse* utilizzate da un algoritmo in modo rigoroso
- Le MdT costituiscono un modello di computazione elementare, quasi un assembler
  - "maneggevole" nelle dimostrazioni matematiche per la sua semplicità

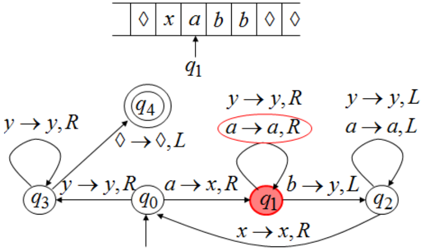
  ma al tempo stesso del tutto generale
  - potente quanto *qualsiasi* linguaggio di programmazione
  - con una (eventuale) perdita di efficienza solo polinomiale
    - "preserva la classe dei problemi trattabili"
    - e le riduzioni

Le Macchine di Turing

"*Sorprendente quanto poco serva per avere tutto*"

Le MdT in due righe

- Una MdT è costituita da:
  - un programma finito detto relazione di transizione
    - un automa a stati finiti
  - una memoria illimitata detta nastro
    - una stringa di simboli



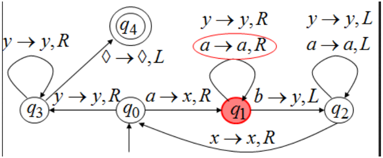
Struttura delle MdT

- Il nastro (tape)
  - è suddiviso in *celle*
    - come le parole di memoria dei moderni computer ma...
  - è ad accesso sequenziale (con una testina di lettura/scrittura)
    - ispirato dalle macchine programmabili note negli anni '30
  - contrariamente ai nastri perforati di carta si può *riscrivere*
    - le memorie elettromeccaniche (poi a relè) escono 2 anni dopo (Z1 di Konrad Zuse)
    - per i nuclei di ferrite dobbiamo aspettare i '50
    - per i supporti magnetici dovremo aspettare la *Olivetti Programma 101* nei primi '60
  - e diversamente dai computer reali *si estende all'infinito*
    - sebbene in ogni istante la porzione di nastro usata sia *finita*
    - è come poter fare un numero grande a piacere di *malloc* senza mai andare *out of memory*

Struttura delle MdT (deterministiche)

- Il programma, per ogni stato e per ogni possibile simbolo letto dalla testina, specifica
  - quale simbolo riscrivere al suo posto (assegnamento)
  - in che direzione muovere la testina (prox operando)
  - in quale stato spostarsi (goto)
- Equivale a istruzioni del tipo:

$$
\begin{array}{llll}
q_i\text{:} & \textbf{if } \text{tape[pos]}=s_1 & \textbf{then } \text{tape[pos]}=s'_1;\ \text{pos++;} & \textbf{goto } q'_1 \\
& \textbf{else if } \text{tape[pos]}=s_2 & \textbf{then } \text{tape[pos]}=s'_2;\ \text{pos--;} & \textbf{goto } q'_2 \\
& \vdots & & \\
& \textbf{else if } \text{tape[pos]}=s_n & \textbf{then } \text{tape[pos]}=s'_n;\ \text{pos++;} & \textbf{goto } q'_n
\end{array}
$$



MdT – Definizione formale

Macchina di Turing deterministica

È una quadrupla $M = (K, \Sigma, \delta, s)$ dove

- K è un insieme finito di *stati* (le *labels* q_i)
- $s \in K$ è lo *stato iniziale*
- Σ è un insieme finito di simboli (l'*alfabeto* di M)
- δ è la *funzione di transizione* (programma)

$$\delta : K \times \Sigma \rightarrow (K \cup \{h, "yes", "no"\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\} \qquad (1)$$

Σ contiene i simboli speciali $\sqcup$ (blank) e $\triangleright$ (inizio nastro)

Le celle hanno la stessa capienza limitata (come in realtà)

h=halting state, "*yes*"/"*no*"=accepting/rejecting state

Spostamento testina (sinistra, destra, non spostare)

MdT - Notazione per l'output

- Sia x l'input di M:
  - $M(x) = \text{"}yes\text{"}$: M termina nello stato "*yes*"
    - diciamo che M *accetta* x
  - $M(x) = \text{"}no\text{"}$: M termina nello stato "*no*"
    - diciamo che M *rigetta* x
  - $M(x) = y$: M termina nello stato h e y è l'output
  - $M(x) = \nearrow$: M non termina (o *diverge*)

Esempio informale di computazione: $M(010) = \sqcup 010$

$p \in K,$	$\sigma \in \Sigma$	$\delta(p, \sigma)$
$s,$	0	$(s, 0, \rightarrow)$
$s,$	1	$(s, 1, \rightarrow)$
$s,$	$\sqcup$	$(q, \sqcup, \leftarrow)$
$s,$	$\triangleright$	$(s, \triangleright, \rightarrow)$
$q,$	0	$(q_0, \sqcup, \rightarrow)$
$q,$	1	$(q_1, \sqcup, \rightarrow)$
$q,$	$\sqcup$	$(q, \sqcup, -)$
$q,$	$\triangleright$	$(h, \triangleright, \rightarrow)$
$q_0,$	0	$(s, 0, \leftarrow)$
$q_0,$	1	$(s, 0, \leftarrow)$
$q_0,$	$\sqcup$	$(s, 0, \leftarrow)$
$q_0,$	$\triangleright$	$(h, \triangleright, \rightarrow)$
$q_1,$	0	$(s, 1, \leftarrow)$
$q_1,$	1	$(s, 1, \leftarrow)$
$q_1,$	$\sqcup$	$(s, 1, \leftarrow)$
$q_1,$	$\triangleright$	$(h, \triangleright, \rightarrow)$

0.	$s,$	$\underline{\triangleright}010$
1.	$s,$	$\triangleright\underline{0}10$
2.	$s,$	$\triangleright0\underline{1}0$
3.	$s,$	$\triangleright01\underline{0}$
4.	$s,$	$\triangleright010\underline{\sqcup}$
5.	$q,$	$\triangleright01\underline{0}\sqcup$
6.	$q_0,$	$\triangleright01\sqcup\underline{\sqcup}$
7.	$s,$	$\triangleright01\underline{\sqcup}0$
8.	$q,$	$\triangleright0\underline{1}\sqcup0$
9.	$q_1,$	$\triangleright0\sqcup\underline{\sqcup}0$
10.	$s,$	$\triangleright0\underline{\sqcup}10$
11.	$q,$	$\triangleright\underline{0}\sqcup10$
12.	$q_0,$	$\triangleright\sqcup\underline{\sqcup}10$
13.	$s,$	$\triangleright\underline{\sqcup}010$
14.	$q,$	$\underline{\triangleright}\sqcup010$
15.	$h,$	$\triangleright\underline{\sqcup}010$

