Linguaggi e complessità

Indice

Fino ad ora abbiamo studiato le proprietà di decidibilità dei linguaggi, ossia, dato un linguaggio $L \subseteq \Sigma^*$, ci siamo interessati all'esistenza di una macchina di Turing che, con input $x \in \Sigma^*$, fosse in grado di determinare in un numero *finito* di istruzioni l'appartenenza o meno di x ad L. Potrebbe, però, accadere che, seppure di lunghezza finita, la sequenza di istruzioni eseguite da una macchina di Turing per accettare (o rigettare) il suo input sia molto lunga. Talmente lunga, da superare la durata di una vita umana. Da un punto di vista pratico, dunque, l'esistenza di una macchina siffatta non sembra rendere il linguaggio che essa decide più decidibile di un linguaggio non decibile...

In questa dispensa ci interessiamo alla *quantità* di risorse di cui una macchina di Turing ha bisogno per accettare o rigettare un input, considerando, in particolare, come risorse il numero di istruzioni ed il numero di celle del nastro che occorrono per portare a termine una computazione. Ci occuperemo, quindi, di linguaggi decidibili classificandoli rispetto alla quantità di risorse sufficienti a decidere l'appartenenza o meno a ciascuno di essi di ogni parola possibile.

6.1 Misure di complessità

Una misura di complessità è una funzione che associa ad ogni macchina di Turing (determinitica o non deterministica) e ad ogni suo possibile input un valore numerico che corrisponde al "costo" (vedremo fra breve i diversi significati che assoceremo a tale termine) della computazione della macchina sull'input considerato. Ad esempio, le misure di complessità *dtime* e *ntime*, che studieremo in dettaglio nel seguito, contano il numeri di passi di una computazione eseguita, rispettivamente, da una macchina di Turing deterministica e da una macchina di Turing non deterministica: in questo caso, dunque, il costo di una computazione è il numero di passi di cui essa consiste.
Affinché una funzione f possa essere considerata una misura di complessità, essa deve soddisfare le due seguenti proprietà, note come *assiomi di Blum*:

1) la funzione f è definita solo per computazioni che terminano - intuitivamente, questo significa che, se una computazione $T(x)$ non termina, non è sensato considerare che tale computazione abbia come costo un valore finito;

2) f deve essere una funzione calcolabile, ossia, deve esistere una macchina di Turing $\mathscr{M}$ che, ricevendo in input una macchina di Turing T ed un suo input x, calcola $f(T,x)$ ogniqualvolta $f(T,x)$ è definita (cioè, ogniqualvolta $T(x)$ termina) - intuitivamente, questo significa che, il costo di una computazione $T(x)$ (che termina) dobbiamo poterlo calcolare effettivamente.

Definiamo, ora, formalmente, le misure di complessità che utilizzeremo nell'ambito di questo corso. Iniziamo con le misure di complessità che si riferiscono a computazioni deterministiche. Per ogni macchina di Turing deterministica T, di tipo riconoscitore o di tipo trasduttore, definita su un alfabeto Σ, e per ogni $x \in \Sigma^*$ tali che $T(x)$ termina, definiamo le due funzioni seguenti:

$dtime(T,x)$ = numero di istruzioni eseguite da $T(x)$

e

$dspace(T,x)$ = numero di celle utilizzate da $T(x)$.

Dimostriamo ora che le funzioni $dtime$ e $dspace$ soddisfano i due assiomi di Blum.

1) Per definizione, per ogni macchina di Turing deterministica T e per ogni $x \in \Sigma^*$, $dtime(T,x)$ e $dspace(T,x)$ sono definite se e solo se $T(x)$ termina.

2) Consideriamo una modifica U_{dtime} della macchina di Turing universale U descritta nella Dispensa 2. U_{dtime} è ottenuta dotando U di un nastro aggiuntivo N_5 da utilizzare, con input T ed x, come contatore del numero di istruzioni della computazione $T(x)$: $U_{dtime}(T,x)$ si comporta come $U(T,x)$ con l'unica differenza che, al termine del passo 5 descritto nella Dispensa 2 (ossia, dopo avere eseguito una istruzione della macchina T su input x ed essersi preparata ad eseguire l'istruzione successiva) scrive un 1 sul nastro N_5 e muove a destra la testina su tale nastro. Al termine della computazione $U_{dtime}(T,x)$ (se essa termina) il nastro N_5 conterrà, codificato in unario, il numero di passi eseguiti dalla computazione $T(x)$: dunque, $dtime$ è una funzione calcolabile. La dimostrazione che $dspace$ è una funzione calcolabile è simile ed è, pertanto, omessa.

Analogamente alle funzioni *dtime* e *dspace*, possiamo definire le funzioni *ntime* e *nspace* che si riferiscono a computazioni non deterministiche. Per ogni macchina di Turing non deterministica (di tipo riconoscitore) NT, definita su un alfabeto Σ, e per ogni $x \in \Sigma^*$ tali che $NT(x)$ termina nello stato di accettazione, definiamo le due funzioni seguenti:

$$ntime(NT,x) = \text{minimo numero di istruzioni eseguite da una computazione deterministica accettante di } NT(x)$$

e

$$nspace(NT,x) = \text{minimo numero di celle utilizzate da una computazione deterministica accettante di } NT(x).$$

Osserviamo che le funzioni *ntime* e *nspace* sono definite solo per computazioni che accettano (a differenza delle funzioni *dtime* e *dspace* definite per computazioni che, genericamente, terminano). Questa differenza è collegata alla asimmetria delle definizioni di accettazione e di rigetto di una macchina non deterministica: ricordiamo, infatti, che, mentre affinché $NT(x)$ termini nello stato di accettazione è sufficiente che una delle sue computazioni deterministiche accetti (e, quindi, quella che accetta per prima interrompe l'intera computazione non deterministica), affinché $NT(x)$ termini nello stato di rigetto è necessario che tutte le sue computazioni deterministiche rigettino (e, quindi, per poter terminare nello stato di rigetto la macchina deve attendere che tutte le sue computazioni deterministiche terminino). Tale differenza, come vedremo più avanti, si riflette nel diverso comportamento dei linguaggi complemento, rispetto ai linguaggi che complementano, nelle classi di complessità deterministiche e non deterministiche.

Le funzioni *ntime* e *nspace* soddisfano anch'esse i due assiomi di Blum. La dimostrazione di questa affermazione è simile a quella mostrata per le classi *dtime* e *dspace*; essa presenta, però, qualche difficoltà tecnica legata alla necessità di utilizzare, diciamo così, una macchina di Turing universale non deterministica ed è, pertanto, omessa.

Mostriamo, ora, le relazioni che intercorrono fra le misure di complessità appena definite.

Teorema 6.1: *Sia T una macchina di Turing deterministica, definita su un alfabeto Σ (non contenente il simbolo $\Box$) e un insieme degli stati Q, e sia $x \in \Sigma^*$ tale che $T(x)$ termina. Allora,*

$$dspace(T,x) \leq dtime(T,x) \leq dspace(T,x)|Q|(|\Sigma|+1)^{dspace(T,x)}.$$

Analogamente, sia NT una macchina di Turing non deterministica, definita su un alfabeto Σ e un insieme degli stati Q, e sia $x \in \Sigma^*$ tale che $NT(x)$ termina nello stato di accettazione. Allora,

$$nspace(NT,x) \leq ntime(NT,x) \leq nspace(NT,x)|Q|(|\Sigma|+1)^{nspace(NT,x)}.$$

Dimostrazione: Banalmente, una computazione deterministica che termina in k passi non può utilizzare più di k celle del nastro: dunque, $dspace(T,x) \leq dtime(T,x)$. Inoltre, detta $NT_1(x)$ la computazione deterministica di NT che accetta x in $ntime(NT,x)$ passi, tale computazione utilizza al più $ntime(NT,x)$ celle di memoria. Quindi, il minimo numero di celle utilizzate da una computazione deterministica accettante di $NT(x)$ è al più $ntime(NT,x)$, ossia, $nspace(T,x) \leq ntime(T,x)$.

Il numero di stati globali per una computazione di una macchina di Turing (deterministica o non deterministica), definita su un alfabeto Σ e un insieme degli stati Q, che utilizza h celle del nastro è al più $h|Q|(|\Sigma|+1)^h$: infatti, $(|\Sigma|+1)^h$ sono tutte le possibili parole di h simboli di $\Sigma \cup \{\Box\}$, ossia, tutte le possibili configurazioni delle h celle utilizzate, e per ognuna di tali configurazioni la testina può trovarsi su ognuna delle h celle e la macchina può essere in ognuno dei Q stati interni. Ovviamente, il numero di passi eseguiti dalla macchina nella sua computazione non può eccedere tale numero di stati globali e, quindi, $dtime(T,x) \leq dspace(T,x)|Q|(|\Sigma|+1)^{dspace(T,x)}$ e $ntime(NT,x) \leq nspace(NT,x)|Q|(|\Sigma|+1)^{nspace(NT,x)}$. □

Nella Dispensa 2 abbiamo introdotto il modello di macchina di Turing a $k > 1$ nastri (più l'eventuale nastro di output) e abbiamo mostrato la sua equivalenza, dal punto di vista della calcolabilità, con il modello di macchina ad un nastro (Paragrafo 2.4). Poi, nel Paragrafo 2.5, abbiamo mostrato l'equivalenza, sempre dal punto di vista della calcolabilità, fra macchine che operano sull'afabeto binario e macchine che operano su alfabeti di cardinalità maggiore di 2. I prossimi due teoremi estendono, in un certo senso, i risultati dei Paragrafi 2.4 e 2.5 alla teoria della complessità mostrando che i modelli sono *polinomialmente correlati*.

Teorema 6.2: *Per ogni macchina di Turing deterministica T a $k > 1$ nastri definita su un alfabeto Σ esistono una macchina di Turing deterministica ad un nastro T_1 definita sull'alfabeto Σ e tre costanti $c_1, c_2, c_3 \in \mathbb{N}$ tali che, per ogni*

$x \in \Sigma^*$ l'esito della computazione $T(x)$ coincide con l'esito della computazione $T_1(x)$ e inoltre, per ogni $x \in \Sigma^*$ tale che $T(x)$ e $T_1(x)$ terminano,

$$dtime(T_1,x) \leq c_1\ dtime(T,x)^{c_2} + c_3 \qquad e \qquad dspace(T_1,x) \leq c_1\ dspace(T,x)^{c_2} + c_3.$$

Analogamente, per ogni macchina di Turing non deterministica NT a $k > 1$ nastri definita su un alfabeto Σ esistono una macchina di Turing non deterministica ad un nastro NT_1 definita sull'alfabeto Σ e tre costanti $c_1, c_2, c_3 \in \mathbb{N}$ tali che, per ogni $x \in \Sigma^*$ l'esito della computazione $NT(x)$ coincide con l'esito della computazione $NT_1(x)$ e inoltre, per ogni $x \in \Sigma^*$ tale che $NT(x)$ e $NT_1(x)$ accettano,

$$ntime(NT_1,x) \leq c_1\ ntime(NT,x)^{c_2} + c_3 \qquad e \qquad nspace(NT_1,x) \leq c_1\ nspace(NT,x)^{c_2} + c_3.$$

.

Schema della dimostrazione: È sufficiente ricordare la simulazione di una macchina a $k > 1$ nastri mediante una macchina ad un nastro descritta nel Paragrafo 2.4 della Dispensa 2. □

Teorema 6.3: *Per ogni macchina di Turing deterministica T ad un nastro definita su un alfabeto Σ, con $|\Sigma| > 2$, esistono una macchina di Turing deterministica ad un nastro T_1 definita sull'alfabeto $\{0,1\}$, una codifica binaria b degli elementi di Σ e tre costanti $c_1, c_2, c_3 \in \mathbb{N}$ tali che, per ogni $x \in \Sigma^*$ l'esito della computazione $T(x)$ coincide con l'esito della computazione $T_1(b(x))$ e inoltre, per ogni $x \in \Sigma^*$ tale che $T(x)$ e $T_1(b(x))$ terminano,*

$$dtime(T_1,b(x)) \leq c_1\ dtime(T,x)^{c_2} + c_3 \qquad e \qquad dspace(T_1,b(x)) \leq c_1\ dspace(T,x)^{c_2} + c_3.$$

Analogamente, per ogni macchina di Turing non deterministica NT ad un nastro definita su un alfabeto Σ, con $|\Sigma| > 2$, esistono una macchina di Turing non deterministica ad un nastro NT_1 definita sull'alfabeto $\{0,1\}$, una codifica binaria b degli elementi di Σ e tre costanti $c_1, c_2, c_3 \in \mathbb{N}$ tali che, per ogni $x \in \Sigma^*$ l'esito della computazione $NT(x)$ coincide con l'esito della computazione $NT_1(b(x))$ e inoltre, per ogni $x \in \Sigma^*$ tale che $NT(x)$ e $NT_1(b(x))$ accettano,

$$ntime(NT_1,x) \leq c_1\ ntime(NT,x)^{c_2} + c_3 \qquad e \qquad nspace(NT_1,x) \leq c_1\ nspace(NT,x)^{c_2} + c_3.$$

Schema della dimostrazione: È sufficiente ricordare quanto descritto nel Paragrafo 2.5 della Dispensa 2. □

Osserviamo che, invece, mentre il Teorema 2.1 mostra l'equivalenza del modello non deterministico e del modello deterministico dal punto di vista della calcolabilità, la sua dimostrazione presenta una simulazione di una macchina non deterministica NT con grado di non determinismo h mediante una macchina deterministica T tale che, se NT accetta in tempo $t(n)$, allora T opera in tempo $t'(n) \approx h^{t(n)}$, come chiariremo nel prossimo paragrafo; in effetti, è aperta la questione se le macchine deterministiche e le macchine non deterministiche siano o meno polinomialmente correlate. Come vedremo, questa questione è alla base dei più importanti problemi aperti nella Teoria della Complessità Computazionale.

Concludiamo questo paragrafo con un teorema che mette in relazione le misure di complessità *dtime* e *dspace* appena definite con i programmi scritti nel linguaggio **PascalMinimo**.

Teorema 6.4: *Sia $f : \Sigma_i \to \Sigma_o$ una funzione calcolabile. Per ogni programma $\mathscr{P}$ scritto in linguaggio* **PascalMinimo** *che calcola f esistono una macchina di Turing deterministica $T_{\mathscr{P}}$ di tipo trasduttore che calcola f e sei costanti intere $a_1, a_2 a_3, , b_1, b_2, b_3 \in \mathbb{N}$ tali che, per ogni $x \in \Sigma_i^*$ tale che $f(x)$ è definita:*

1) *$dtime(T,x) \leq a_1\ \alpha(x)^{a_2} + a_3$, dove $\alpha(x)$ è il numero totale di istruzioni di assegnazione e di confronti fra variabili semplici[1] eseguite da $\mathscr{P}$ con input x,*

2) *$dspace(T,x) \leq b_1\ \beta(x)^{b_2} + b_3$, dove $\beta(x)$ è il numero di celle di memoria utilizzate da $\mathscr{P}$ con input x.*

Schema della dimostrazione: La dimostrazione segue da una analisi della dimostrazione del Teorema 3.5 della Dispensa 3 ed è omessa. □

[1] Con il termine *variabile semplice* ci riferiamo ad una variabile non strutturata, ossia, una variabile che può contenere un singolo elemento dell'alfabeto di lavoro.