Definizione formale di computazione

Configurazioni

Configurazione

È una tripla (q, w, u) dove

- $q \in K$ (è lo *stato corrente*, il "program counter")
- w, u sono stringhe in Σ^*
  - w è la stringa a sinistra del cursore, compreso il simbolo sotto la testina
  - u è la stringa a destra del cursore

Definizione formale di computazione

Configurazioni

- Esempio:

4.	$s,$	$\triangleright 010\underline{\sqcup}$	$(s, \triangleright 010\sqcup, \epsilon)$	(ϵ=stringa vuota)
5.	$q,$	$\triangleright 01\underline{0}\sqcup$	$(q, \triangleright 010, \sqcup)$	
6.	$q_0,$	$\triangleright 01 \sqcup \underline{\sqcup}$	$(q_0, \triangleright 010\sqcup, \sqcup)$	

- Alcuni testi scrivono (q, w, u) come la stringa wqu
  - sfruttando l'ipotesi che $K \cap \Sigma = \emptyset$

Definizione formale di computazione

Singolo passo di M

- Siano $w = w_1 \cdots w_m$ e $u = u_1 \cdots u_n$ stringhe in Σ^*
- Scriviamo $(q, w, u) \xrightarrow{M} (q', w', u')$ sse
  - $\delta(q, w_m) = (r, \sigma, -)$ e $(q', w', u') = (r, w_1 \cdots w_{m-1}\sigma, u)$
  - $\delta(q, w_m) = (r, \sigma, \leftarrow)$ e $(q', w', u') = (r, w_1 \cdots w_{m-1}, \sigma u)$
  - $\delta(q, w_m) = (r, \sigma, \rightarrow)$ e $(q', w', u') = (r, w_1 \cdots w_{m-1}\sigma u_1, u_2 \cdots u_n)$
    - se $u \neq \epsilon$, altrimenti
  - $\delta(q, w_m) = (r, \sigma, \rightarrow)$ e $(q', w', u') = (r, w_1 \cdots w_{m-1}\sigma\sqcup, \epsilon)$

Definizione formale di computazione

Iterazione per k passi

- Scriviamo $(q, w, u) \xrightarrow{M^k} (q', w', u')$ (dove $\sigma \in \Sigma$) sse
  - esistono $k + 1$ configurazioni (q_i, w_i, u_i), $i = 0, 1, \ldots, k$
  - $(q, w, u) = (q_0, w_0, u_0)$
  - $(q_i, w_i, u_i) \xrightarrow{M} (q_{i+1}, w_{i+1}, u_{i+1})$, $i = 0, 1, \ldots, k - 1$
  - $(q_k, w_k, u_k) = (q', w', u')$

Definizione formale di computazione

Computazione

Scriviamo $(q, w, u) \xrightarrow{M^*} (q', w', u')$ sse esiste $k \geq 0$ tale che

$$(q, w, u) \xrightarrow{M^k} (q', w', u')$$

- Ora possiamo formalizzare le definizioni di output così
  - $M(x) =$ "*yes*" sse $\exists w, u$ t.c. $(s, \triangleright, x) \xrightarrow{M^*} (\text{"}yes\text{"}, w, u)$
  - $M(x) =$ "*no*" sse $\exists w, u$ t.c. $(s, \triangleright, x) \xrightarrow{M^*} (\text{"}no\text{"}, w, u)$
  - $M(x) = y$ sse $\exists w, u$ t.c. $(s, \triangleright, x) \xrightarrow{M^*} (h, \triangleright w, u)$ e $y = wu$

Esempio: Successore di un numero binario

$p \in K,$	$\sigma \in \Sigma$	$\delta(p, \sigma)$
$s,$	0	$(s, 0, \rightarrow)$
$s,$	1	$(s, 1, \rightarrow)$
$s,$	$\sqcup$	$(q, \sqcup, \leftarrow)$
$s,$	$\triangleright$	$(s, \triangleright, \rightarrow)$
$q,$	0	$(h, 1, -)$
$q,$	1	$(q, 0, \leftarrow)$
$q,$	$\triangleright$	$(h, \triangleright, \rightarrow)$

$$
\begin{array}{lcl}
(s, \triangleright, 11011) & \xrightarrow{M} & (s, \triangleright 1, 1011) \\
 & \xrightarrow{M} & (s, \triangleright 11, 011) \\
 & \xrightarrow{M} & (s, \triangleright 110, 11) \\
 & \xrightarrow{M} & (s, \triangleright 1101, 1) \\
 & \xrightarrow{M} & (s, \triangleright 11011, \epsilon) \\
 & \xrightarrow{M} & (s, \triangleright 11011\sqcup, \epsilon) \\
 & \xrightarrow{M} & (q, \triangleright 11011, \sqcup) \\
 & \xrightarrow{M} & (q, \triangleright 1101, 0\sqcup) \\
 & \xrightarrow{M} & (q, \triangleright 110, 00\sqcup) \\
 & \xrightarrow{M} & (h, \triangleright 111, 00\sqcup)
\end{array}
$$

$$M(11011) = 11100$$

Esempio: Successore di un numero binario

C'è un baco! Se l'input è $11\cdots1$ restituisce 0 (overflow)

$p \in K,$	$\sigma \in \Sigma$	$\delta(p,\sigma)$
$s,$	0	$(s, 0, \rightarrow)$
$s,$	1	$(s, 1, \rightarrow)$
$s,$	$\sqcup$	$(q, \sqcup, \leftarrow)$
$s,$	$\triangleright$	$(s, \triangleright, \rightarrow)$
$q,$	0	$(h, 1, -)$
$q,$	1	$(q, 0, \leftarrow)$
$q,$	$\triangleright$	$(h, \triangleright, \rightarrow)$

$$
\begin{array}{lcl}
(s, \triangleright, 11) & \xrightarrow{M} & (s, \triangleright 1, 1) \\
 & \xrightarrow{M} & (s, \triangleright 11, \epsilon) \\
 & \xrightarrow{M} & (s, \triangleright 11\sqcup, \epsilon) \\
 & \xrightarrow{M} & (q, \triangleright 11, \sqcup) \\
 & \xrightarrow{M} & (q, \triangleright 1, 0\sqcup) \\
 & \xrightarrow{M} & (q, \triangleright, 00\sqcup) \\
 & \xrightarrow{M} & (h, \triangleright 0, 0\sqcup)
\end{array}
$$

$M(11) = 00$?!?