Si osservi che, essendo le macchine di Turing di tipo riconoscitore particolari trasduttori che calcolano funzioni booleane, il precedente teorema vale anche quando si considerano programmi scritti in linguaggio **PascalMinimo** che decidono circa l'appartenenza di una parola ad un linguaggio.

6.2 Funzioni time- e space-constructible

Introduciamo, in questo paragrafo, due sottoclasse delle funzioni totali calcolabili contenenti le funzioni da $\mathbb{N}$ a $\mathbb{N}$ che, per essere calcolate, utilizzano una quantità di risorse (tempo di calcolo o spazio di memoria) proporzionale al loro valore. Come vedremo nel prossimo paragrafo, tali sottoclassi giocano un ruolo importante nella definizione di classi di complessità significative.

Definizione 6.1: Una funzione totale calcolabile $f : \mathbb{N} \to \mathbb{N}$ è time-constructible se esiste una macchina di Turing T di tipo trasduttore che, preso in input un intero n espresso in notazione unaria (ossia, come sequenza di n '1'), scrive sul nastro output il valore $f(n)$ in unario e $dtime(T,n) \in \mathbf{O}(f(n))$.

Definizione 6.2: Una funzione totale calcolabile $f : \mathbb{N} \to \mathbb{N}$ è space-constructible se esiste una macchina di Turing T di tipo trasduttore che, preso in input il valore n espresso in notazione unaria, scrive sul nastro output il valore $f(n)$ in unario e $dspace(T,n) \in \mathbf{O}(f(n))$.

Nel resto del paragrafo, ci occuperemo di dimostrare la time-constructibility di alcune semplici funzioni.

6.2.1 La funzione n^k è time-constructible

Sia k una costante positiva. Poiché è banale dimostrare che la funzioni $t(n) = 1$ è time-constructible, in quanto segue limiteremo la nostra attenzione al caso $k \geq 1$.

Vogliamo definire una macchina di Turing T che, preso in input un valore n espresso in notazione unaria (ossia, 1^n, una successione di 1 lunga n) calcola il valore n^k in notazione unaria (ossia, 1^{n^k}, una sequenza di 1 lunga n^k).

Poiché l'unica operazione direttamente disponibile su una tale macchina di Turing è la concatenazione di 1 (che, intuitivamente, corrisponde ad una addizione), è necessario far corrispondere in T a ciascuna moltiplicazione nm una sequenza $m+m+\ldots+m$ di n addizioni.

La macchina T che andiamo a definire lavora in k fasi distinte in modo tale che durante la fase i viene calcolato n^i. Essa utilizza 3 nastri infiniti, dei quali sono utilizzate soltanto le porzioni ad indirizzi positivi, a partire dalle celle di indirizzo 0, ed il cui significato è di seguito descritto:

- il nastro N_1 contiene l'input;
- durante la fase i, con $i = 2,\ldots,k$, il nastro N_2 contiene il risultato parziale $m = n^{i-1}$ calcolato durante la fase $i-1$, mentre è vuoto durante la fase 1;
- al termine della fase i, per $i = 1,\ldots,k$, il nastro N_3 contiene l'output della fase i, ossia, n^i. Al termine della computazione, il risultato n^k si troverà, dunque, sul nastro N_3.

L'insieme degli stati Q utilizzati da T è partizionato in k sottoinsiemi, come descritto di seguito:

- un sottoinsieme costituito dallo stato iniziale q_0 e dagli stati q_{10} e q_{11}, utilizzati durante la fase 1;
- un sottoinsieme costituito dal solo stato finale q_F;
- $k-1$ sottoinsiemi $Q_2, Q_3, \ldots, Q_k$ tali che, per $i = 1,\ldots,k$, durante la fase i la macchina si trova in uno stato $q \in Q_i$. Come chiarito meglio nel seguito, a sua volta l'insieme Q_i è costituito dagli stati:

  $$q_i,\ q_{i0},\ q_{i1},\ q_i^{SCRIVE},\ q_i^{IND1},\ q_i^{IND2}.$$

All'inizio della computazione, con la macchina che si trova nello stato q_0, ha inizio la fase 1: in tale stato, se la testina di N_1 legge blank allora la macchina entra nello stato finale e la computazione termina, altrimenti, se la testina di N_1 legge 1, la macchina entra nello stato q_{10} ed il contenuto del nastro N_1 viene copiato sul nastro N_3:

- fino a quando la testina del nastro N_1 legge 1, viene scritto un 1 sul nastro N_3, le testine di N_1 ed N_3 avanzano di una posizione e la macchina rimane nello stato q_{10};
- quando la testina di N_1 legge $\Box$, la macchina entra nello stato q_{11} e le testine dei nastri N_1 ed N_3 si muovono a sinistra di una posizione. Ora il nastro N_1 viene riavvolto: nello stato q_{11}, se la testina di N_1 legge 1 allora riscrive 1 e si muove a sinistra, e la macchina rimane nello stato q_{11}; se invece la testina di N_1 legge $\Box$ allora la testina del nastro N_1 riscrive $\Box$ e si muove a destra (posizionandosi sul primo 1), e la macchina entra in un novo stato che dipende dal valore di k: se $k > 1$ allora la macchina entra nello stato q_2 (per iniziare la fase 2), altrimenti entra nello stato finale q_F.

A questo punto, il nastro N_3 contiene il valore $n = n^1$ e la sua testina è posizionata sull'ultimo 1; inoltre, le testine di N_1 e di N_2 sono posizionate sulle rispettive celle di indirizzo 0. Ora, come abbiamo visto,

- se $k > 1$, con la macchina nello stato q_2, ha inizio la seconda fase al termine della quale il nastro N_3 conterrà il valore n^2;
- se $k = 1$, allora la macchina si trova nello stato finale q_F.

Quanto appena descritto è un invariante di tutte le fasi della computazione: al termine della fase i, con $i \geq 1$,

- il nastro N_3 contiene il valore n^i e la sua testina è posizionata sul carattere 1 più a destra,
- il nastro N_1 contiene il valore n e la sua testina è posizionata sulla cella di indirizzo 0,
- la testina del nastro N_2 è posizionata sulla cella di indirizzo 0,
- infine:
  - se $k > i$, la macchina entra nello stato q_{i+1} e ha inizio la fase $i+1$;
  - se $k = i$, allora la macchina entra nello stato finale q_F con l'output correttamente memorizzato sul nastro N_3.

Assumiamo, dunque, che l'invariante sia vero al termine della fase i, con $i \geq 1$, e descriviamo le operazioni che avvengono nella fase i.

All'inizio della fase $i \geq 2$, la macchina si trova nello stato q_i. Innanzi tutto, il contenuto del nastro N_3 (pari a n^{i-1}) viene copiato sul nastro N_2 e le testine di N_2 ed N_3 vengono riavvolte all'inizio dei rispettivi nastri:

1. nello stato q_i:
   - 1.a) se la testina di N_3 legge 1, la testina di N_2 scrive 1 e si muove a destra, la testina N_3 si muove a sinistra, e la macchina rimane nello stato q_i;
   - 1.b) se la testina di N_3 legge $\Box$, le testine di N_2 ed N_3 riscrivono $\Box$ e si muovono, rispettivamente, a sinistra e a destra, e la macchina entra nello stato q_{i0} (osserviamo che la testina di N_3 è ora posizionata sulla cella di indirizzo 0);
2. nello stato q_{i0}:
   - 2.a) se la testina di N_2 legge 1, riscrive 1 e si muove a sinistra, e la macchina rimane nello stato q_{i0};
   - 2.b) se la testina di N_2 legge $\Box$, riscrive $\Box$ e si muove a destra, e la macchina entra nello stato q_{i1} (ora anche la testina di N_2 è posizionata sulla cella di indirizzo 0).

Il passo successivo, nella fase i, consiste nel calcolare il valore n^i: questo avviene scrivendo sul nastro N_3 la concatenazione n volte del contenuto del nastro N_2 (che è pari ad n^{i-1}), cioè, $n \cdot n^{i-1} = n^i$. A questo scopo, la macchina esegue sostanzialmente il seguente algoritmo:

3. se la macchina è nello stato q_{i1}

   3.1) se la testina di N_1 legge 1 (assumiamo che essa stia scandendo la cella h del nastro N_1) allora la macchina entra nello stato q_i^{SCRIVE}, e si prepara a concatenare per la h-esima volta il contenuto del nastro N_2 al contenuto del nastro N_3;

   3.2) se la testina di N_1 legge $\square$ allora (ha terminato la n-esima concatenazione della fase i e sul nastro N_3 è scritto n^i):

      3.2.1) se $i < k$ la macchina entra nello stato q_i^{IND1} e si prepara a riavvolgere il nastro N_1 per iniziare la fase $i+1$;

      3.2.2) se $i = k$ la macchina entra nello stato q_F;

4. quando la macchina è nello stato q_i^{SCRIVE}, essa concatena al contenuto del nastro N_3 il contenuto del nastro N_2:

   4.1) fino a quando la testina di N_2 legge 1, la testina di N_3 scrive 1 e si muove a destra e la testina di N_2 si muove a destra senza modificare il simbolo letto,

   4.2) quando la testina di N_2 legge $\square$ allora la concatenazione del contenuto di N_2 al contenuto di N_3 è terminata (cioè, assumendo che la testina del nastro N_1 sia posizionata sulla cella $h \leq n$, è stato calcolato hn^{i-1}) e il nastro N_2 deve essere riavvolto: allo scopo, la macchina entra nello stato q_i^{IND2};

5. quando la macchina è nello stato q_i^{IND2}, la testina di N_2 si muove a sinistra fino a quando raggiunge la cella di indirizzo 0; a questo punto la testina del nastro N_1 avanza di una posizione e la macchina entra nello stato q_{i1} (cioè, si prepara a concatenare n^{i-1} ad hn^{i-1} sul nastro N_3, e, quindi, a calcolare $(h+1)n^{i-1}$);

6. quando la macchina è nello stato q_i^{IND1}, la testina di N_1 si muove a sinistra fino a posizionarsi sulla cella di indirizzo 0; a questo punto, la fase i è terminata e la macchina entra nello stato q_{i+1}.

Resta da calcolare il numero di passi eseguiti da T.

La fase 1 (in cui la macchina è nello stato q_0) serve a copiare l'input n sul nastro N_3 e a riavvolgere le testine del nastro N_1 e quindi richiede $2n$ passi.

Vediamo ora il numero di passi richiesti dalla fase i, con $2 \leq i \leq k$. Durante la fase i, innanzi tutto il contenuto del nastro N_3 viene copiato sul nastro N_2, in n^{i-1} passi, ed i due nastri vengono riavvolti. Successivamente, la macchina passa n volte dallo stato q_{i1} allo stato q_i^{SCRIVE} (una volta per ogni 1 sul nastro input): ciascuna volta, concatena al contenuto di N_3 il valore n^{i-1} contenuto in N_2 (in n^{i-1} passi) e poi entra nello stato q_i^{IND2} e riavvolge il nastro N_2 (in n^{i-1} passi). Infine, riavvolge il nastro N_1 (in n passi). In definitiva, la fase $i \geq 2$ costa

$$2n^{i-1} + n(n^{i-1} + n^{i-1}) + n = 2n^{i-1} + 2n^i + n.$$

Sommando il costo di tutte le fasi, ed osservando che il costo della fase 1 è $2n$, possiamo quindi concludere che il numero di passi eseguiti da $T(n)$ è

$$2n + \sum_{2 \leq i \leq k} (2n^i + 2n^{i-1} + n) = 2n + (k-1)n + 2\sum_{2 \leq i \leq k} n^i + 2\sum_{2 \leq i \leq k} n^{i-1} \leq (k+1)n + 2(k-1)n^k + 2(k-1)n^{k-1} \leq (5k-3)n^k.$$

Poiché k è costante, questo dimostra che n^k è time-constructible

Precondizioni:	sul nastro N_1 è scritto il valore n codificato in unario; la testina di N_1 è posizionata sul primo 1
1	$n_3 \leftarrow n_1$;
2	$i \leftarrow 2$;
3	**while** $(i \leq k)$ **do begin**
4	$n_2 \leftarrow n_3$;
5	$h \leftarrow 1$;
6	**while** $(h \leq n_1)$ **do begin**
7	$n_3 \leftarrow n_3 \oplus n_2$;
8	$h \leftarrow h+1$;
9	**end**
10	$i \leftarrow i+1$;
11	**end**

Tabella 6.1: Algoritmo corrispondente alla macchina di Turing che calcola n^k.

6.2.2 La funzione n^n è time-constructible

Nel dimostrare che n^k è una funzione time-constructible, l'ipotesi "k costante" viene utilizzata in due modi diversi. Innanzi tutto, la cardinalità dell'insieme degli stati della macchina che abbiamo progettato per calcolare n^k è $5(k-1)+4$; in secondo luogo, abbiamo dimostrato che il numero di passi richiesto da tale macchina è limitato superiormente da $(4k-2)n^k$. Pertanto, l'approccio che abbiamo utilizzato per dimostrare che n^k è una funzione time-constructible non può essere direttamente esteso alla funzione n^n: infatti, non è pensabile definire una macchina di Turing con un numero di stati dipendente dalla lunghezza dell'input, né il numero di passi $(4n-2)n^n$ è proporzionale ad n^n. Comunque, prenderemo spunto da tale approccio per tentare di dimostrare che anche n^n è una funzione time-constructible. A questo scopo, ricordiamo quanto fatto per il calcolo della funzione n^k traducendo il progetto della macchina di Turing in un algoritmo codificato nel linguaggio ad alto livello **PascalMinimo**: se indichiamo con una variabile i il numero della fase, con h una variabile che indica la posizione della testina sul nastro N_1, con una variabile n_j il contenuto del nastro N_j $(j = 1,2,3)$, con l'istruzione "$n_u \leftarrow n_v$;" la copia del contenuto del nastro N_v sul nastro N_u , e con "$\oplus$" l'operatore di concatenazione di due stringhe, tale algoritmo è descritto in Tabella 6.1.

Si osservi la differenza fra le condizioni del ciclo **while** esterno $(i \leq k)$ e del ciclo **while** interno $(h \leq n_1)$: mentre nel primo caso il valore k è costante (ed, in effetti, è codificato nel numero di stati interni della macchina), nel secondo caso il valore di h deve essere confrontato con un valore dipendente dall'input (e quindi memorizzato sul nastro n_1).

Ora che abbiamo chiarito il diverso ruolo giocato da valori costanti e da valori variabili, risulta piuttosto semplice modificare il precedente algoritmo in modo tale che funzioni per calcolare la potenza n-sima (e, dunque, variabile) di n: è sufficiente sostituire la condizione $i \leq k$ del ciclo **while** esterno con la condizione $i \leq n_1$. Tuttavia, nella traduzione da tale programma scritto nel linguaggio ad alto livello all'insieme di quintuple che descrivono una macchina di Turing, dobbiamo utilizzare il nastro N_1 sia come contatore del numero di fase in cui si trova la macchina (che, non essendo possibile codificare mediante un insieme di stati interni, dovrebbe essere codificata in qualche modo su tale nastro) che come contatore del numero di volte in cui abbiamo sommato il contenuto del nastro N_2 a sé stesso (ciò che accade nel ciclo **while** più interno). Pertanto, per semplificarci la vita, utilizziamo un nastro supplementare, N_4, come contatore del numero di fase e riserviamo ad N_1 il ruolo di contatore del numero di volte in cui abbiamo sommato il contenuto del nastro N_2 a sé stesso. In definitiva, l'algoritmo per il calcolo di n^n, scritto ancora nel linguaggio ad alto livello, è descritto in Tabella 6.2, in cui n_4 denota il contenuto del nastro N_4. Lasciamo come (semplice) esercizio la traduzione di tale algoritmo in un insieme di quintuple che descrivono una macchina di Turing T: osserviamo soltanto che, in questo caso possono essere sufficienti solo 7 stati.

Resta da calcolare il numero di passi eseguiti da T. Le istruzioni che precedono il ciclo **while** esterno e che servono a copiare l'input n sui nastri N_3 ed N_4 richiedono n passi di T (la scrittura sui due nastri è simultanea) e ulteriori n passi servono a riavvolgere il nastro N_4 per eseguire l'istruzione "$i \leftarrow 2$;". Vediamo ora quanti passi richiede l'iterazione i-esima del ciclo **while** esterno, con $2 \leq i \leq n$. Durante tale iterazione, innanzi tutto viene controllato il valore della variabile i (condizione del ciclo) in un singolo passo; poi, in $2n^{i-1}$ passi, viene copiato il contenuto del nastro N_3 sul nastro N_2 (istruzione "$n_2 \leftarrow n_3$;") ed il nastro N_2 viene riavvolto; in seguito, è necessario riavvolgere il nastro N_1 (istruzione "$h \leftarrow 1$;") e questo costa n passi. Viene inoltre eseguito il ciclo **while** interno: per n volte, in un passo viene controllato il valore di h (condizione del ciclo), in n^{i-1} passi al contenuto di N_3 viene concatenato il valore

Precondizioni:	sul nastro N_1 è scritto il valore n codificato in unario; la testina di N_1 è posizionata sul primo 1
1	$n_3 \leftarrow n_1$;
2	$n_4 \leftarrow n_1$;
3	$i \leftarrow 2$;
4	**while** $(i \leq n_4)$ **do begin**
5	$\quad n_2 \leftarrow n_3$;
6	$\quad h \leftarrow 1$;
7	$\quad$ **while** $(h \leq n_1)$ **do begin**
8	$\quad\quad n_3 \leftarrow n_3 \oplus n_2$;
9	$\quad\quad h \leftarrow h+1$;
10	$\quad$ **end**
11	$\quad i \leftarrow i+1$;
12	**end**

Tabella 6.2: Algoritmo corrispondente alla macchina di Turing che calcola n^n.

n^{i-1} contenuto in N_2 ed in altri n^{i-1} passi viene riavvolto il nastro N_2 per eseguire la concatenazione dell'iterazione successiva (istruzione "$n_3 \leftarrow n_3 \oplus n_2$;"). Al termine dell'iterazione i-esima del ciclo **while** esterno, in un passo la testina del nastro N_1 avanza di una posizione (istruzione "$i \leftarrow i+1$;"). In definitiva, l'algoritmo proposto richiede un numero di passi pari a

$$
\begin{aligned}
2n+\sum_{2\leq i\leq n}\left[2n^{i-1}+n+n\left(1+n^{i-1}+n^{i-1}\right)+1\right] &= 2n+\sum_{2\leq i\leq n}\left[2n^{i}+2n^{i-1}+2n+1\right] \\
&= 3n+2n^2+2\sum_{2\leq i\leq n}\left[n^{i}+n^{i-1}\right] \\
&\leq 3n+2n^2+4\sum_{2\leq i\leq n} n^{i} \\
&= 3n+2n^2+4\sum_{0\leq i\leq n} n^{i}-4n-4 \\
&\leq 2n^2+4\sum_{0\leq i\leq n} n^{i} \\
&= 2n^2+4\frac{n^{n+1}-1}{n-1} \\
&\leq 2n^2+4\frac{n^{n+1}}{n-1} \\
&\leq 2n^2+4\frac{n^{n+1}}{\frac{n}{2}} \\
&= 2n^2+8n^n \;\leq\; 10n^n.
\end{aligned}
$$

e questo dimostra che n^n è una funzione time-constructible.

6.3 Classi di complessità di linguaggi

In questo paragrafo iniziamo il processo di classificazione dei linguaggi in base alle risorse tempo e spazio sufficienti alla loro decisione/accettazione. Sia $f : \mathbb{N} \to \mathbb{N}$ una funzione totale calcolabile. Diremo che un linguaggio $L \subseteq \Sigma^*$ *è deciso in tempo (spazio) deterministico* $f(n)$ se esiste una macchina di Turing deterministica T che decide L e tale che, per ogni $x \in \Sigma^*$, $dtime(T,x) = f(|x|)$ ($dspace(T,x) = f(|x|)$). Analogamente, diremo che un linguaggio L *è accettato in tempo (spazio) non deterministico* $f(n)$ se esiste una macchina di Turing non deterministica NT che accetta L e tale che, per ogni $x \in L$, $ntime(NT,x) = f(|x|)$ ($nspace(NT,x) = f(|x|)$).

Una *classe di complessità* è definita mediante una *funzione totale calcolabile* $f : \mathbb{N} \to \mathbb{N}$, che chiameremo *funzione limite della classe*, e rappresenta una classe di linguaggi decidibili o accettabili da una macchina di Turing (deterministica o non deterministica) che utilizza una quantità di risorse limitata da f:

$$\begin{aligned}
\text{DTIME}[f(n)] &= \{L \subseteq \Sigma^* : \Sigma \text{ è un alfabeto finito e } L \text{ è un linguaggio deciso da una macchina di Turing} \\
&\quad \text{deterministica } T \text{ tale che } \forall x \in \Sigma^* \ [\ dtime(T,x) \in \mathbf{O}(f(|x|)\ ]\},
\end{aligned}$$

$$\begin{aligned}
\text{NTIME}[f(n)] &= \{L \subseteq \Sigma^* : \Sigma \text{ è un alfabeto finito e } L \text{ è un linguaggio accettato da una macchina di Turing} \\
&\quad \text{non deterministica } NT \text{ tale che } \forall x \in L \ [\ ntime(NT,x) \in \mathbf{O}(f(|x|)\ ]\},
\end{aligned}$$

$$\begin{aligned}
\text{DSPACE}[f(n)] &= \{L \subseteq \Sigma^* : \Sigma \text{ è un alfabeto finito e } L \text{ è un linguaggio deciso da una macchina di Turing} \\
&\quad \text{deterministica } T \text{ tale che } \forall x \in \Sigma^* \ [\ dspace(T,x) \in \mathbf{O}(f(|x|)\ ]\},
\end{aligned}$$

$$\begin{aligned}
\text{NSPACE}[f(n)] &= \{L \subseteq \Sigma^* : \Sigma \text{ è un alfabeto finito e } L \text{ è un linguaggio accettato da una macchina di Turing} \\
&\quad \text{non deterministica } NT \text{ tale che } \forall x \in L[\ nspace(NT,x) \in \mathbf{O}(f(|x|))]\}.
\end{aligned}$$

Si osservi, nelle definizioni precedenti, che le classi di complessità deterministiche sono definite sulla base di linguaggi *decisi* da macchine di Turing deterministiche, mentre le classi di complessità non deterministiche sono definite sulla base di linguaggi *accettati* da macchine di Turing non deterministiche. Fra le classi di complessità appena introdotte sussistono le relazioni enunciate nei teoremi che seguono.

Teorema 6.5: *Per ogni funzione totale calcolabile* $f : \mathbb{N} \to \mathbb{N}$*,*

$$DTIME[f(n)] \subseteq NTIME[f(n)] \text{ e } DSPACE[f(n)] \subseteq NSPACE[f(n)].$$

Dimostrazione: È sufficiente osservare che una macchina di Turing deterministica è una particolare macchina di Turing non deterministica avente grado di non determinismo pari ad 1 e che ogni parola decisa in k passi è anche accettata in k passi. □

Teorema 6.6: *Per ogni funzione totale calcolabile* $f : \mathbb{N} \to \mathbb{N}$*,*

$$DTIME[f(n)] \subseteq DSPACE[f(n)] \text{ e } NTIME[f(n)] \subseteq NSPACE[f(n)].$$

Dimostrazione: La prova di questo teorema segue direttamente dal Teorema 6.1. Sia $L \subseteq \Sigma^*$ tale che $L \in \text{DTIME}[f(n)]$: allora, esiste una macchina di Turing deterministica T che decide L e tale che, per ogni $x \in \Sigma^*$, $dtime(T,x) \in \mathbf{O}(f(|x|))$. Poiché $dspace(T,x) \leq dtime(T,x)$, questo implica che $dspace(T,x) \in \mathbf{O}(f(|x|))$ e che, dunque, $L \in \text{DSPACE}[f(n)]$.

Analogamente, se $L \in \text{NTIME}[f(n)]$, esiste una macchina di Turing non deterministica NT che accetta L e che, per ogni $x \in L$, $ntime(NT,x) \in \mathbf{O}(f(|x|))$. Per le stesse considerazioni del caso deterministico, da ciò segue che $L \in \text{NSPACE}[f(n)]$. □

Teorema 6.7: *Per ogni funzione totale calcolabile* $f : \mathbb{N} \to \mathbb{N}$*,*

$$DSPACE[f(n)] \subseteq DTIME[2^{\mathbf{O}(1)f(n)}] \text{ e } NSPACE[f(n)] \subseteq NTIME[2^{\mathbf{O}(1)f(n)}].$$

Dimostrazione: Anche in questo caso, la prova segue direttamente dal Teorema 6.1. Sia $L \subseteq \Sigma^*$ tale che $L \in \text{DSPACE}[f(n)]$: allora, esiste una macchina di Turing deterministica T che decide L e tale che, per ogni $x \in \Sigma^*$, $dspace(T,x) \in \mathbf{O}(f(|x|))$. Poiché

$$dtime(T,x) \leq dspace(T,x)|Q|(|\Sigma|+1)^{dspace(T,x)} = 2^{\log dspace(T,x)}|Q|2^{dspace(T,x)\log(|\Sigma|+1)} \leq |Q|2^{[1+\log(|\Sigma|+1)]dspace(T,x)},$$

questo implica che $dtime(T,x) \in \mathbf{O}(2^{\mathbf{O}(1)f(|x|)})$ e che, dunque, $L \in \text{DIME}[2^{\mathbf{O}(1)f(n)}]$.

La dimostrazione per il caso non deterministico è analoga ed è, pertanto, omessa. □

Osserviamo che le relazioni fra classi di complessità dimostrate sino ad ora sono valide per ogni funzione totale e calcolabile mediante la quale la classe è definita. Il prossimo teorema, nel quale viene individuata una relazione fra una classe deterministica ed una non deterministica, richiede, invece, che la funzione che definisce la classe di complessità non deterministica sia time-comstructible.

Teorema 6.8: *Per ogni funzione time-constructible* $f : \mathbb{N} \to \mathbb{N}$,

$$NTIME[f(n)] \subseteq DTIME[2^{\mathbf{O}(f(n))}].$$

Dimostrazione: Sia $L \subseteq \Sigma^*$ tale che $L \in \text{NTIME}(f(n))$; allora esistono una macchina di Turing non deterministica NT che accetta L e una costante h tali che, per ogni $x \in L$, $ntime(NT,x) \leq hf(|x|)$. Inoltre, poiché f è time-constructible, esiste una macchina di Turing (di tipo trasduttore) T_f che, con input la rappresentazione in unario di un intero n, calcola la rappresentazione in unario di $f(n)$ in tempo $\mathbf{O}(f(n))$.

Indichiamo con k il grado di non determinismo di NT (e ricordiamo che k è una costante, indipendente dall'input) e utilizziamo di nuovo la tecnica della simulazione per definire una macchina di Turing deterministica T che simuli il comportamento di NT: su input x, T simula in successione *tutte* le computazioni deterministiche di $NT(x)$ di lunghezza $hf(|x|)$. Più in dettaglio, la macchina T con input x opera come di seguito descritto.

1) Simula la computazione $T_f(|x|)$: per ogni carattere di x, scrive sul nastro N_2 un carattere '1' (ossia, scrive $|x|$ '1' sul nastro N_2) e, in seguito, calcola $f(|x|)$ scrivendolo sul nastro N_3. Infine, concatena h volte il contenuto del nastro N_3 ottenendo il valore $hf(|x|)$.

2) Simula, una alla volta, tutte le computazioni deterministiche di $NT(x)$ di lunghezza $hf(|x|)$ utilizzando, per ciascuna computazione, la posizione della testina sul nastro N_3 come contatore.

Poiché, se $x \in L$, $ntime(NT,x) \leq hf(|x|)$ allora o in $hf(|x|)$ passi $NT(x)$ termina nello stato di accettazione oppure $x \notin L$. Quindi, se dopo aver simulato tutte le computazioni deterministiche di $NT(x)$ di lunghezza $hf(|x|)$, T non ha raggiunto lo stato di accettazione (in accordo alla simulazione di $NT(x)$), allora può correttamente entrare nello stato di rigetto. Questo prova che T decide L.