Esempio: Successore di un numero binario

- Correzione 1: aggiungere nuovi stati
  - raggiunto $\triangleright$, invece di terminare,
  - scrive 1 dopo $\triangleright$
  - poi sposta di una cella a destra il numero che era sul nastro quando si è raggiunto $\triangleright$
- Programmare questa MdT per esercizio
  - suggerimento: servono 2 nuovi stati
  - uno per scrivere 0, uno per scrivere 1

Esempio: Successore di un numero binario

- Correzione 2: Comporre due MdT
- prima di lanciare la MdT M col baco aggiungiamo uno 0 iniziale per evitare l'overflow:
  1. Modificare la MdT che inseriva un $\sqcup$ in testa per farle inserire uno 0; chiamiamola M_0
  2. Concatenare M_0 e M (la MdT col baco)

Concatenazione di M_1 e M_2, $(M_1; M_2)$

Procedimento di interesse generale. $(M_1; M_2)(x) = M_2(M_1(x))$

- Siano $M_i = (K_i, \Sigma, \delta_i, s_i)$, per $i = 1, 2$
- Assumiamo senza perdere generalità che $K_1 \cap K_2 = \emptyset$
  - possiamo sempre ridenominare gli stati

  e che M_1 termini con la testina su $\triangleright$
  - possiamo sempre aggiungere uno stato che riporta la testina all'inizio (fare per esercizio sugli esempi precedenti)
- Definiamo la composizione $(M_1; M_2) = (K_1 \cup K_2, \Sigma, \delta, s_1)$:
  - per ogni $q \in K_1$ e $\sigma \in \Sigma$:
  - se $\delta_1(q, \sigma) = (r, \sigma', D)$ e $r \notin \{h, "yes", "no"\}$, allora $\delta(q, \sigma) = \delta_1(q, \sigma)$
  - se $\delta_1(q, \sigma) = (r, \sigma', D)$ e $r \in \{h, "yes", "no"\}$, allora $\delta(q, \sigma) = (s_2, \sigma', D)$

  - per ogni $q \in K_2$ e $\sigma \in \Sigma$: $\delta(q, \sigma) = \delta_2(q, \sigma)$

Esempio: Riconoscimento di stringhe palindrome

- Problema: dire se una stringa binaria è palindroma (ad es. 11011)
- È un problema di decisione: La MdT deve terminare su "yes" se la stringa è palindroma e su "no" altrimenti
- Vedere l'Esempio 2.3 sul libro e la simulazione su `http://morphett.info/turing/turing.html`
- Piccole differenze nella forma delle MdT
  - niente $\triangleright$
  - niente stati speciali "yes", "no": disegna emoticons invece
    - confrontare con Esempio 2.3 per vedere gli effetti di queste differenze su δ
  - wildcard '*' per rappresentare δ più comodamente
  - potete usare questo simulatore per debuggare i vostri esercizi

Le MdT come algoritmi

Linguaggi e funzioni su stringhe

- Le MdT si prestano naturalmente a due tipi di compiti:
  - calcolare funzioni su stringhe
  - "riconoscere linguaggi"
    - prendere una stringa e dire se "appartiene al linguaggio"
    - un po' come un compilatore controlla se un programma è sintatticamente corretto (rispetto alla grammatica del linguaggio di programmazione)
    - vi sono due modi di "riconoscere": *accettare* e *decidere*

Linguaggi e MdT

- Sia dato un linguaggio $L \subset (\Sigma \setminus \{\sqcup\})^*$
  - un insieme di stringhe finite di simboli di Σ (tranne $\sqcup$)
  - le stringhe $x \in L$ sono quelle "corrette" nel linguaggio L
- Se esiste una MdT M tale che, per ogni stringa $x \in (\Sigma \setminus \{\sqcup\})^*$

$$M(x) = \begin{cases} \text{"}yes\text{"} & \text{se } x \in L \\ \text{"}no\text{"} & \text{altrimenti} \end{cases}$$

allora diciamo che

1. *M decide L*
2. L è un linguaggio *ricorsivo* (es.: stringhe binarie palindrome)

(riformulazione con MdT di nozioni da Elementi di Informatica Teorica)

Linguaggi e MdT (II)

- Dato un linguaggio $L \subset (\Sigma \setminus \{\sqcup\})^*$
- Se esiste una MdT M tale che, per ogni stringa $x \in (\Sigma \setminus \{\sqcup\})^*$

$$M(x) = \begin{cases} \text{"}yes\text{"} & \text{se } x \in L \\ \nearrow & \text{altrimenti} \end{cases}$$

allora diciamo che

1. *M accetta L*
2. *L è un linguaggio ricorsivamente enumerabile* (es.: le formule valide della logica del 1^o ordine)

(riformulazione con MdT di nozioni da Elementi di Informatica Teorica)

Linguaggi e MdT (III)

- Nota:
  - nell'accettazione di linguaggi ricorsivamente enumerabili solo la risposta positiva è utile
    - se $x \notin L$ la computazione di M non termina
  - non è un "vero" concetto algoritmico
  - solo un modo utile di *categorizzare i problemi*, indicandone la "complessità intrinseca"
- Ritroveremo un pattern simile con le MdT *non* deterministiche

Linguaggi ricorsivi e ricorsivamente enumerabili

Proposizione (2.1 nel libro)

Se L è ricorsivo allora è ricorsivamente enumerabile

Prova:

- Se L è ricorsivo allora esiste M che lo decide.
- Trasformiamo M in una macchina M' che accetta L:
- in δ rimpiazziamo gli stati finali con un nuovo stato q;
- q non fa altro che spostare la testina a destra all'infinito, rimanendo sempre in q.