Il numero di computazioni deterministiche di $NT(x)$ di lunghezza $hf(|x|)$ è $k^{hf(|x|)}$ e ciascuna di esse viene simulata da T in $\mathbf{O}(f(|x|))$ passi. Poiché il passo 1) descritto sopra richiede $\mathbf{O}(f(|x|)$ passi, questo prova che

$$dtime(T,x) \in \mathbf{O}(f(|x|)k^{hf(|x|)}) \subseteq \mathbf{O}(2^{\mathbf{O}(f(|x|))}).$$

Infine, in virtù del Teorema 6.2, esiste una macchina T_1 ad un nastro tale che, per ogni input x, $o_{T_1}(x) = o_T(x)$ e

$$dtime(T_1,x) \leq dtime(t,x)^c \subseteq \mathbf{O}(2^{\mathbf{O}(f(|x|))}).$$

Questo conclude la dimostrazione che $L \in \text{DTIME}[2^{\mathbf{O}(f(|x|))}]$. □

Accanto alle classi DTIME, NTIME, DSPACE, NSPACE possiamo considerare i loro complementi:

$$\begin{aligned}\text{coDTIME}[f(n)] \quad = \quad & \{L \subseteq \Sigma^* : L^c \text{ è deciso da una macchina di Turing deterministica } T \\ & \wedge \forall x \in \Sigma^* \; [\; dtime(T,x) \in \mathbf{O}(f(|x|)\;]\},\end{aligned}$$

$$\begin{aligned}\text{coNTIME}[f(n)] \quad = \quad & \{L \subseteq \Sigma^* : L^c \text{ è accettato da una macchina di Turing non deterministica } NT \\ & \wedge \forall x \in L \; [\; ntime(NT,x) \in \mathbf{O}(f(|x|)\;]\},\end{aligned}$$

$$\begin{aligned}\text{coDSPACE}[f(n)] \quad = \quad & \{L \subseteq \Sigma^* : L^c \text{ è deciso da una macchina di Turing deterministica } T \\ & \wedge \forall x \in \Sigma^* \; [\; dspace(T,x) \in \mathbf{O}(f(|x|)\;]\},\end{aligned}$$

$$\text{coNSPACE}[f(n)] \quad = \quad \{L \subseteq \Sigma^* : L^c \text{ è accettato da una macchina di Turing non deterministica } NT \\ \wedge \forall x \in L[\, nspace(NT,x) \in \mathbf{O}(f(|x|))]\}.$$

Teorema 6.9: *Per ogni funzione totale calcolabile $f : \mathbb{N} \to \mathbb{N}$,*

$$DTIME[f(n)] = coDTIME[f(n)] \text{ e } DSPACE[f(n)] = coDSPACE[f(n)].$$

Dimostrazione: Sia $L \subseteq \Sigma^*$ tale che $L \in \text{coDTIME}[f(n)]$. Allora, $L^c \in \text{DTIME}[f(n)]$ ed esiste una macchina di Turing deterministica T che decide L^c tale che, per ogni $x \in \Sigma^*$, $dtime(T,x) \in \mathbf{O}(f(|x|))$. Sia, allora, T' la macchina di Turing deterministica identica a T in tutto tranne che negli stati finali, che risultano invertiti: $T(x)$ termina nello stato di accettazione se e soltanto se $T'(x)$ termina nello stato di rigetto e viceversa. Quindi, T' decide L e inoltre, per ogni $x \in \Sigma^*$, $dtime(T',x) = dtime(T,x)$. Questo prova che $L \in \text{DTIME}[f(n)]$ e, quindi, poiché L è un generico linguaggio in $\text{coDTIME}[f(n)]$, questo prova che $\text{coDTIME}[f(n)] \subseteq \text{DTIME}[f(n)]$.
In modo simmetrico si dimostra che $\text{DTIME}[f(n)] \subseteq \text{coDTIME}[f(n)]$ e, quindi, $\text{coDTIME}[f(n)] = \text{DTIME}[f(n)]$.

La dimostrazione che $\text{DSPACE}[f(n)] = \text{coDSPACE}[f(n)]$ è analoga ed è, pertanto, omessa. □

Come abbiamo già osservato, le classi di complessità sono state definite sulla base di funzioni totali calcolabili. Questo è ragionevole, in quanto sarebbe inutile sapere che un dato linguaggio è contenuto in una certa classe di complessità se poi non fossimo in grado di calcolare la quantità di risorse richieste per decidere circa l'appartenenza ad esso di una parola. La calcolabilità da sola, però, non permette di evitare strane relazioni fra le classi di complessità, come evidenzieremo nella parte finale di questo paragrafo. Iniziamo ad osservare che, quella enunciata nel prossimo teorema, è una proprietà decisamente ragionevole (in accordo alla classificazione dei linguaggi che ci proponiamo di derivare) delle classi di complessità.

Teorema 6.10: *Per ogni coppia di funzioni totali calcolabili $f : \mathbb{N} \to \mathbb{N}$ e $g : \mathbb{N} \to \mathbb{N}$ tali che definitivamente*[2] *$f(n) \leq g(n)$, $DTIME[f(n)] \subseteq DTIME[g(n)]$, $NTIME[f(n)] \subseteq NTIME[g(n)]$, $DSPACE[f(n)] \subseteq DSPACE[g(n)]$ e $NSPACE[f(n)] \subseteq NSPACE[g(n)]$.*

Dimostrazione: Sia $L \subseteq \Sigma^*$ tale che $L \in DTIME[f(n)]$; allora esiste una macchina di Turing deterministica T che decide L e tale che $dtime(T,x) \in \mathbf{O}(f(|x|)) \subseteq \mathbf{O}(g(|x|))$. Questo significa che $L \in \text{DTIME}[g(n)]$.
Gli altri casi sono dimostrati analogamente. □

Di contro, nella definizione di una teoria della complessità in grado di classificare significativamente i linguaggi in classi di complessità crescente, sarebbe auspicabile che l'inclusione inversa non fosse verificata. Questo è contraddetto dal prossimo teorema.

Teorema 6.11: **Gap Theorem**. *Esiste una funzione totale calcolabile $f : \mathbb{N} \to \mathbb{N}$ tale che $DTIME[2^{f(n)}] \subseteq DTIME[f(n)]$.*

Dimostrazione: Sia $\mathscr{D} \subseteq \}0,1\}^*$ l'insieme dei linguaggi decidibili e $\mathscr{T}$ l'insieme delle macchine di Turing che decidono i linguaggi in $\mathscr{D}$. Poiché $\mathscr{T}$ è un sottoinsieme dell'insieme delle macchine di Turing, esso è numerabile. Consideriamo, allora, una numerazione $T_1, T_2, \ldots$ delle macchine di Turing in $\mathscr{T}$ e definiamo

$$f(n) = \min\{m : \forall i \leq n,\ \forall x \text{ con } |x| = n\ [\ dtime(T_i,x) > 2^{2m}\ \vee\ dtime(T_i,x) < m\ ].$$

La funzione f appena definita è totale. Inoltre, essa è calcolabile. Infatti, per calcolare $f(n)$ utilizziamo il procedimento descritto in Tabella 6.3. Si osservi che, ponendo un limite sul numero di passi che ciascuna macchina può utilizzare, la condizione dell'istruzione **if** alla linea 8 viene valutata in tempo finito. Allora, per dimostrare che f è calcolabile è sufficiente mostrare che il ciclo **while** alla linea 3 termina per ogni valore di n, ossia, che, in tempo finito, verrà trovato un intero m tale che, per ogni una stringa $x \in P_n$ e per , per ogni $i \leq n$, la computazione $T_i(x)$ termina entro $m-1$ passi oppure non termina entro 2^{2m} passi. Allo scopo, osserviamo che, per ogni $i \in \mathbb{N}$, la macchina $T_i(x)$ termina per ogni parola x; sia allora $M_n = \max\{dtime(T_i,x) : i \leq n \wedge |x| = n\}$. Allora, se ciascun valore $m \leq M_n$ ha causato l'interruzione del ciclo **while** alla linea 6 prima di aver esaminato tutte le parole in P_n, il valore $m = M_n + 1$ garantisce che, per ogni parola $x \in P_n$, tutte le computazioni $T_i(x)$ terminino in meno di m istruzioni e che, quindi, la variabile *ricomincia* non assuma il valore `vero` causando, in tal modo, per ogni $i \leq n$, la terminazione del ciclo **while** alla linea 6 solo quando tutte le parole in P_n sono state considerate.

[2] Una proprietà vale *definitivamente* se esiste $n_0 \in \mathbb{N}$ tale che la proprietà vale per ogni $n > n_0$

Input:	$n \in \mathbf{N}$.
1	$m \leftarrow 1$;
2	$i \leftarrow 1$;
3	**while** $(i \leq n)$ **do begin**
4	$ricomincia \leftarrow$ `falso`;
5	$P_n \leftarrow \{x : \ |x| = n\}$;
6	**while** $(P_n \neq \emptyset \ \wedge \ ricomincia =$ `falso`$)$ **do begin**
7	estrai x da P_n;
8	**if** $(DTIME_i(x) \geq m \wedge \ DTIME_i(x) \leq 2^{2m})$ **then begin**
9	$ricomincia \leftarrow$ `vero`;
10	$m \leftarrow m+1$;
11	$i \leftarrow 1$;
12	**end**;
13	**end**;
14	**if** $(ricomincia =$ `falso`$)$ **then** $i \leftarrow i+1$;
15	**end**;
16	**Output**: m.

Tabella 6.3: Algoritmo che calcola la funzione $f(n)$ descritta nel Teorema 6.11 .

Sia $L \in \mathscr{D}$ un linguaggio tale che $L \in DTIME[2^{f(n)}]$. Allora, esiste $k \in \mathbf{N}$ tale che $L = L(T_k)$ (ossia, $T_k \in \mathscr{T}$ decide L) e, per ogni parola $x \in \{0,1\}^*$, $dtime(T_k,x) \in \mathbf{O}(2^{f(|x|)})$.
Consideriamo, inoltre, un intero n tale che $n \geq k$; per definizione della funzione f, per ogni parola y con $|y| = n$, vale che $dtime(T_k,y) > 2^{2f(n)}$ oppure $dtime(T_k,y) < f(n)$.
Poiché $L(T_k) \in \text{DTIME}[2^{f(n)}]$, allora, per ogni parola x, $dtime(T_k,x) \in \mathbf{O}(2^{f(|x|)})$, ossia, esistono due costanti c e n_0 tali che $dtime(T_k,x) \leq c2^{f(|x|)})$ per ogni x tale che $|x| \geq n_0$.
Allora, per quanto appena osservato, per ogni x tale che $|x| \geq \max\{k,n_0\}$ vale

$$\left[\ dtime(T_k,x) > 2^{2f(|x|)}) \ \vee \ dtime(T_k,x) < f(|x|))\ \right] \ \wedge \ dtime(T_k,x) \leq c2^{f(|x|)}). \tag{6.1}$$

Segue dal predicato 6.1 che potrebbe accadere che, per qualche $x \in \{0,1\}^*$ tale che $|x| \geq k$, valga

$$2^{2f(|x|)} < dtime(T_k,x) \leq c2^{f(|x|)}).$$

Osserviamo, però, che, nota la costante c, esiste sempre un valore n_c (costante) tale che $c2^{f(n)} < 2^{2f|x|)}$ per ogni $n \geq n_c$. Allora, $2^{2f(|x|)} < dtime(T_k,x) \leq c2^{f(|x|)})$ è possibile solo se $|x| < n_c$. Quindi, poiché $L(T_k) \in \text{DTIME}[2^{f(n)}]$, detto P_{exp} l'insieme delle parole x tali che $dtime(T_k,x) > 2^{2f(|x|)}$, vale la relazione seguente:

$$|P_{exp}| = |\{x \in \{0,1\}^* : dtime(T_k,x) > 2^{2f(|x|)}\}| \leq |\{x \in \{0,1\}^* : |x| \leq n_c\}| \leq n_c 2^{n_c},$$

ossia, P_{exp} ha dimensione costante. Questo implica che decidere se $x \in \{0,1\}^*$ è contenuta in $L(T_k) \cap P_{exp}$ richiede tempo costante - ossia, esistono una macchina di Turing deterministica T_{exp} che decide $L(T_k) \cap P_{exp}$ e una costante k_{exp} tali che $dtime(T_{exp},x) \leq k_{exp}$[3].
Analogamente, poiché l'insieme P_0 delle parole in $\{0,1\}^*$ di lunghezza al più $\max\{k,n_0\}$ ha dimensione

$$|P_0| \leq \max\{k,n_0\}2^{\max\{k,n_0\}},$$

anche decidere se se $x \in \{0,1\}^*$ è contenuta in $L(T_k) \cap P_0$ richiede tempo costante - ossia, esistono una macchina di Turing deterministica T_0 che decide $L(T_k) \cap P_0$ e una costante k_0 tali che $dtime(T_0,x) \leq k_0$.

Possiamo ora costruire una nuova macchina T' che, con input $x \in \{0,1\}^*$, opera come segue:

1) se $x \in P_0$, allora simula la computazione $T_0(x)$;

[3] L'insieme di dimensione costante $L(T_k) \cap P_{exp}$ è, in qualche modo, cablato nella macchina (ad esempio, utilizzando un apposito insieme di stati per gestire il numero costante di parole in tale insieme).

2) altrimenti, se $x \in P_{exp}$, allora simula la computazione $T_{exp}(x)$;

3) altrimenti simula la computazione $T_k(x)$;

La macchina T' decide $L(T_k)$. Inoltre, poiché decidere se se $x \in P_0$ o se $x \in P_{exp}$ richiede tempo costante, i passi 1) e 2) richiedono tempo costante. Infine, per ogni $x \in \{0,1\}^* - (P_0 \cup P_{exp})$, si ha che $dtime(T',x) = dtime(T_k,x)$: in questo caso, poiché $x \notin P_{exp}$ allora $dtime(T',x) \leq c2^{f(|x|)} \leq 2^{2f(|x|)}$; allora, poiché $|x| \geq \max\{k, n_c\}$, in virtù del predicato 6.1, $dtime(T',x) < f(|x|)$. In conclusione, $L \in \text{DTIME}[f(n)]$, da cui segue l'asserto. □

Si osservi che la funzione costruita nella prova del Gap Theorem non è time- constructible.
Inoltre, le stesse considerazioni fatte sin qui per le classi di complessità temporale possono essere ripetute per le classi di complessità spaziali.

Quindi, al fine di non incorrere in comportamenti "anomali" del tipo evidenziato nel Gap Theorem, le funzioni che vengono utilizzate per definire le classi di complessità rilevanti nell'ambito della Teoria della Complessità sono sempre funzioni time-constructible e space-constructible.

6.4 Perché la notazione asintotica?

Nelle definizioni delle classi di complessità presentate nel Paragrafo 6.3 abbiamo utilizzato la notazione **O**: ad esempio, $L \in \text{DTIME}[f(n)]$ se L viene deciso da una macchina di Turing deterministica in un numero di passi che è in $\mathbf{O}(f(|x|))$, per ogni input x. Questa scelta è motivata dai seguenti due teoremi, per la cui dimostrazione formale si rimanda al libro di testo.

Teorema 6.12: **Compressione lineare.** *Sia $s : \mathbb{N} \to \mathbb{N}$ una funzione tale che, per ogni $n \in \mathbb{N}$, $s(n) \geq n$. Sia $L \subseteq \Sigma^*$ un linguaggio deciso da una macchina di Turing deterministica ad un nastro T in spazio $s(n)$ e sia $c \in \mathbb{N}^+$ una costante. Allora esiste una macchina di Turing ad un nastro T' che decide L tale che $dspace(T',x) \leq |x| + \left\lceil \frac{s(|x|)}{c} \right\rceil$.*

Schema della dimostrazione: La macchina T', con input $x \in \Sigma^*$, lavora in due fasi: durante la prima fase scrive una parola $y \in (\Sigma \cup \Box)^c$ che rappresenta x in forma compressa dopo il primo $\Box$ alla destra di x e cancella x dal nastro sostituendo ogni carattere di x con $\Box$. In dettaglio, il primo carattere di y è costituito dalla concatenazione dei primi c caratteri di x, il secondo carattere di y dalla concatenazione dei successivi c caratteri di x, e così via, fino al carattere h-esimo, con $h = \left\lfloor \frac{|x|}{c} \right\rfloor$; se $h < \frac{|x|}{c}$, y contiene un ultimo carattere, costituito dalla concatenazione degli ultimi $|x| - hc$ caratteri di x e di $c - (|x| - hc)$ caratteri '$\Box$'.

Durante la seconda fase T' simula $T(x)$ lavorando sulla parola y.
Sia Q l'insieme degli stati di T; allora l'insieme Q' degli stati di T' utilizzati durante la seconda fase è l'insieme $Q \times \{1,2,\ldots,c\}$: informalmente, in questo modo, uno stato $(q,j) \in Q'$ di T' rappresenta sia lo stato interno di T che il fatto che la testina di T sta scandendo il j-esimo carattere della parola di c caratteri memorizzati all'interno di una cella di T'.
Allora, definendo $\tau_m = -1$ se $m = s$, $\tau_m = 0$ se $m = f$ e $\tau_m = 1$ se $m = d$, una quintupla $\langle q, \overline{\sigma}, \overline{\sigma}', q', m \rangle$ di T, con $\overline{\sigma} \in \Sigma \cup \{\Box\}$, è trasformata in T' nell'insieme di quintuple descritte di seguito

(1) $\langle (q,j), S, S', (q', j+\tau_m), f \rangle$ $\quad \forall j : 1 < j < c \ \wedge\ \forall\, S \in (\Sigma \cup \Box)^c :$ il j-esimo carattere di S è $\overline{\sigma}\ \wedge$ S' differisce da S solo per il j-esimo carattere che in S' è $\overline{\sigma}'$,

(2) $\langle (q,1), S, S', (q', 1+\tau_m), f \rangle$ $\quad$ se $m \neq s \ \wedge\ \forall\, S \in (\Sigma \cup \Box)^c$: il primo carattere di S è $\overline{\sigma}\ \wedge$ S' differisce da S solo per il primo carattere che in S' è $\overline{\sigma}'$,

(3) $\langle (q,1), S, S', (q', c), s \rangle$ $\quad$ se $m = s \ \wedge \forall\, S \in (\Sigma \cup \Box)^c$: il primo carattere di S è $\overline{\sigma}\ \wedge$ S' differisce da S solo per il primo carattere che in S' è $\overline{\sigma}'$,

(4) $\langle (q,c), S, S', (q', 1+\tau_m), f \rangle$ $\quad$ se $m \neq d \ \wedge \forall\, S \in (\Sigma \cup \Box)^c$: l'ultimo carattere di S è $\overline{\sigma}\ \wedge$ S' differisce da S solo per l'ultimo carattere che in S' è $\overline{\sigma}'$,

(5) $\langle (q,c), S, S', (q', 1), d \rangle$ $\quad$ se $m = d \ \wedge \forall\, S \in (\Sigma \cup \Box)^c$: l'ultimo carattere di S è $\overline{\sigma}\ \wedge$ S' differisce da S solo per l'ultimo carattere che in S' è $\overline{\sigma}'$.

Infine, poiché la porzione di nastro di T' non occupata dalla codifica dell'input x contiene simboli "$\Box$", è necessario gestire separatamente il caso in cui una computazione di T' utilizza una cella per la prima volta eseguendo una quintupla di T in cui si sovrascrive il carattere "$\Box$" che essa contiene con un carattere σ: in tal caso, la computazione

di T' dovrà scrivere il carattere σ preceduto o seguito da $c-1$ caratteri "$\Box$". Più in particolare, si osservi che una computazione di T' utilizza una cella per la prima volta solo in seguito all'esecuzione di una quintupla del tipo (3) o del tipo (5) fra quelle dscritte sopra (ossia, una quintupla che fa muovere la testina di T'); pertanto, la trasformazione in quintuple di T' di ciascuna quintupla di T del tipo $\langle q, \Box, \sigma, q', m\rangle$ è completata dalle quintuple seguenti:

$$\text{a seguito dell'esecuzione di una quintupla del tipo (5):} \begin{cases} \langle (q,1), \Box, \sigma\Box^{c-1}, (q', 1+\tau_m), f\rangle, & \text{se } m \neq s \\ \langle (q,1), \Box, \sigma\Box^{c-1}, (q', c), s\rangle, & \text{se } m = s, \end{cases}$$

$$\text{a seguito dell'esecuzione di una quintupla del tipo (3):} \begin{cases} \langle (q,c), \Box, \Box^{c-1}\sigma, (q', c+\tau_m), f\rangle & \text{se } m \neq d \\ \langle (q,c), \Box, \Box^{c-1}\sigma, (q', 1), d\rangle & \text{se } m = d. \end{cases}$$

Le dimostrazioni formali che T' simula effettivamente T e che T' opera in spazio $s'(n) \leq n + \left\lceil \frac{s(n)}{c} \right\rceil$ sono lasciate per esercizio. □

Teorema 6.13: **Accelerazione lineare.** *Sia $L \subseteq \Sigma^*$ un linguaggio deciso da una macchina di Turing deterministica ad un nastro T tale che, per ogni $x \in \Sigma^*$, $dtime(T,x) = t(|x|)$ e sia $k > 0$ una costante. Allora:*

- *esiste una macchina di Turing ad un nastro T'' tale che T'' decide L e, per ogni $x \in \Sigma^*$, $dtime(T'',x) \leq \frac{t(|x|)}{k} + \boldsymbol{O}(|x|^2)$, e inoltre*
- *esiste una macchina di Turing a due nastri $\overline{T}''$ tale che $\overline{T}''$ decide L e, per ogni $x \in \Sigma^*$, $dtime(\overline{T}'',x) \leq \frac{t(|x|)}{k} + \boldsymbol{O}(|x|)$.*

Schema della dimostrazione: Sia $c = 8k$. Consideriamo la macchina T' descritta nella dimostrazione del Teorema 6.12 ed osserviamo che $dtime(T',x) = dtime(T,x) + \mathbf{O}(|x|^2)$: più in dettaglio, la prima fase di T' richiede $\mathbf{O}(|x|^2)$ passi, e la seconda fase $dtime(T,x)$ passi.

Anche la macchina T'' opera in due fasi: la sua prima fase coincide con la prima fase di T', durante la quale la parola x suo input viene compressa mediante l'alfabeto $(\Sigma \cup \Box)^c$, e poi, durante la sua seconda fase, simula la seconda fase di T'.
Nel corso della seconda fase, T'' deve simulare "tanti" passi di T' mediante "pochi" passi: in particolare, faremo in modo di simulare c passi di T' in un numero costante h di passi di T'', con $h < c$. Notiamo, subito, che non può essere $h = 1$: infatti, se, ad esempio, T' si trova in uno stato (q, j) con $j < c$ e deve simulare c quintuple di T che spostano ciascuna la testina a sinistra di una posizione (modificando, ogni volta, il contenuto delle celle celle scandite) allora T' ha bisogno spostare la propria testina a sinistra (una sola volta); pertanto, anche T'' deve spostare la testina di una posizione a sinistra, e questo richiede almeno due passi: durante il primo passo T'' modifica i primi j simboli nella cella che sta scandendo, e durante il secondo passo T'' modifica gli ultimi $c-j$ simboli nella cella a sinistra di quella che stava scandendo. Un caso analogo si verificherebbe se T dovesse eseguire c spostamenti consecutivi a destra. In altre parole, per potere eseguire c passi di T', T'' potrebbe aver bisogno di conoscere, oltre alla parola y_{att} scritta nella cella che sta scandendo anche la parola y_{sin} scritta nella cella a sinistra della cella che sta scandendo o la parola y_{des} scritta nella cella a destra della cella che sta scandendo. Allora, per potere avere a disposizione sia y_{att} che y_{sin} che y_{des}, prima di eseguire c quintuple consecutive di T' (e, quindi, di T), a partire da uno stato (q, j), T'' esegue i 3 passi seguenti:

- riscrivendo quello che legge, entra nello stato (q, j, sin) e si sposta di una cella a sinistra,
- leggendo y_{sin}, riscrive y_{sin}, entra nello stato (q, j, y_{sin}) e si sposta di una cella a destra (tornando nella posizione di partenza),
- leggendo y_{att}, riscrive y_{att}, entra nello stato (q, j, y_{sin}, y_{att}) e si sposta di una cella a destra (assumiamo che in tale cella sia scritta la parola y_{des}).

A questo punto, T'' è in grado di simulare c passi consecutivi della computazione di T' (o, equivalentemente, di T) utilizzando solo le informazioni presenti nel proprio stato interno e il contenuto della cella sulla quale è posizionata la testina: se la macchina T' si trova nello stato interno (q, j) e la sua testina è posizionata sulla cella che contiene y_{att}, e la cella alla sua sinistra contiene y_{sin}, e la cella alla sua destra contiene y_{des}, ed esegue c istruzioni, allora, al

termine di tali c istruzioni avrà modificato i contenuti di una, due o tre di tali celle (e nessun altra cella) scrivendovi, rispettivamente, y'_{att}, y'_{sin} e y'_{des} (ove $y'_{xxx} = y_{xxx}$ se il contenuto della cella non viene modificato), avrà posizionato la sua testina su una di esse e sarà entrata in un nuovo stato interno (q', j'). Corrispondentemente, T'' esegue i seguenti (al più 4) passi:

- scrive y'_{des} (sovrascrivendo y_{des}), entra nello stato $(q', j', scriviAtt, y'_{sin}, y'_{att})$ e sposta la testina a sinistra,
- scrive y'_{att} (sovrascrivendo y_{att}), entra nello stato $(q', j', scriviSin, y'_{sin})$ e sposta la testina a sinistra,
- scrive y'_{sin} (sovrascrivendo y_{sin}), ed entra in uno stato che dipende da dove viene posizionata la testina di T' al termine delle c istruzioni:
  - se la testina deve essere posizionata sulla cella alla sinistra di quella scandita inizialmente (quella che ora contiene y'_{sin}), allora T'' entra nello stato (q', j') e rimane ferma;
  - se la testina deve essere posizionata sulla cella sulla quale era posizionata inizialmente (quella che ora contiene y'_{att}), allora T'' entra nello stato (q', j') e si muove a destra;
  - se la testina deve essere posizionata sulla cella alla destra di quella scandita inizialmente (quella che ora contiene y'_{des}), allora T'' entra nello stato (q', j', des) e si muove a destra; successivamente, esegue un ultimo passo nel quale riscrive y'_{att}, entra nello stato (q', j') e si muove a destra.

Ora, dopo avere eseguito al più 7 passi, T'' si trova nello stesso stato globale in cui si trova T' dopo avere eseguito c passi. Quindi, ricordando che $c = 8k$, se x è tale che $t(|x|) \geq 56k$,

$$\begin{aligned} dtime(T'',x) &\leq \mathbf{O}(|x|^2) + 7\left\lceil \frac{dtime(T',x)}{c} \right\rceil = \mathbf{O}(|x|^2) + 7\left\lceil \frac{dtime(T,x)}{c} \right\rceil = \mathbf{O}(|x|^2) + 7\left\lceil \frac{t(|x|)}{8k} \right\rceil \\ &\leq \mathbf{O}(|x|^2) + 7\left(\frac{t(|x|)}{8k} + 1\right) = \mathbf{O}(|x|^2) + \frac{7t(|x|)}{8k} + 7 \leq \mathbf{O}(|x|^2) + \frac{7t(|x|)}{8k} + \frac{t(|x|)}{8k} = \frac{t(|x|)}{k} + \mathbf{O}(|x|^2). \end{aligned}$$

Se, invece se x è tale che $t(|x|) < 56k$, allora

$$dtime(T'',x) \leq \mathbf{O}(|x|^2) + 7\left\lceil \frac{t(|x|)}{8k} \right\rceil < \mathbf{O}(|x|^2) + 7\left\lceil \frac{56k}{8k} \right\rceil = 49 + \mathbf{O}(|x|^2) = \frac{49k)}{k}\mathbf{O}(|x|^2) < \frac{t(|x|)}{k} + \mathbf{O}(|x|^2).$$

La macchina $\overline{T}''$ opera in maniera molto simile a T'': con input x scritto sul primo nastro, durante la prima fase esegue la compressione di x scrivendo, però, la parola compressa y sul secondo nastro (invece che sul primo nastro a destra di x) e poi, durante la seconda fase, esegue le stesse istruzioni di T'' lavorando soltanto sul secondo nastro.
È semplice dimostrare che, disponendo di un secondo nastro, la prima fase di $\overline{T}''$ richiede $\mathbf{O}(|x|)$ passi. Questa osservazione, unita al fatto che il costo della seconda fase di $\overline{T}''$ coincide con il costo della seconda fase di T'', porta alla conclusione che

$$dtime(\overline{T}'',x) \leq \mathbf{O}(|x|) + \frac{t(|x|)}{k}.$$

Si osservi, infine, che abbiamo codificato nello stato interno tutte le c posizioni della testina su una parola di c caratteri, unito alle singole parole e alle coppie di parole che possono essere lette nelle istruzioni sopra descritte. Allora, la cardinalità dell'insieme Q'' degli stati di T'' (e analogamente per quanto riguarda l'insieme degli stati di $\overline{T}''$) è stata notevolmente aumentata rispetto a quella dell'insieme Q degli stati di T: è, in effetti, semplice verificare che

$$|Q''| \approx 2c[Q| + 2c|Q|(|\Sigma|+1)^c + 2c|Q|(|\Sigma|+1)^{2c}.$$

□

Concludiamo il paragrafo osservando che risultati analoghi a quelli dimostrati nei Teoremi 6.12 e 6.13 valgono anche per le misure di complessità *nspace* e *ntime*.