QED

Funzioni su stringhe e MdT

- Data una funzione $f : (\Sigma \setminus \{\sqcup\})^* \to \Sigma^*$
  - notare che $\sqcup$ può comparire solo nell'output
- Se esiste una MdT M tale che, per ogni stringa $x \in (\Sigma \setminus \{\sqcup\})^*$

$$M(x) = f(x)$$

allora diciamo che

1. *M calcola f*
2. f è una funzione *ricorsiva*

(c'è anche la versione *ricorsivamente enumerabile* che si applica alle funzioni *parziali*)

MdT e *problemi*

- Abbiamo introdotto le MdT per formalizzare gli algoritmi che risolvono problemi di decisione e ottimizzazione
- Ma cosa hanno a che vedere linguaggi e funzioni su stringhe con quei problemi?
  - dove le istanze non sono stringhe, ma oggetti matematici come grafi, reti e numeri
- Semplice: dobbiamo rappresentare le istanze del problema come stringhe (encoding)
- Una volta fissata la rappresentazione, un algoritmo per un problema di decisione è una MdT che decide il linguaggio corrispondente
  - ovvero la risposta della MdT coincide con la risposta al problema ("yes" o "no")
  - il "linguaggio corrispondente al problema" è l'insieme delle stringhe che codificano le istanze positive del problema

MdT e *problemi* (II)

- Se il problema ha un output più complicato (vedi MAX FLOW)
- allora bisogna decidere una rappresentazione con stringhe anche per l'output
- Un algoritmo che risolve il problema è una MdT che calcola la corrispondente funzione su stringhe

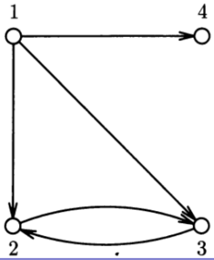
MdT e problemi (III)

- Il metodo è del tutto generale: qualunque oggetto matematico finito può essere codificato con una stringa:
  - gli elementi di insiemi finiti: con interi in binario (es.: i nodi di un grafo)
  - le n-uple: con parentesi tonde e virgole (es.: gli archi)
  - gli insiemi finiti: con graffe e virgole
  - le matrici: per righe, separandole con ';'
- Questi sono solo esempi: si possono concepire e usare diversi encoding
  - e la scelta non dovrebbe influenzare la misura di complessità del *problema*

Di nuovo sugli encoding *ragionevoli*

- Dovrebbero essere *correlati polinomialmente*
  - dati due encoding A e B deve esistere un polinomio p
  - tale che, per ciascuna istanza del problema
  - se è codificata con n caratteri in A
  - allora è codificata con al massimo $p(n)$ caratteri in B

- Esempio positivo: codifica grafi con lista o matrice di adiacenza. L'aumento è al più quadratico



"$\{(1, 100), (1, 10), (10, 11), (1, 11), (11, 10)\}$"

"$(0111; 0010; 0100; 0000)$"

Di nuovo sugli encoding *ragionevoli* (II)

- Esempio negativo: Ricordate la rappresentazione delle capacità in MAX FLOW
  - le rappresentazioni in unario sono esponenzialmente più lunghe di quelle posizionali (binarie, decimali, esadecimali o altro)
    - $n = a^{\log_a n}$; invece:
    - $\log_b n = \log_b a \cdot \log_a n$
  - non lo consideriamo un encoding ragionevole
  - nel libro si assume sempre che i numeri siano rappresentati in binario

decim.	bin.	unario
1	1	1
2	10	11
	⋮	
50	110010	11111111111111111111111111111111111111111111111111

Potenziando (?) il nostro assembler

MdT a più nastri

MdT a k nastri

- Per rendere certe dimostrazioni più facili e certe definizioni più chiare, conviene potenziare le MdT aggiungendo più nastri
  - fungono da *variabili ausiliarie*
- Vedremo che non si aumenta il potere espressivo delle MdT
- e che non se ne aumenta *significativamente* la performance
  - il numero di nastri non cambia le classi di complessità dei problemi

MdT con k nastri (e k testine)

Definizione: k-string Turing machine

Una n-upla $M = (K, \Sigma, \delta, s)$

- K, Σ, s sono come prima
- $\delta : K \times \Sigma^k \to (K \cup \{h, \text{"}yes\text{"}, \text{"}no\text{"}\}) \times (\Sigma \times \{\leftarrow, \rightarrow, -\})^k$

- Esempio con due nastri:
- $\delta(q, \sigma_1, \sigma_2) = (r, \sigma_1', D_1, \sigma_2', D_2)$ significa che
  - quando lo stato corrente è q e le testine dei due nastri leggono rispettivamente σ_1 e σ_2
  - lo stato successivo sarà r
  - il simbolo σ_1 sul primo nastro viene riscritto con σ_1' e la prima testina spostata come dice D_1
  - il simbolo σ_2 sul secondo nastro viene riscritto con σ_2' e la seconda testina spostata come dice D_2

MdT con k nastri

Input e output

- Il primo e l'ultimo nastro hanno un significato particolare:

- Inizio e input:
  - il primo nastro contiene l'input ($\triangleright x$)
  - gli altri solo $\triangleright$

- Terminazione e output:
  - $M(x) = ''yes''/''no''$ sse lo stato finale è $''yes''/''no''$
  - $M(x) = y$ sse lo stato finale è h e l'ultimo nastro (il k-esimo) contiene $\triangleright y$

Esempio: MdT a 2 nastri per le palindrome

- Copia l'input sul secondo nastro (s), poi riporta la prima testina all'inizio (q) e scandisce i nastri in senso inverso verificando l'uguaglianza (p). Da $(n+1)n/2$ a soli $3n$ passi

$p \in K,$	$\sigma_1 \in \Sigma$	$\sigma_2 \in \Sigma$	$\delta(p, \sigma_1, \sigma_2)$
$s,$	0	$\sqcup$	$(s, 0, \rightarrow, 0, \rightarrow)$
$s,$	1	$\sqcup$	$(s, 1, \rightarrow, 1, \rightarrow)$
$s,$	$\triangleright$	$\triangleright$	$(s, \triangleright, \rightarrow, \triangleright, \rightarrow)$
$s,$	$\sqcup$	$\sqcup$	$(q, \sqcup, \leftarrow, \sqcup, -)$
$q,$	0	$\sqcup$	$(q, 0, \leftarrow, \sqcup, -)$
$q,$	1	$\sqcup$	$(q, 1, \leftarrow, \sqcup, -)$
$q,$	$\triangleright$	$\sqcup$	$(p, \triangleright, \rightarrow, \sqcup, \leftarrow)$
$p,$	0	0	$(p, 0, \rightarrow, \sqcup, \leftarrow)$
$p,$	1	1	$(p, 1, \rightarrow, \sqcup, \leftarrow)$
$p,$	0	1	$(\text{“no”}, 0, -, 1, -)$
$p,$	1	0	$(\text{“no”}, 1, -, 0, -)$
$p,$	$\sqcup$	$\triangleright$	$(\text{“yes”}, \sqcup, -, \triangleright, \rightarrow)$