6.5 Specifiche classi di complessità

Siamo pronti a definire alcune fra le più rilevanti classi di complessità:

- **P** $= \cup_{k\in\mathbb{N}}$ DTIME$[n^k]$: è la classe dei linguaggi decidibili *in tempo deterministico polinomiale*;
- **NP** $= \cup_{k\in\mathbb{N}}$ NTIME$[n^k]$: è la classe dei linguaggi accettabili *in tempo non deterministico polinomiale*;
- **PSPACE** $= \cup_{k\in\mathbb{N}}$ DSPACE$[n^k]$: è la classe dei linguaggi decidibili *in spazio deterministico polinomiale*;
- **NPSPACE** $= \cup_{k\in\mathbb{N}}$ NSPACE$[n^k]$: è la classe dei linguaggi accettabili *in spazio non deterministico polinomiale*;
- **EXPTIME** $= \cup_{k\in\mathbb{N}}$ DTIME$[2^{n^k}]$: è la classe dei linguaggi decidibili *in tempo deterministico esponenziale* ove l'esponente che descrive la funzione limite è un polinomio;
- **NEXPTIME** $= \cup_{k\in\mathbb{N}}$ NTIME$[2^{n^k}]$: è la classe dei linguaggi accettabili *in tempo non deterministico esponenziale* ove l'esponente che descrive la funzione limite è un polinomio.

Le relazioni enunciate nel seguente corollario sono conseguenze dirette dei teoremi nel paragrafo 6.3

Corollario 6.1: *Valgono le seguenti relazioni di inclusione:*

$$\textbf{\textit{P}} \subseteq \textbf{\textit{NP}} \text{ e } \textbf{\textit{PSPACE}} \subseteq \textbf{\textit{NPSPACE}}; \tag{6.2}$$

$$\textbf{\textit{P}} \subseteq \textbf{\textit{PSPACE}} \text{ e } \textbf{\textit{NP}} \subseteq \textbf{\textit{NPSPACE}}; \tag{6.3}$$

$$\textbf{\textit{PSPACE}} \subseteq \textbf{\textit{EXPTIME}} \text{ e } \textbf{\textit{NPSPACE}} \subseteq \textbf{\textit{NEXPTIME}}; \tag{6.4}$$

$$\textbf{\textit{NP}} \subseteq \textbf{\textit{EXPTIME}}. \tag{6.5}$$

Dimostrazione: Caso per caso, vediamo il teorema da cui ciascuna inclusione deriva:

1) le inclusioni (6.2) sono conseguenza diretta del Teorema 6.5;
2) le inclusioni (6.3) sono conseguenza diretta del Teorema 6.6;
3) le inclusioni (6.4) sono conseguenza diretta del Teorema 6.7;
4) l'inclusione (6.5) è conseguenza diretta del Teorema 6.8.

□

Ricordiamo che, nel Paragrafo 6.3, abbiamo definito le classi di complessità complemento di classi date. Così, accanto alle classi introdotte all'inizio di questo paragrafo, possiamo indicare i loro complementi:

$$co\mathbf{P} = \{L \subseteq \Sigma^* : \Sigma \text{ è un alfabeto finito e } L^c \in \mathbf{P}\},$$

$$co\mathbf{NP} = \{L \subseteq \Sigma^* : \Sigma \text{ è un alfabeto finito e } L^c \in \mathbf{NP}\},$$

e così via.

Il seguente corollario è conseguenza diretta del Teorema 6.9:

Corollario 6.2: ***coP = P***.

6.5.1 Un classe di complessità di funzioni: la classe FP

Introduciamo, infine, una classe di complessità di funzioni (e non di linguaggi), la classe delle funzioni calcolabili in tempo deterministico polinomiale:

$$\mathbf{FP} = \bigcup_{k\in\mathbb{N}} \{f : \Sigma_1^* \to \Sigma_2^* : \exists \text{ una macchina di Turing deterministica } T \text{ che calcola } f \text{ e } \forall x \in \Sigma_1^* [\ dtime(T,x) \in \mathbf{O}(n^k)\}.$$

Come vedremo, questa classe riveste importanza non trascurabile nella teoria delle complessità computazionale.

6.6 Inclusioni proprie e improprie e la congettura P $\neq$ NP

Abbiamo, fino ad ora, dimostrato inclusioni non proprie fra le classi di complessità che abbiamo introdotto; per ciascuna di esse non siamo in grado di dimostrare né l'inclusione propria né la coincidenza delle due classi che la costituiscono. Nei prossimi due teoremi, invece, dimostriamo la separazione fra due classi deterministiche, **P** e **EXPTIME**, e la coincidenza di una classe deterministica con la corrispondente non deterministica, ossia, **PSPACE** e **NPSPACE**. Entrambe le dimostrazioni sono basate sulla tecnica della simulazione.

Teorema 6.14: ***P*** $\subset$ ***EXPTIME***

Dimostrazione: Mostriamo, innanzi tutto, che esiste un linguaggio $L \subseteq \{0,1\}^*$ deciso in tempo deterministico $t(n) = n2^{2n}$ tale che $L \notin \text{DTIME}[2^n]$. La dimostrazione è per diagonalizzazione. Ricordiamo che le macchine di Turing (deterministiche) ad un nastro di tipo riconoscitore definite sull'alfabeto $\{0,1\}$ costituiscono un insieme numerabile e lo schema di codifica di una tale macchina è descritto nel Paragrafo 5.1 della Dispensa 5.

Definiamo, allora, il seguente linguaggio:

$$\begin{aligned} L \quad = \quad & \{z \in \{0,1\}^* : z = 1^i 0x \ \wedge\ x \text{ è la codifica binaria di una parola } k \in \mathbb{N} \ \wedge\ k \text{ è la codifica di una macchina} \\ & \text{di Turing deterministica ad un nastro di tipo riconoscitore definita sull'alfabeto } \{0,1\} \\ & \wedge\ T_k(z) \text{ termina in } 2^{2|z|} \text{ passi } \wedge\ T_k(z) \text{ rigetta}\}. \end{aligned}$$

Innanzi tutto, proviamo che $L \notin \text{DTIME}[2^n]$. Infatti, supponiamo per assurdo che $L \in \text{DTIME}[2^n]$, cioè, che esistano una macchina di Turing deterministica ad un nastro T ed una costante $c \in \mathbb{N}$ tali che T decide L in tempo $c2^n$. Sia $\hat{x}$ l'indice di detta macchina e sia $z = 1^i 0\hat{x}$ tale che $c2^{|z|} \leq 2^{2|z|}$: allora, poiché la computazione $T(z) = T_{\hat{x}}(z)$ termina in $c2^{|z|}$ passi, se $T_{\hat{x}}(z)$ termina nello stato di accettazione allora $z \notin L$ (e, dunque $T = T_{\hat{x}}$, che decide L non può accettare z); viceversa, se $T_{\hat{x}}(z)$ termina nello stato di rigetto allora $z \in L$ (e, dunque $T = T_{\hat{x}}$, che decide L non può rigettare z). Da cui l'assurdo.

Mostriamo, ora, che $L \in \text{DTIME}[n2^{2n}]$. Prima di procedere, osserviamo esplicitamente che $f(n) = 2^{2n}$ è una funzione time-constructible e, quindi, esiste una macchina di Turing T_C di tipo trasduttore che, con input un intero n in notazione unaria, scrive sul nastro di output il valore 2^{2n}, in unario, in tempo proporzionale a 2^{2n}.
Consideriamo una variante $\overline{U}$ della macchina di Turing universale U descritta nella Dispensa 2 che utilizza 7 nastri:

- N_1, il nastro su cui memorizziamo l'input z;
- N_2, il nastro su cui verrà scritta la parola x, se z è nella forma $1^i 0x$, ossia, quella che dovrebbe essere la descrizione delle quintuple di T_x;
- N_3, il nastro su cui sarà memorizzato, ad ogni passo, lo stato attuale della macchina T_x;
- N_4, il nastro su cui sarà memorizzato lo stato di rigetto della macchina T_x;
- N_5, il nastro su cui verrà scritto in unario il valore $|z|$;
- N_6, il nastro su cui verrà scritto in unario il valore $2^{2|z|}$;
- N_7, il nastro di lavoro di U.

Le computazioni della macchina $\overline{U}$ sono suddivise in sei fasi.
Durante la prima fase, in un numero di passi proporzionale a $|z|$, $\overline{U}$ verifica che l'input z sia della forma 1^i0x. Se questo non è vero, $\overline{U}(z)$ rigetta, altrimenti ha inizio la fase 2.
Durante la seconda fase, $\overline{U}$ verifica che x sia la codifica di una macchina di Turing (ossia, rispetti le regole descritte nel Paragrafo 5.1 della Dispensa 5) e, se è così, copia x sul nastro N_2 ed inizia la terza fase, altrimenti entra nello stato di rigetto. È possibile verificare che anche la fase 2 richiede un numero di passi proporzionale a $|z|$.
Durante la terza fase, $\overline{U}$ copia lo stato iniziale di T_x su N_3 e lo stato di rigetto di T_x su N_4: poiché tali stati sono memorizzati all'inizio della stringa x, e poiché ciascun elemento dell'insieme degli stati Q_x di T_x è rappresentato con $\lceil \log |Q_x| \rceil$ bit, la fase 3 termina in un numero di passi proporzionale a $\lceil \log |Q_x| \rceil \leq \lceil \log |x| \rceil$.
Durante la fase 4, $\overline{U}$ scrive sul nastro N_5 il valore di z in unario (è possibile verificare che, a questo scopo, sono sufficienti un numero di passi proporzionale a $|z|$).
Durante la fase 5, $\overline{U}$ simula il comportamento della macchina T_C e, in un numero di passi proporzionale a $2^{2|z|}$, scrive il valore $2^{2|z|}$, in unario, sul nastro N_6 posizionando la testina sull'ultimo 1 scritto.
Infine, durante la fase 6 della computazione $\overline{U}(z)$ avviene la simulazione di $2^{2|z|}$ passi della computazione $T_x(z)$ (utilizzando il nastro N_1 come nastro di lavoro). La simulazione avviene come descritto nella Dispensa 2 relativamente alla macchina di Turing Universale con l'ulteriore accorgimento, ogni volta che viene eseguito un passo della computazione $T_x(z)$, di spostare di una posizione a sinistra la testina sul nastro N_6 e, se ciò non è già avvenuto, di terminare la computazione non appena tale testina legge il simbolo $\Box$: in questo modo, garantiamo che $\overline{U}(z)$ simuli non più di $2^{2|z|}$ passi di $T_x(z)$. Lo stato finale con cui termina la computazione $\overline{U}(z)$ dipende sia dall'eventualità che la computazione $T_x(z)$ sia terminata entro $2^{2|z|}$ passi, sia, nel caso in cui ciò sia avvenuto, dallo stato finale della computazione $T_x(z)$: se $T_x(z)$ ha rigettato entro $2^{2|z|}$ passi allora $\overline{U}(z)$ accetta, se $T_x(z)$ ha accettato entro $2^{2|z|}$ passi allora $\overline{U}(z)$ rigetta, altrimenti se $T_x(z)$ non ha terminato entro $2^{2|z|}$ passi allora $\overline{U}(z)$ rigetta. Ricordando quanto descritto nel Paragrafo 2.5 della Dispensa 2, è possibile mostrare che la simulazione di un singolo passo di $T_x(z)$ richiede un numero di passi proporzionale a $|x| \leq |z|$. Quindi, la fase 6 della computazione $\overline{U}(z)$ termina in un numero di passi proporzionale a $|z|2^{2|z|}$.

In definitiva, $\overline{U}$ decide L e ciascuna computazione $\overline{U}(z)$ termina in $\mathbf{O}(|z|2^{2|z|})$ passi. Quindi, $L \in \text{DTIME}[n2^{2n}]$.

Riassumendo, abbiamo definito un linguaggio $L \in \text{DTIME}[n2^{2n}] - \text{DTIME}[2^n]$ e, dunque, abbiamo dimostrato che $\text{DTIME}[2^n] \subset \text{DTIME}[n2^{2n}]$.

Osserviamo ora che, per il Teorema 6.10, $\mathbf{P} \subseteq \text{DTIME}[2^n]$ e quindi, in conclusione, ancora in virtù del Teorema 6.10,

$$\mathbf{P} \subseteq \text{DTIME}[2^n] \subset \text{DTIME}[n2^{2n}] \subseteq \text{DTIME}[2^{3n}] \subseteq \mathbf{EXPTIME}.$$

□

Teorema 6.15: ***PSPACE*** $=$ ***NPSPACE***

Dimostrazione: Sia Σ un alfabeto finito, e sia $L \in \Sigma^*$ un linguaggio in **NPSPACE**, ossia, un linguaggio accettato da una macchina di Turing non deterministica NT ad un nastro in spazio n^k, per qualche costante $k \in \mathbb{N}$ (ossia, k è indipendente da n). Allora, in virtù del Teorema 6.1, esiste una costante $c \in \mathbb{N}$ tali che NT decide L in tempo 2^{cn^k}.

Poiché L è accettato da NT in spazio n^k, allora, per ogni $x \in \Sigma^*$, esiste una computazione determinstica δ di $NT(x)$ che accetta x in spazio $|x|^k$ e dunque, per quanto appena osservato, in tempo $\tau = 2^{cn^k}$. Ricordiamo che una computazione deterministica è una successione di stati globali, e sia $\langle SG_0(x), SG_1(x), \ldots, SG_\tau(x) \rangle$ la successione di stati globali corrispondenti a δ: $SG_0(x)$ è lo stato globale iniziale di δ, $SG_\tau(x)$ è uno stato globale finale di δ e, per ogni i compreso fra 1 e $\tau - 1$, esiste una quintupla nell'insieme delle quituple di NT che permette di passare da $SG_i(x)$ a $SG_{i+1}(x)$.
Rappresentiamo uno stato globale di NT mediante una parola nell'alfabeto $\{0, 1, \Box\} \cup Q_{NT}$, dove Q_{NT} è l'insieme degli stati di NT: se, ad un dato istante, il nastro di NT contiene la parola $y_1 y_2 \ldots y_h$, la macchina si trova nello stato interno $q \in Q_{NT}$ e la testina scandisce l'i-esimo simbolo del nastro (con $1 \leq i \leq h$), allora la parola che rappresenta tale stato è

$$y_1 \ldots y_{i-1} q y_i \ldots y_h$$

Senza perdita di generalità, assumiamo che la macchina NT, prima di entrare nello stato di accettazione, cancelli il contenuto del proprio nastro; in tal modo, esiste un unico *stato globale di accettazione* per NT: quello in cui il nastro contiene soltanto $\Box$, la macchina è nello stato di accettazione q_A e la testina è posizionata sulla cella di indirizzo 0 del nastro. In altre parole, l'unico stato globale di accettazione SG_A di NT è, semplicemente, la parola $q_A\Box$.

Algoritmo `simulaNT`		
	Input:	stringa $x_1x_2\ldots x_n$. Il programma utilizza, inoltre, le costanti seguenti: P che memorizza l'insieme delle quintuple di NT, e il valore k. Infine, x, P, k e la variabile *lunghezza* inizializzata alla linea 3 sono globali, ossia, visibili alla funzione `dimezza` e a tutte le sue invocazioni ricorsive.
	1	$SG_0 \leftarrow q_0x_0x1\ldots x_n$;
	2	$SG_A \leftarrow q_A\Box$;
	3	$lunghezza \leftarrow n^k$;
	4	**if** (`dimezza`(SG_0, SG_A, cn^k) **then** $esito \leftarrow$ accetta;
	5	**else** $esito \leftarrow$ rigetta;
	6	**Output:** $esito$

Tabella 6.4: Algoritmo `simulaNT` che simula, deterministicamente, la computazione della Macchina di Turing non deterministica NT.

Infine, se x è la parola $x_1x_2\ldots x_n \in \Sigma^n$, allora lo stato globale iniziale di $NT(x)$ è $SG_0(x) = q_0x_1x_2\ldots x_n$, dove $q_0 \in Q_{NT}$ è lo stato iniziale di NT.

Definiamo, ora, una macchina di Turing deterministica T che simula il comportamento di NT. Per ogni $x \in \Sigma^*$, $T(x)$ verifica se esiste una computazione deterministica δ di $NT(x)$ che accetta x utilizzando $|x|^k$ celle. Poiché L è accettato da NT in spazio n^k, $x \in L$ se e solo se esiste una computazione deterministica di NT che accetta x in spazio $|x|^k$ e, quindi, è lunga non più di $\tau = 2^{c|x|^k}$ passi; pertanto, T trova δ se e soltanto se $x \in L$. Questo significa che $T(x)$ deve verificare, utilizzando al più $|x|^k$ celle, se esiste una sequenza di al più τ transizioni che permette di passare dallo stato globale iniziale $SG_0(x)$ allo stato globale di accettazione SG_A.
Invece di descrivere T, descriviamo, in Tabella 6.4, il programma `simulaNT` scritto in linguaggio **PascalMinimo** che esegue lo stesso compito: `simulaNT` consiste nell'invocazione della funzione ricorsiva `dimezza`.
La funzione ricorsiva `dimezza` (in Tabella 6.5) utilizza tre parametri, gli stati globali SG_1 e SG_2 e l'intero i, e verifica se esiste una sequenza di al più 2^i transizioni che permette di passare dallo stato globale SG_1 allo stato globale SG_2. La verifica avviene direttamente se $SG_1 = SG_2$ oppure se $i = 0$: in quest'ultimo caso, $2^i = 1$, ovvero, cerchiamo una quintupla in P che porti dallo stato globale SG_0 allo stato globale SG_1. Nel caso, invece, $i > 0$ (e $SG_1 \neq SG_2$) la verifica avviene ricorsivamente, cercando uno stato globale intermedio SG tale che esista una sequenza di (al più) 2^{i-1} transizioni che faccia passare da SG_1 a SG **e** esista una sequenza di (al più) 2^{i-1} transizioni che faccia passare da SG a SG_2.

Analizziamo, ora, la quantità di celle di memoria sufficienti ad eseguire la funzione `dimezza` con input $x = x_1x_2\ldots x_n$. Osserviamo che, ogni volta che una invocazione della funzione `dimezza` esegue la linea 7 e invoca ricorsivamente sé stessa, deve salvare il proprio stato corrente, ossia i due stati globali con cui è stata invocata (SG_1 e SG_2 in Tabella 6.5) e lo stato globale che utilizza per l'nvocazione ricorsiva che sta per lanciare (SG in Tabella 6.5), di dimensione (al più) $n^k + |Q|$ ciascuno, e la variabile *trovata*. Sempre alla linea 7, una volta che l'invocazione `dimezza`$(SG_1, SG, i-1)$ è terminata, lo spazio che essa aveva utilizzato può essere, poi, riutilizzato per l'invocazione `dimezza`$(SG, SG_2, i-1)$. In conclusione, l'invocazione `dimezza`(SG_0, SG_A, cn^k) utilizza

$$\sum_{i=0}^{cn^k} \Big[3(n^k + |Q|) + 1\Big] \;=\; (cn^k + 1)(3n^k + 2|Q| + 1) \in \mathbf{O}(n^{2k})$$

celle di memoria.
Infine, lo spazio richiesto dall'algoritmo `simulaNT` con input $x = x_1x_2\ldots x_n$ è la somma dello spazio richiesto dall'invocazione `dimezza`(SG_0, SG_A, cn^k) e dello spazio richiesto dall'inizializzazione della variabile *lunghezza* alla linea 3: poiché la funzione n^k è space-constructible, tale inizializzazione richiede un numero di celle di memoria in $\mathbf{O}(n^k)$ e, quindi, l'intero algoritmo richiede spazio in $\mathbf{O}(n^{2k})$.

Osserviamo, ora, che l'algoritmo `simulaNT` termina sempre la sua esecuzione, su ogni input $x \in \Sigma^*$. Allora, per il Teorema 6.4 l'esistenza dell'algoritmo `simulaNT` implica l'esistenza di una macchina di Turing deterministica T che decide L e tale che, per ogni $x \in \Sigma^*$, $dspace(T,x) \in \mathbf{O}(|x|^{2k})$. Quindi, $L \in$ **PSPACE**. □

In questo paragrafo abbiamo mostrato un esempio di una coppia di classi fra le quali vale l'inclusione propria, e un

```
    boolean funzione dimezza (stato globale SG1, stato globale SG2, int i)
1   begin
2       trovata ← falso;
            // la sequenza di transizioni da SG1 a SG2 non è stata ancora trovata
3       if (SG1 = SG2) then trovata ← vero;
4       else if (i = 0 ∧ ∃ una quintupla che realizza la transizione da SG1 a SG2) then
            trovata ← vero;
5       else if (i > 0) then
6           for each (stato globale SG di al più lunghezza simboli) do begin
7               trovata ← dimezza(SG1, SG, i − 1) ∧ dimezza(SG, SG2, i − 1) );
8               if (trovata = vero) then break
                // se lo stato globale SG è raggiungibile da SG1 in (al più) 2^(i−1) transizioni
                   e se lo stato globale SG2 è raggiungibile da SG in (al più) 2^(i−1) transizioni,
                   allora il ciclo for viene interrotto
9           end
10      return: trovata;
11  end
```

Tabella 6.5: La funzione ricorsiva che simula la computazione di una fissata Macchina di Turing non deterministica NT.

esempio di di una coppia di classi in l'inclusione si è rivelata essere una uguaglianza. Tali risultati sono, in effetti, estendibili ad altre classi: ad esempio, con tecniche simili a quelle utilizzate nei due teoremi di questo paragrafo, possiamo provare che **NP** $\subset$ **NEXPTIME** e che **PSPACE** $\subset$ **EXPSPACE** (la classe dei linguaggi decidibili in spazio deterministico esponenziale), ed anche che **EXPSPACE** = **NEXPSPACE**. Se osserviamo con attenzione questi risultati, ci accorgiamo che siamo riusciti a confrontare con precisione classi spaziali deterministiche con classi spaziali non deterministiche, classi temporali deterministiche con altre classi classi temporali deterministiche, e classi temporali non deterministiche con altre classi temporali non deterministiche.

Quello che, invece, appare molto più complesso è riuscire a confrontare classi temporali deterministiche con classi temporali non deterministiche: mentre è banale dimostrare che **P** $\subseteq$ **NP** (Teorema 6.1) non è stato sino ad ora possibile né dimostrare che **P** = **NP** né che **P** $\neq$ **NP**. In effetti, la dimostrazione del Teorema 6.8 mostra una macchina deterministica T che simula una macchina non deterministica NT, ma, in tale dimostrazione, T opera in tempo *esponenziale* nel tempo utilizzato da NT. Né sembra evidente come simulare in tempo polinomiale il non determinismo mediante macchine deterministiche. Alla luce di queste considerazioni, la maggior parte degli esperti nel campo della complessità computazionale, è propensa a ritenere che valga la seconda ipotesi, che costituisce la

Congettura fondamentale della teoria della complessità computazionale: P $\neq$ NP.

6.7 Riduzioni polinomiali, completezza, linguaggi separatori

Nel Paragrafo 6.3 abbiamo mostrato alcune relazioni di inclusione fra classi di complessità. Tuttavia, tranne pochissime eccezioni, in quasi nessun caso è stato possibile provare che si trattasse di inclusioni *proprie*, ossia, in quasi nessun caso è stato possibile dimostrare che le due classi coinvolte fossero non coincidenti.

Una possibile tecnica di dimostrazione che due classi di complessità C_1 e C_2 tali che $C_1 \subseteq C_2$ sono distinte consiste nell'individuare un *linguaggio separatore*, ossia, un linguaggio $L \in C_2 - C_1$. Naturalmente, si tratta di un compito tutt'altro che semplice: in particolare, la dimostrazione che $L \notin C_1$ richiede di mostrare che *nessuna* macchina di Turing (deterministica o non deterministica, dipendentemente dal tipo di classe C_1) è in grado di decidere o accettare il liguaggio L utilizzando la quantità di risorse indicata nella definizione di C_1.

È possibile, comunque, individuare i linguaggi *candidati* ad essere separatori fra due classi utilizzando una nozione collegata al concetto di riducibilità funzionale (si veda la Dispensa 5, Paragrafo 5.5), la nozione di linguaggio *completo* per una data classe. Prima di definire tale nozione, osserviamo che il concetto di riducibilità funzionale può essere specializzata richiedendo che la funzione che trasforma parole di un linguaggio in parole di un altro linguaggio soddisfi

qualche predicato π: ad esempio, potremmo richiedere che la parola cui applichiamo la funzione di riduzione e la parola in cui viene trasformata abbiano la stessa lunghezza. Chiamiamo π-riduzione una riduzione funzionale che soddisfa il predicato π, e scriviamo $L_1 \preceq_\pi L_2$ quando un linguaggio L_1 è π-riducibile al linguaggio L_2.

Definizione 6.3: Sia C una classe di complessità di linguaggi e sia $\preceq_\pi$ una generica π-riduzione. Un linguaggio $L \subseteq \Sigma^*$ è *C-completo rispetto alla π-riducibilità* se:

a) $L \in C$ e

b) per ogni altro $L' \in C$, vale che $L' \preceq_\pi L$.

La nozione di completezza di un linguaggio rispetto ad una π-riduzione si rivela utile quando la classe cui si riferisce (o qualche sua sottoclasse) è *chiusa* rispetto alla π-riduzione.

Definizione 6.4: Una classe di complessità C è *chiusa* rispetto ad una generica π-riduzione se, per ogni coppia di linguaggi L_1 ed L_2 tali che $L_1 \preceq_\pi L_2$ e $L_2 \in C$, si ha che $L_1 \in C$.

La chiusura di una classe C rispetto ad una π-riduzione può essere utilizzata per dimostrare l'appartenenza di un linguaggio L a C: segue direttamante dalla definizione che, se sappiamo che una classe di complessità C è chiusa rispetto ad una π-riduzione e che un certo linguaggio L' appartiene a C, allora, se dimostriamo che $L \preceq_\pi L'$, possiamo dedurre che anche L appartiene a C.

D'altro canto, la chiusura di una sottoclasse C' di C rispetto ad una π-riduzione permette di affermare che i linguaggi C-completi rispetto alla π-riduzione sono candidati ad essere i linguaggi separatori fra C e C', come affermato nel prossimo teorema.

Teorema 6.16: *Siano C e C' due classi di complessità tali che $C' \subseteq C$. Se C' è chiusa rispetto ad una π-riduzione allora, per ogni linguaggio L che sia C-completo rispetto a tale π-riduzione, $L \in C'$ se e solo se $C = C'$.*

Dimostrazione: Banalmente, se $C = C'$ allora $L \in C'$.

Viceversa, supponiamo che $L \in C'$. Poiché L è C completo rispetto alla π-riducibilità, allora, per ogni $\overline{L} \in C$, $\overline{L} \preceq_\pi L$. Poiché C' è chiusa rispetto alla π-riduzione, questo implica che, per ogni $\overline{L} \in C$, $\overline{L} \in C'$: quindi, $C = C'$. □

Introduciamo, ora, una particolare π-riduzione in cui il predicato π specifica il costo computazionale del calcolo della funzione:

Definizione 6.5: Siano $L_1 \subseteq \Sigma_1^*$ e $L_2 \subseteq \Sigma_2^*$ due linguaggi; diciamo che L_1 è *polinomialmente riducibile* ad L_2, e scriviamo $L_1 \preceq_p L_2$, se esiste una funzione totale e calcolabile $f : \Sigma_1^* \to \Sigma_2^*$ tale che $f \in \mathbf{FP}$ e

$$\forall x \in \Sigma_1^* \, [\, x \in L_1 \Leftrightarrow f(x) \in L_2 \,].$$

Poiché nel resto di queste dispense faremo sempre riferimento alla riducibilità polinomiale, al fine di semplificare le notazioni da ora in avanti scriveremo semplicemente $L_1 \preceq L_2$ per indicare che L_1 è riducibile polinomialmente ad L_2.

Tutte le classi di complessità introdotte nel Paragrafo 6.5 sono chiuse rispetto alla riducibilità polinomiale. Dimostriamo questa proprietà solo relativamente alla classe **P**, lasciando come semplice esercizio la dimostrazione del Teorema 6.18.

Teorema 6.17: *La classe **P** è chiusa rispetto alla riducibilità polinomiale.*

Dimostrazione: Siano $L_1 \subseteq \Sigma_1^*$ e $L_2 \subseteq \Sigma_2^*$ due linguaggi tali che $L_1 \preceq L_2$ e $L_2 \in \mathbf{P}$. Indichiamo con $f : \Sigma_1 \to \Sigma_2$ la funzione in **FP** che riduce L_1 ad L_2, con T_f la macchina di Turing (di tipo trasduttore) che calcola f in tempo polinomiale, e con T_2 la macchina di Turing deterministica (di tipo riconoscitore) che accetta L_2 in tempo polinomiale. Senza perdita di generalità, assumiamo che T_f e T_2 siano entrambe dotate di un singolo nastro (oltre al nastro di output di T_f).

Poiché T_f e T_2 operano in tempo polinomiale, esistono due costanti $h, k \in \mathbb{N}$ tali che, per ogni $x \in \Sigma_1^*$ e per ogni $y \in \Sigma_2^*$, $dtime(T_f, x) \leq |x|^h$ e $dtime(T_2, y) \leq |y|^k$.

Combiniamo, ora, le due macchine T_f e T_2 per ottenere una nuova macchina di Turing deterministica T_1 che decide L_1. T_1 dispone di due nastri: sul primo nastro, all'inizio della computazione è scritto l'input $x \in \Sigma_2^*$. La macchina

opera come segue: innanzi tutto, $T_1(x)$ simula la computazione $T_f(x)$, leggendo x dal primo nastro e scrivendo $f(x)$ sul secondo nastro; a questo punto, $T_1(x)$ simula la computazione $T_2(f(x))$ sul secondo nastro. Quindi, se $T_2(f(x))$ accetta allora $T_1(x)$ accetta, se $T_2(f(x))$ rigetta allora $T_1(x)$ rigetta ($T_2(f(x))$ non può non terminare perché T_2 decide L_2): in altri termini, $T_1(x)$ accetta se e soltanto se $T_2(f(x))$ accetta, ossia, se e soltanto se $f(x) \in L_2$.