MdT a k nastri

Configurazioni e computazioni

- Le configurazioni sono generalizzate analogamente: $(q, w_1, u_1, \ldots, w_k, u_k)$
- Così come le computazioni di uno più passi:
  - $\xrightarrow{M}$: ripete le modifiche del nastro per tutti i k nastri (definizione tediosa ma semplice)
  - $\xrightarrow{M^n}$: concatenazione di n transizioni $\xrightarrow{M}$
  - $\xrightarrow{M^*}$: concatenazione di un qualunque numero di transizioni

Esercitazione

Svolgere la computazione della MdT a 2 nastri per le palindrome sull'input 01010

MdT a k nastri

Formalizzazione input e output, decisione, accettazione ecc.

- $M(x) = \text{"}yes\text{"}$ se $(s, \triangleright, x, \triangleright, \epsilon, \ldots, \triangleright\epsilon) \xrightarrow{M^*} (\text{"}yes\text{"}, \ldots)$
- $M(x) = \text{"}no\text{"}$ se $(s, \triangleright, x, \triangleright, \epsilon, \ldots, \triangleright\epsilon) \xrightarrow{M^*} (\text{"}no\text{"}, \ldots)$
- $M(x) = y$ se $(s, \triangleright, x, \triangleright, \epsilon, \ldots, \triangleright\epsilon) \xrightarrow{M^*} (h, \ldots, \triangleright w_k, u_k)$ e $y = w_k, u_k$
- date queste definizioni, decisione e accettazione di linguaggi, calcolo di funzioni ecc. sono definiti come prima

Risorse di calcolo, potere espressivo ed efficienza

Risorse di calcolo

- Tempo impiegato da un algoritmo / MdT
  - numero di passi richiesti dalla computazione
- Spazio (di memoria) impiegato da un algoritmo / MdT
  - quantità di nastro richiesta dalla computazione

Time complexity

Il tempo richiesto da M sull'input x è t sse

$$(s, \triangleright, x, \triangleright, \epsilon, \ldots, \triangleright, \epsilon) \xrightarrow{M^t} (H, \ldots)$$

dove $H \in \{h, \text{"}yes\text{"}, \text{"}no\text{"}\}$.

Se $M(x) = \nearrow$ si conviene che $t = \infty$.

M opera in tempo $f(n)$ sse per ogni input x, il tempo richiesto da M su x è $\leq f(|x|)$.[1]

[1] $|x|$ denota la *lunghezza* della stringa x

La classe di complessità $\mathbf{TIME}(f(n))$

- È una classe di *problemi*: linguaggi $L \subset (\Sigma \setminus \{\sqcup\})^*$

Definizione: $\mathbf{TIME}(f(n))$

$L \in \mathbf{TIME}(f(n))$ sse L è deciso da una MdT multinastro M che opera in tempo $f(n)$

- Osservate che <u>non</u> si richiede che M sia l'algoritmo *più efficiente* per L
- quindi se $L \in \mathbf{TIME}(f(n))$ allora $L \in \mathbf{TIME}(g(n))$ per tutte le g che crescono più in fretta di f
  - Prova: si modifica M per farle "perdere tempo" prima di terminare

La classe di complessità $\mathbf{TIME}(f(n))$

- Esempio: $L =$ Stringhe binarie palindrome
- Col primo algoritmo i passi sono $(n+1) + n + (n-1) + \cdots + 1$ quindi la MdT opera in tempo

  $$f(n) = \frac{(n+2)(n+1)}{2}$$

  quindi $L \in \mathbf{TIME}\left(\frac{(n+2)(n+1)}{2}\right)$
- È una *worst case analysis* (se la stringa non è palindroma la MdT si può fermare prima...)
- Nota: si può dimostrare che *qualunque* MdT a 1 nastro che decide L opera in tempo $\Omega(n^2)$ (vedere i Problemi 2.8.4 e 2.8.5 sul libro). Tuttavia...

La classe di complessità $\mathbf{TIME}(f(n))$

- Esempio 2: $L =$ Stringhe binarie palindrome
- Col <u>secondo</u> algoritmo (2 nastri) la MdT (nel caso peggiore) opera in tempo

$$f(n) = 3n + 3$$

quindi vale anche $L \in \mathbf{TIME}\big(3n + 3\big)$

- Questo mostra che con le MdT multinastro si può avere uno *speedup* quadratico

Massimo speedup con MdT multinastro

- Lo speedup quadratico è anche il massimo possibile *in generale*

Teorema (speedup multinastro)

Per ogni MdT a k nastri M che opera in tempo $f(n)$,
esiste una MdT (a 1 nastro) M' che opera in tempo $O(f(n)^2)$
ed è equivalente a M:

$$\text{per ogni input } x,\ M(x) = M'(x).$$

Dimostrazione dello speedup multinastro (I)

- Sia $M = (K, \Sigma, \delta, s)$. Definiamo $M' = (K', \Sigma', \delta', s)$.
  - notare che lo stato iniziale è lo stesso
- Con il singolo nastro di M' possiamo simulare i k nastri di M "concatenandoli"
  - servono nuovi simboli "separatori" $\triangleright', \triangleleft$
- Corrispondenza delle configurazioni di M e M':
  - $(s, w_1, u_1, \ldots, w_k, u_k)$
  - $(s, \triangleright, w_1' u_1 \triangleleft w_2' u_2 \triangleleft \ldots w_k' u_k \triangleleft \triangleleft)$

dove

- ogni w_i' si ottiene da w sostituendo $\triangleright$ con $\triangleright'$
  - per poterci "passare sopra"
- il doppio $\triangleleft\triangleleft$ finale serve a marcare la fine della stringa di M'

Dimostrazione dello speedup multinastro (II)

- Poi dobbiamo indicare la posizione del cursore per ogni nastro
  - per ogni simbolo originale $\sigma \in \Sigma$ introduciamo una versione "marcata" $\underline{\sigma}$ che indica dov'è il cursore
  - e definiamo $\underline{\Sigma} = \{\underline{\sigma} \mid \sigma \in \Sigma\}$

Dimostrazione dello speedup multinastro (III)