Ricordiamo, ora che f è una riduzione da L_1 ad L_2 e, quindi, $f(x) \in L_2$ se e soltanto se $x \in L_1$. In conclusione, T_1 termina per ogni input e $T_1(x)$ accetta se e soltanto se $x \in L_1$, ossia, T_1 decide L_1.

Resta da mostrare che $T_1(x)$ opera in tempo polinomiale in $|x|$. La simulazione di $T_f(x)$ richiede $dtime(T_f,x) \leq |x|^h$ passi e la simulazione di $T_2(f(x))$ richiede $dtime(T_2,f(x)) \leq |f(x)|^k$ passi: dunque, $dtime(T_1,x) \leq |x|^h + |f(x)|^k$. Resta da capire quanto è grande $|f(x)|$: allo scopo, osserviamo che, poiché $dtime(T_f,x) \leq |x|^h$ e T_f deve almeno scrivere il suo output $f(x)$, allora $|f(x)| \leq |x|^h$ (altrimenti T_f non riuscirebbe a scriverla sul suo nastro di output in $|x|^h$ passi). Quindi,

$$dtime(T_1,x) \leq |x|^h + |f(x)|^k \leq\leq |x|^h + (|x|^h)^k = |x|^h + |x|^{hk}$$

e, poiché h e k sono costanti, questo prova che $L_1 \in \mathbf{P}$. □

Teorema 6.18: *Le classi **NP**, **PSPACE**, **EXPTIME**, **NEXPTIME**sono chiuse rispetto alla riducibilità polinomiale.*

Dimostrazione: La dimostrazione è pressoché identica a quella del teorema 6.17 ed è, pertanto, lasciata per esercizio. □

Il Teorema 6.17 ci fornisce uno strumento utile per individuare possibili linguaggi separatori per le classi **P** e **NP**. Infatti:

Corollario 6.3: *Se **P** ≠ **NP** allora, per ogni linguaggio **NP**-completo L, L ∉ **P**.*

Dimostrazione: Supponiamo che L sia un linguaggio **NP**-completo e che $L \in P$. Poiché L è **NP**-completo, per ogni linguaggio $L' \in \mathbf{NP}$, $L' \preceq L$; ma, se $L \in \mathbf{P}$, poiché **P** è chiusa rispetto a $\preceq$, questo implica che, per ogni $L' \in \mathbf{NP}$, $L' \in \mathbf{P}$. Ossia, **P= NP**, contraddicendo l'ipotesi. □

6.8 La classe coNP

Abbiamo enunciato, al termine del Paragrafo 6.6, la congettura fondamentale della Teoria della Complessità Computazionale. Detta Teoria contiene numerose congetture, la seconda delle quali riguarda la relazione fra **NP** e co**NP**.

Ricordiamo che **NP** è la classe dei linguaggi *accettati* in tempo polinomiale da una macchina di Turing non deterministica; pertanto, co**NP** *è la classe dei linguaggi il cui complemento è accettato in tempo polinomiale da una macchina di Turing non deterministica*, ossia,

$$\text{co}\mathbf{NP} = \{L : L^c \in \mathbf{NP}\}.$$

A questo punto, potrebbe sembrare che, analogamente alla classe **P** che coincide con il suo complemento, le due classi, **NP** e co**NP** coincidano. Per chiarire questa questione, iniziamo con il ricordare la asimmetria delle definizioni di accettazione e di rigetto di una macchina di Turing non deterministica: mentre una parola viene accettata se *esiste* una computazione ddeterministica dalla macchina che termina nello stato di accettazione, una parola viene rigettata se *tutte* le computazioni deterministiche terminano nello stato di rigetto. Cerchiamo, allora, di capire le conseguenze di questa asimmetria.

Sia Σ un alfabeto finito e sia $L \in \Sigma$ un linguaggio deciso da una macchina di Turing non deterministica NT. Mostriamo, ora, che, per decidere L^c non è sufficiente, come nel caso deterministico (si veda il Teorema 6.9), invertire gli stati di accettazione e di rigetto di NT. Senza perdita di generalità, possiamo sempre supporre che, per ogni $y \in \Sigma^*$, una delle computazioni deterministiche di $NT(y)$ termini nello stato dei rigetto: infatti, se così non fosse, potremmo costruire, a partire da NT una macchina NT' identica ad NT il cui insieme delle quintuple si ottiene aggiungendo all'insieme delle quintuple di NT le quintuple seguenti:

$$\langle q_0, a, a, q_R, f\rangle, \qquad \forall a \in \Sigma,$$

ove q_0 è lo stato iniziale di NT. In questo modo, qualunque sia $y \in \Sigma^*$, una delle computazioni di $NT'(y)$ termina nello stato di rigetto q_R. Inoltre, poiché tutte le quintuple di NT sono anche quintuple di NT', per ogni $y \in \Sigma^*$:

- se esiste una computazione deterministica di $NT(y)$ che termina nello stato di accettazione, allora esiste anche una computazione deterministica di $NT'(y)$ che termina nello stato di accettazione;
- se nessuna computazione deterministica di $NT(y)$ termina nello stato di accettazione, allora nessuna computazione deterministica di $NT'(y)$ termina nello stato di accettazione.

Questo prova che, per ogni $y \in \Sigma^*$, $NT'(y)$ accetta se e soltanto se $NT(y)$ accetta.
Quindi, da ora in avanti in questo paragrafo assumeremo sempre che, per ogni $y \in \Sigma^*$, una delle computazioni deterministiche di $NT(y)$ termini nello stato dei rigetto.

Infine, costruiamo, a partire da NT una macchina $\overline{NT}$ identica ad NT tranne che per il fatto che gli stati di accettazione e di rigetto sono invertiti: tutte le quintuple di NT che portano nello stato di accettazione diventano quintuple di $\overline{NT}$ che portano nello stato di rigetto, tutte le quintuple di NT che portano nello stato di rigetto diventano quintuple di $\overline{NT}$ che portano nello stato di accettazione, tutte le altre quintuple rimangono invariate.
Poiché $\overline{NT}$ è una macchina di Turing non deterministica, affinché $\overline{NT}$ accetti una parola in $x \in \Sigma^*$ è sufficiente che una delle sue computazioni deterministiche termini nello stato di accettazione. Sia, allora, $x \in L$ (e, dunque, $x \notin L^c$): poiché $x \in L$, certamente una delle computazioni deterministiche di $NT(x)$ termina nello stato di accettazione. D'altra parte, però, in virtù dell'assunzione fatta sopra, una delle computazioni deterministiche di $NT(x)$ termina nello stato di rigetto e, quindi, la computazione corrispondente $\overline{NT}(x)$ terminerà nello stato di accettazione, inducendo, così, $\overline{NT}$ ad accettare $x \notin L^c$.
Questo prova che il linguaggio accettato da $\overline{NT}$ non è L^c.

Osserviamo che, comunque scelto $L \in \mathbf{NP}$, con $L \subseteq \Sigma^*$, è sempre possibile, data la macchina di Turing non deterministica NT che accetta L, derivare da essa una macchina di Turing deterministica T che accetta L^c: ricordiamo, infatti, il Teorema 2.1 della Dispensa 2 ed il corrispondente algoritmo illustrato nelle Tabelle 3.4 e 3.5 della Dispensa 3 che mostrano come, a partire da una macchina di Turing non deterministica NT, sia possibile costruire una macchina di Turing deterministica T che accetti lo stesso linguaggio. Inoltre, poiché $L \in \mathbf{NP}$, sappiamo che esiste $k \in \mathbb{N}$ tale che, per ogni $x \in L$, $ntime(NT,x) \leq |x|^k$. Allora, possiamo modificare l'algoritmo in Tabella 3.4 come illustrato in Tabella 6.6 in modo che esso simuli soltanto computazioni di $|x|^k$ passi di NT e che prenda decisioni opposte rispetto ad NT circa l'accettazione dell'input, così da decidere L^c:

- se la funzione `simulaRicorsivo` non ha restituito q_A entro gli $|x|^k$ passi a disposizione, ossia nessuna computazione deterministica di NT ha accettato entro i primi $|x|^k$ passi, possiamo concludere che nessuna computazione deterministica accetterà mai x e che, quindi, $x \in L^c$ e possiamo terminare nello stato di accettazione;
- se, invece, la funzione `simulaRicorsivo` ha restituito q_A entro gli $|x|^k$ passi a disposizione, ossia una computazione deterministica di NT ha accettato entro i primi $|x|^k$ passi, possiamo concludere che $x \in L$ e possiamo terminare nello stato di rigetto.

La macchina T che decide L^c corrisponde all'algoritmo in Tabella 6.6 . Questo significa che, per ogni $L \in \mathbf{NP}$, esiste una macchina di Turing deterministica (e, quindi, non deterministica con grado di non determinismo pari ad 1) che accetta L^c.

Se, però guardiamo con attenzione la dimostrazione, ci accorgiamo che, per ogni possibile input x, $T(x)$ deve simulare, nel caso peggiore, *tutte* le computazioni deterministiche di $NT(x)$ lunghe $|x|^k$ passi; quindi, se NT ha grado di non determininismo d, allora possiamo soltanto essere certi del fatto che la computazione $T(x)$ termina in $p(|x|)$ passi con $p(|x|) \in \mathbf{O}(d^{|x|^k})$.

In conclusione, la conoscenza di una macchina di Turing non deterministica che accetta (o decide) un dato linguaggio in tempo polinomiale non sembra, in generale, essere d'aiuto per la costruzione di una macchina di Turing non deterministica che, *in tempo polinomiale*, accetti il linguaggio complemento. Questa osservazione ci porta ad enunciare la

Seconda Congettura della teoria della complessità computazionale: co$\mathbf{NP} \neq \mathbf{NP}$.

Le due congetture che abbiamo incontrato non sono, comunque, totalmente indipendenti, come illustrato nel seguente teorema.

Input:	stringa $x_1 \dots x_n$ memorizzata nell'array N, con $N[i] = x_i$ per $i = 1, \dots, n$. Il programma utilizza anche le seguenti costanti: q_0, q_A, q_R e inoltre gli array Q, Σ, Q_1, S_1, S_2,Q_2, M descritti nel testo,
1	$i \leftarrow 1$;
2	$primaCella \leftarrow 1$;
3	$ultimaCella \leftarrow n$;
4	**while** $(q \neq q_A \ \wedge \ q \neq q_R \ \wedge \ i \leq n^k)$ **do begin**
5	$q \leftarrow$ `simulaRicorsivo`$(q_0, 1, N, i)$;
6	$i \leftarrow i + 1$;
7	**end**
8	**if** $(q \neq q_A)$ **then Output:** q_R;
9	**else Output:** q_A

Tabella 6.6: Algoritmo che simula $|x|^k$ passi della computazione non deterministica $NT(x)$ complementando la decisione circa l'accettazione di x.

Teorema 6.19: *Se coNP $\neq$ NP, allora P $\neq$ NP.*

Dimostrazione: Supponiamo che sia **P** = **NP**; allora, co**P** = co**NP**. In virtù del Corollario 6.2, co**P** = **P**; allora,

$$\text{co}\mathbf{NP} = \text{co}\mathbf{P} = \mathbf{P} = \mathbf{NP}.$$

□

Osserviamo che, però, non è stata dimostrata fino ad ora l'implicazione opposta. Proprio per tale ragione, la proposizione co**NP** $\neq$ **NP** è stata introdotta come seconda congettura della complessità computazionale.

Consideriamo nuovamente la riducibilità polinomiale: anche la classe co**NP** risulta chiusa rispetto ad essa.

Teorema 6.20: *La classe coNP è chiusa rispetto alla riducibilità polinomiale.*

Dimostrazione: La dimostrazione è analoga a quella del Teorema 6.17 ed è lasciata per esercizio. □

A questo punto, possiamo caratterizzare i linguaggi "più difficili" in co**NP**, ossia, i linguaggi co**NP**-completi e mostrare che essi sono i candidati ad essere i linguaggi separatori fra **NP** e co**NP**. Questo è l'obiettivo dei prossimi due teoremi.

Teorema 6.21: *Un linguaggio L è NP-completo se e soltanto se L^c è coNP-completo*

Dimostrazione: Se L è **NP**-completo, allora $L \in$ **NP** e, quindi, $L^c \in$ co**NP**.
Inoltre, in virtù della completezza L per la classe **NP**, per ogni $L' \in$ **NP**, $L' \preceq L$. Questo significa che, per ogni $L' \in$ **NP**, esiste una funzione $f_{L'} : \Sigma' \to \Sigma$ (indicando con Σ' e con Σ gli alfabeti sui quali sono definiti, rispettivamente, L' e L) tale che $f_{L'} \in$ **FP** e, per ogni $x \in \Sigma'$, $x \in L$ se e soltanto se $f_{L'}(x) \in L$.
Ma questo significa che, per ogni $x \in \Sigma'$, $x \notin L$ se e soltanto se $f_{L'}(x) \notin L$, ossia, per ogni $x \in \Sigma'$, $x \in L^c$ se e soltanto se $f_{L'}(x) \in L^c$: quindi, L^c è completo per co**NP**.

La dimostrazione della proposizione inversa è pressoché identica ed è lasciata come esercizio. □

Teorema 6.22: *Se esiste un linguaggio L NP-completo tale che L $\in$ NP $\cap$ coNP, allora NP=coNP.*

Dimostrazione: Se $L \in$ **NP** $\cap$ co**NP**, allora anche $L^c \in$ **NP** $\cap$ co**NP**, e, in particolare, $L^c \in$ **NP**. Inoltre, poiché L è **NP**-completo, dal Teorema 6.21, segue che L^c è co**NP**-completo, e quindi, per ogni $L' \in$ co**NP**, si ha che $L' \preceq L^c$. Ma **NP** è chiusa rispetto alla riducibilità polinomiale (Teorema 6.18) e $L^c \in$ **NP**: allora, per ogni linguaggio $L' \in$ co**NP**, si ha che $L' \in$ **NP**. Quindi, co**NP** $\subseteq$ **NP**.

Mostriamo ora l'inclusione opposta. Poiché L è **NP**-completo allora, per ogni $L'' \in$ **NP** si ha che $L'' \preceq L$; ma $L \in$ **NP** $\cap$ co**NP**, allora, in particolare, $L \in$ co**NP**. Segue allora dalla chiusura di co**NP** rispetto alla riducibilità polinomiale (Teorema 6.20) che $L'' \in$ co**NP**: allora, per ogni linguaggio $L'' \in$ **NP**, si ha che $L'' \in$ co**NP**. Quindi, **NP** $\subseteq$ co**NP**.

Le due inclusioni dimostrano il teorema. □