- Inizio della simulazione: da $\rhd x$ all'encoding dei k nastri
  1. traslare l'input x a destra di una posizione inserendo $\rhd'$ all'inizio ($\rhd x \leadsto \rhd \rhd' x$)
     - servono $|\Sigma| + 1$ nuovi stati dedicati a questo
     - ogni stato "memorizza" il simbolo precedente, lo copia in loco, sposta la testina a destra e va nello stato corrispondente al simbolo sovrascritto
  2. appendere la stringa $\lhd(\rhd'\lhd)^{k-1}\lhd$ che rappresenta gli altri $k-1$ nastri vuoti
     - servono $2k$ nuovi stati

Dimostrazione dello speedup multinastro (IV)

- Simulare le transizioni di M
  1. con una prima passata del nastro si raccolgono i simboli letti dai k cursori
     - serve un nuovo stato $q\_\sigma_1\_\ldots\_\sigma_i$ per ogni stato originale $q \in K$ e ogni possibile sequenza di simboli $\sigma_1 \ldots \sigma_i$ con $i \leq k$
     - ciascuno di essi "ricorda" i simboli letti fino a quel momento
     - scandisce il nastro verso destra e quando trova un $\underline{\sigma_{i+1}}$ passa a $q\_\sigma_1\_\ldots\_\sigma_{i+1}$
     - la scansione si ferma a $\lhd\lhd$
  2. poi sulla base dello stato $q\_\sigma_1\_\ldots\_\sigma_k$ raggiunto si modifica il nastro secondo quanto specificato da δ
     - scandire di nuovo il nastro cercando i simboli marcati $\underline{\sigma_i}$
     - modificare i simboli adiacenti come previsto da δ
     - se la testina va su $\lhd$ (oltre il limite del nastro):
     - si rimpiazza $\lhd$ con un nuovo simbolo $\underline{\lhd}$ e si cerca $\lhd\lhd$
     - si ritorna indietro, traslando tutto 1 posizione a destra fino a ritrovare $\underline{\lhd}$, che viene traslato come $\lhd$ e sostituito con $\underline{\sqcup}$

Dimostrazione dello speedup multinastro (V)

- Ripetere il procedimento finchè si raggiunge un stato finale
- Poi convertire il risultato nel formato delle MdT a 1 nastro:
  - traslare a sinistra la rappresentazione dell'ultimo nastro cancellando i precedenti
  - eliminare il $\lhd\lhd$ finale

Dimostrazione dello speedup multinastro (VI)

- Analisi di complessità
  - Poichè M termina in tempo $\leq f(|x|)$ (per ipotesi),
  - anche la lunghezza dei nastri non può superare $f(x)$
    - si aggiunge al più una nuova cella ad ogni passo
  - Quindi la lunghezza massima della stringa di M' non supera

$$k(f(|x|)+1)+1 \quad \text{(considerando i } \lhd\text{)}$$

  - Simulare una transizione richiede
    - 1 scansione avanti-indietro per leggere i simboli attuali: $2k(f(|x|)+1)+2$ passi
    - similmente per modificare ognuno dei k nastri simulati
  - Il totale è $O(k^2 f(|x|)^2)$, ovvero $O(f(|x|)^2)$
    - perchè k è una costante (non dipende dall'input x ma solo dall'algoritmo M)

QED

Commenti sullo speedup quadratico

- Esistono altre simulazioni, vedere i problemi alla fine del Cap. 2
  - ad es. la cella i-esima di M' contiene la k-upla di simboli contenuti nella i-esima cella dei k nastri
- Le MdT sono un modello di computazione potente e stabile
  - l'aggiunta di nastri non permette di risolvere più problemi
  - e aumenta l'efficienza al più in modo polinomiale
- Esistono altre estensioni delle MdT (che non vedremo) e altri modelli di calcolo (alcuni riportati nel libro)
  - Le MdT continuano a risolvere la stessa classe di problemi
  - con una perdita di efficienza al più polinomiale

Il modello di calcolo RAM (Random Access Machine)

- Le macchine RAM hanno un array di *registri* capaci di contenere interi arbitrariamente grandi
- Il set di istruzioni è:

Instruction	**Operand**	**Semantics**
READ	j	$r_0 := i_j$
READ	$\uparrow j$	$r_0 := i_{r_j}$
STORE	j	$r_j := r_0$
STORE	$\uparrow j$	$r_{r_j} := r_0$
LOAD	x	$r_0 := x$
ADD	x	$r_0 := r_0 + x$
SUB	x	$r_0 := r_0 - x$
HALF		$r_0 := \lfloor \frac{r_0}{2} \rfloor$
JUMP	j	$\kappa := j$
JPOS	j	**if** $r_0 > 0$ **then** $\kappa := j$
JZERO	j	**if** $r_0 = 0$ **then** $\kappa := j$
JNEG	j	**if** $r_0 < 0$ **then** $\kappa := j$
HALT		$\kappa := 0$

Il modello di calcolo RAM (Random Access Machine)

Teorema (2.5)

Se una RAM calcola una funzione ϕ in tempo $f(n)$ allora esiste una MdT a 7 nastri che calcola ϕ in tempo $O(f(n)^3)$.

(Si può anche dimostrare che le RAM possono simulare qualunque MdT)

- Di nuovo, risolvono la stessa classe di problemi con una differenza polinomiale in efficienza
- Rispetto a un modello (RAM) molto vicino ai calcolatori reali
  - in effetti più potente perchè i registri hanno capienza illimitata

Altri modelli di calcolo importanti

- Negli anni '30 sono stati introdotti numerosi modelli di calcolo
  - motivati dal tentativo di automatizzare la dimostrazione di teoremi
  - collegato allo sforzo di sistematizzazione e formalizzazione della matematica
- Oltre alle MdT e RAM:
  - I Sistemi di Post
  - Le funzioni generali ricorsive di Kleene
  - Il λ-calcolo di Church
    - preso come fondazione dal linguaggio funzionale LISP
- Hanno tutti la stessa potenza delle MdT che li possono simulare con un overhead polinomiale

La tesi di Church

Non dimostrabile perchè non formalizzabile

- Ogni tentativo ragionevole di modellare matematicamente algoritmi e performance
  - programma finito
  - strutture dati finite ma illimitate
- produce un modello di calcolo che esprime le stesse funzioni delle MdT
- con una perdita di efficienza al più polinomiale

Le classi **TIME, SPACE** e le costanti

Le costanti non contano

- Nel valutare la complessità dei problemi e la performance degli algoritmi si usa molto la notazione $O()$ che ignora le costanti
- Non si fa solo per comodità: in un certo senso *il modello di costo è insensibile alle costanti*
  - teorema di speedup e suo analogo per lo spazio
- Ne approfittiamo per modellare l'utilizzo di memoria degli algoritmi

Linear speedup theorem

Teorema (linear speedup)

Se $L \in \mathbf{TIME}(f(n))$ allora per ogni $\varepsilon > 0$, $L \in \mathbf{TIME}(f'(n))$, dove

$$f'(n) = \varepsilon f(n) + n + 2$$

- Ad es. se $L \in \mathbf{TIME}(150n^2)$ allora anche $L \in \mathbf{TIME}(n^2 + n + 2)$
  - nota: entrambi $O(n^2)$
  - la notazione $O()$ è l'unica "significativa"

Linear speedup theorem - Dimostrazione (I)

- Sia $M = (K, \Sigma, \delta, s)$ una MdT a k nastri che decide L in tempo $f(n)$
- Simuleremo M con una MdT $M' = (K', \Sigma', \delta', s')$ a k stati (se $k > 1$) oppure a 2 stati se $k = 1$
- Idea principale: usare 1 simbolo per rappresentarne m della macchina originale
  - come allungare la parola dell'hardware
  - sappiamo già che velocizza le operazioni...
  - m dipende solo da ε e verrà stimato più tardi
- Poi M' potrà eseguire m istruzioni di M in 2 soli passi
  - si ingrandiscono esponenzialmente alfabeto e programma

Linear speedup theorem - Dimostrazione (II)

- Σ', oltre a Σ, contiene le m-uple di simboli di Σ
  - $\Sigma' = \Sigma \cup \Sigma^m$
- Fase preliminare: comprimere l'input x in blocchi lunghi m e copiarlo sulla seconda stringa
  - s' scandisce il primo nastro
  - usiamo stati $s'\_\sigma_1\_\ldots\_\sigma_i$ $(i < m)$ per "ricordare" i simboli incontrati
  - arrivati a m, se lo stato attuale è $s'\_\sigma_1\_\ldots\_\sigma_{m-1}$ e il simbolo corrente σ_m, scriviamo il *singolo simbolo* $(\sigma_1 \ldots \sigma_m) \in \Sigma'$ sul secondo nastro
  - se il nastro finisce prima di m, riempiamo i simboli mancanti con $\sqcup$
  - dopo $m\lceil \frac{|x|}{m} \rceil + 2$ passi, il 2^o nastro contiene una versione di x ridotta di un fattore m

Linear speedup theorem - Dimostrazione (III)

- da ora in avanti il 2^o nastro viene trattato come quello dell'input e inizia la simulazione vera e propria
  - se durante la compressione si è cancellato x, il primo nastro ora è vuoto e può essere utilizzato al posto del secondo
- nel resto della computazione, le celle dei nastri contengono solo blocchi di m simboli
  - che vengono letti e scritti in singoli passi
- M' simulerà iterativamente m passi di M alla volta con ≤ 6 dei propri passi
  - ciascuna di queste iterazioni viene chiamata *stage* (fase)

Linear speedup theorem - Dimostrazione (IV)

- Le configurazioni di M sono rappresentate con l'aiuto di appositi stati di M' che "memorizzano" lo stato di M e la posizione dei suoi cursori:
  - se M è nello stato q
  - e l'i-esimo cursore è sulla cella ℓ_i dell'i-esimo nastro (per $1 \leq i \leq k$)
  - allora M' si trova nello stato $(q, \ell_1 \bmod m, \ldots, \ell_k \bmod m)$
    - $\ell_i \bmod m$ è l'*offset* dell'i-esimo cursore nel suo blocco
    - notare che questi stati sono in numero *finito*: $|K|km$
  - inoltre l'i-esimo cursore di M' si trova sulla cella numero $\lceil \frac{\ell_i}{m} \rceil$

Linear speedup theorem - Dimostrazione (V)

- Adesso, per ogni nastro, M' "memorizza" in un apposito stato i 2 simboli a sinistra e a destra del cursore (oltre allo stato q di M)
  - con uno spostamento a sinistra, due a destra e uno a sinistra
  - lo stato: $(q, (\sigma'_1 \dots \sigma'_m), (\sigma''_1 \dots \sigma''_m))$
- ora lo stato corrente e il simbolo corrente "rappresentano" un intervallo di $3m$ simboli originali intorno al cursore di M
  - notare che in m mosse il cursore di M *non può uscire da questo intervallo*
- le successive m mosse di M modificheranno il blocco centrale e *uno* dei due adiacenti
  - richiede 2 sole mosse di M'
  - δ' dice direttamente come modificare i due blocchi (e quali)

Linear speedup theorem - Dimostrazione (VI)

- Analisi di complessità
  - se M termina in $f(|x|)$ passi,
  - allora M' termina entro il numero di passi:

$$\begin{array}{ll} |x| + 2 + & \text{fase di compressione} \\ +\, 6\left\lceil \frac{f(|x|)}{m} \right\rceil & \text{6 passi ogni } m \end{array}$$

- Ora per dimostrare il teorema basta scegliere $m = \left\lceil \frac{6}{\varepsilon} \right\rceil$
  - così M' termina entro $|x| + 2 + \varepsilon f(|x|)$ passi
  - per cui M' opera in tempo $\varepsilon f(n) + n + 2$

QED

Costanti nello spazio...

- Un risultato analogo vale per lo spazio di memoria richiesto dagli algoritmi
- Naturalmente dobbiamo prima definire le corrispondenti classi di complessità
- Gli approcci più immediati hanno dei difetti

Uso della memoria

Versioni naive

- Prima idea: se $(s, \triangleright, x, \triangleright, \epsilon, \ldots, \triangleright, \epsilon) \xrightarrow{M^*} (H, w_1, u_1, \ldots, w_k, u_k)$
- allora lo spazio utilizzato è la somma delle lunghezze dei nastri
  - tanto non si accorciano mai!
  - quindi lo spazio usato da M su x è $\sum_{i=1}^{k} |w_i u_i|$
- Alternativa: lo spazio usato è la lunghezza del nastro più lungo
  - quindi $\max_{i=1}^{k} |w_i u_i|$
  - è praticamente la stessa cosa a meno di una costante:
  - $\max_{i=1}^{k} |w_i u_i| \leq \sum_{i=1}^{k} |w_i u_i| \leq k \cdot \max_{i=1}^{k} |w_i u_i|$

Problema delle misure di spazio naive

- Il nastro con l'input è sempre lungo almeno $|x| + 1$
- Le misure naive non riescono a distinguere
  - algoritmi che usano variabili ausiliarie lunghe $O(|x|)$
  - da quelli che usano variabili ausiliarie notevolmente più piccole (ad es. $O(\log |x|))$
  - perchè in entrambi i casi la misura sarebbe $O(|x|)$

Problema delle misure di spazio naive

Esempio: riconoscimento palindrome con spazio ausiliario logaritmico (I)

- Le stringhe palindrome possono essere riconosciute da una MdT M con 3 nastri tale che:
  - il primo nastro contiene l'input e *viene solo letto*
  - gli altri due nastri contengono due indici compresi tra 0 e $|x|$ usati per accedere ai caratteri corrispondenti dell'input
  - gli indici sono rappresentati in binario, quindi sono lunghi $\lceil \log_2 |x| \rceil$
- Di nuovo, la macchina lavora in *fasi*
  - Il secondo nastro è inizializzato a 1
  - alla fase i il suo contenuto rappresenterà i

Problema delle misure di spazio naive

Esempio: riconoscimento palindrome con spazio ausiliario logaritmico (II)

- Durante la fase i, prima si cerca il simbolo i-esimo dell'input, x_i, e si "memorizza" con un opportuno stato q_x_i:

    1. il terzo nastro è inizializzato a $j = 1$
    2. viene confrontato con il secondo nastro (scandendoli alla ricerca del primo simbolo diverso)
    3. se $j < i$ allora si sposta il cursore dell'input a destra e si incrementa j (usando le istruzioni per il successore di un numero binario, ricordate?)
    4. quando $j = i$, leggiamo il simbolo attuale x_i e passiamo nello stato q_x_i corrispondente

Problema delle misure di spazio naive

Esempio: riconoscimento palindrome con spazio ausiliario logaritmico (III)

- Poi si cerca il simbolo in posizione $|x| - i + 1$ per confrontarlo con x_i

  1. spostiamo la testina alla fine dell'input e passiamo in un apposito stato r_x_i che cerca $x_{|x|-i+1}$
  2. inizializziamo nuovamente $j = 1$
  3. r_x_i sposta il cursore dell'input a sinistra e incrementa j finchè $j = i$; poi sposta il cursore a destra
  4. se il simbolo x_j sotto il cursore dell'input è diverso dall'x_i memorizzato negli stati utilizzati fino a quel momento, la MdT termina con "no"
  5. altrimenti si riportano i cursori all'inizio, si incrementa i e si passa alla fase successiva

Problema delle misure di spazio naive

Esempio: riconoscimento palindrome con spazio ausiliario logaritmico (IV)

- Se a un certo punto il simbolo x_i è $\sqcup$, allora l'input è stato completamente scandito e la MdT termina con "yes"
- Confrontare con il vecchio algoritmo per le palindrome:
  1. copiava l'input nel secondo nastro
  2. scandiva i due nastri in senso inverso
  3. la "variabile ausiliaria" (secondo nastro) era lunga $|x| + 1$
- I due algoritmi usano quantità di spazio ausiliario molto diverse
  - quello vecchio ne usa esponenzialmente di più di quello nuovo
- Vorremmo una misura di utilizzo dello spazio che rendesse questo visibile

Verso una misura dello spazio

Definizione: MdT con input e output

Sono MdT a k nastri tali che:

- $k \geq 2$
- il primo nastro (input) non viene mai modificato (read only)
- l'ultimo nastro (output) viene scandito solo verso destra (write only)
- il cursore del primo nastro non va mai oltre il $\sqcup$ successivo all'input

Verso una misura dello spazio

- Le MdT con input e output hanno la stessa espressività e performance delle altre (non modificano le classi di complessità):

Teorema: Potenza delle MdT con input e output

- Per ogni MdT a k nastri M che opera in tempo $f(n)$
- esiste una MdT con input e output e con $k+2$ nastri M'
- che ha lo stesso output e opera in tempo $O(f(n))$

- Prova: M' copia l'input sul secondo nastro; poi simula M usando i nastri $2 \cdots k+1$; infine copia il nastro $k+1$ sul $k+2$. QED

Risorse di calcolo

Formalizzazione

Definizione: Space complexity

Sia M una MdT a k nastri. In generale, se

$$(s, \triangleright, x, \triangleright, \epsilon, \ldots, \triangleright, \epsilon) \xrightarrow{M^*} (H, w_1, u_1, \ldots, w_k, u_k)$$

$(H \in \{h, \text{"}yes\text{"}, \text{"}no\text{"}\})$, lo spazio richiesto da M sull'input x è

$$\sum_{i=1}^{k} |w_i u_i|.$$

Tuttavia, se M è una MdT con input e output, lo spazio richiesto è $\sum_{i=2}^{k-1} |w_i u_i|$.

M opera in spazio $f(n)$ sse per ogni input x, lo spazio richiesto è $\leq f(|x|)$.

La classe di complessità $\mathbf{SPACE}(f(n))$

- È una classe di *problemi*: linguaggi $L \subset (\Sigma \setminus \{\sqcup\})^*$

Definizione: $\mathbf{SPACE}(f(n))$

$L \in \mathbf{SPACE}(f(n))$ sse L è deciso da una MdT multinastro M che opera in spazio $f(n)$

- Di nuovo, le costanti non contano

Teorema

Se $L \in \mathbf{SPACE}(f(n))$, allora per ogni $\varepsilon > 0$, $L \in \mathbf{SPACE}(\varepsilon f(n) + 2)$.

- Prova: simile al linear speedup (vedere Problema 2.8.14)

Tirando le somme

- Abbiamo formalizzato in termini matematici
  - problemi (linguaggi)
  - algoritmi (MdT)
  - le risorse di calcolo *tempo* e *spazio* utilizzate dagli algoritmi
  - le classi di problemi $\mathbf{TIME}(f(n))$ e $\mathbf{SPACE}(f(n))$ (classi di complessità)
- Abbiamo dimostrato che queste classi di complessità sono insensibili alle costanti
- Quindi d'ora in avanti solo notazione $O()$

Capitolo di riferimento

Papadimitriou

- Parte I, Capitolo 2, tutti i paragrafi tranne il 2.7 e le prove del 2.6