Lezione 1: problemi, modelli di calcolo, algoritmi

Miriam Di Ianni

3 gennaio 2011

1 Problemi e algoritmi

Cosa è un problema come lo conosciamo: un insieme di dati ed una domanda.

Esempio. La somma delle lunghezze di due segmenti è 18 e la loro differenza 10. Calcolare le lunghezza dei due segmenti.

Per ogni problema, posso trovare metodi di soluzione differenti:

- $(18 + 10)/2 = 14$ e $18 - 14 = 4$
- $(18 - 10)/2 = 4$ e $10 + 4 = 14$

Una volta che ho un metodo di soluzione, posso risolvere tanti problemi *simili*: tutti quelli che utilizzano lo stesso insieme di dati, ma che differiscono per il loro valore.

Esempio. Il problema `SommaEDifferenzaSegmenti` del calcolo delle lunghezze l_1 ed l_2 di due segmenti dati la loro somma s e la loro differenza d.

Definizione 1 *Un problema è definito da un insieme di istanze, per ciascuna delle quali è necessario trovare una soluzione. Una istanza del problema è un insieme di dati. Una soluzione di una istanza è un insieme di valori, correlati ai dati, che rispettano un insieme di vincoli.*

Nell'esempio, una istanza del problema `SommaEDifferenzaSegmenti` è una coppia di valori numerici non negativi s e d; una soluzione è una coppia di valori numerici non negativi l_1 e l_2 tali che $l_1 + l_2 = s$ e $l_1 - l_2 = d$.

Definizione 2 *Un* algoritmo *è la descrizione di una sequenza di* passi elementari *che permettono ad un qualche esecutore di calcolare una soluzione di un problema a partire da sua una istanza.*

Osserviamo che il concetto di algoritmo è strettamente connesso a quello di *esecutore* dello stesso. In questo senso, resta da chiarire cosa intendiamo con *passo elementare*. In generale, con il termine "passo elementare" ci riferiamo ad una operazione il cui significato sia immediatamente comprensibile a chi dovrà eseguire effettivamente l'algoritmo. Ad esempio, se l'esecutore è un bambino di terza elementare il seguente non è un passo elementare di un algoritmo per il problema `SommaEDifferenzaSegmenti`:

calcola la soluzione dell'equazione $x + d + x = s$.

Esempio. Un algoritmo che risolve il problema `SommaEDifferenzaSegmenti` che debba essere eseguito da uno studente di scuola secondaria superiore è il seguente:

dati: somma di due segmenti s e loro differenza d.

1. calcola $x = s + d$;
2. calcola $l_1 = x/2$;
3. calcola $l_2 = s - l_1$;

soluzione: l_1 ed l_2.

Ma anche il seguente è un algoritmo che risolve il problema `SommaEDifferenzaSegmenti`:

dati: somma di due segmenti s e loro differenza d.

1. calcola $y = s - d$;
2. calcola $l_1 = y/2$;
3. calcola $l_2 = d + l_1$;

soluzione: l_1 ed l_2.

Il resto di questa lezione sarà dedicato proprio a chiarire il concetto di *passo elementare*. Prima di procedere, però, riportiamo la nostra attenzione sul concetto che, come abbiamo già osservato, possiamo definire diversi algoritmi per la soluzione di uno stesso problema. Vediamo, ora, un altro importante esempio di questo fatto

Esempio: algoritmi di ordinamento. Si consideri un insieme di n contenitori numerati da 0 a $n-1$, ossia, $c[0], c[1], \ldots, c[n-1]$, in ciascuno dei quali è contenuto un numero intero (per i più esperti, stiamo considerando un array c di interi). Vogliamo scambiare i contenuti dei contenitori in modo tale che, al termine del procedimento, i numeri contenuti nei contenitori siano ordinati in maniera non decrescente, ossia: il contenitore $c[n-1]$ contenga il numero più grande fra tutti quelli contenuti in $c[0], c[1], \ldots, c[n-1]$, il contenitore $c[n-2]$ contenga il secondo numero più grande, e, in generale, per $i = 0, \ldots, n-1$, il contenitore $c[n-i]$ contenga l'i-esimo numero più grande fra tutti quelli contenuti in $c[0], c[1], \ldots, c[n-1]$.

Per risolvere questo compito possiamo utilizzare algoritmi differenti. Nei due algoritmi che prendiamo in considerazione, l'ordinamento degli oggetti avviene in n fasi distinte: in ogni fase è garantito che almeno un numero sia inserito nel contenitore corretto. Più precisamente, durante la fase i è garantito che l'i-esimo numero più grande sia inserito nel contenitore $c[n-i]$. Detto questo, i due algoritmi che stiamo per presentare differiscono per il modo in cui i numeri raggiungono il corretto contenitore.

Il primo algoritmo che prendiamo in considerazione è l'algoritmo *Selection Sort* (ordinamento per selezione): ad ogni fase, viene prima selezionato il numero più grande fra quelli che ancora non sono stati inseriti nel contenitore corretto e, poi, esso viene inserito nel contenitore corretto. Possiamo, quindi, suddividere ciascuna fase in due sotto-fasi:

- durante la prima sotto-fase della fase i viene selezionato l'i-esimo numero più grande, ossia, il contenitore h in cui esso è contenuto, e
- durante la seconda sotto-fase della fase i tale numero viene inserito nel contenitore $n - i$, ossia, il contenuto del contenitore h viene scambiato con il contenuto del contenitore $n - i$.

L'algoritmo *Selection Sort* è descritto formalmente in Tabella 1.

Il secondo algoritmo che prendiamo in considerazione è l'algoritmo *Bubble Sort* (ordinamento a bolle): ad ogni fase, per ogni coppia di contenitori consecutivi, vengono confrontati i loro contenuti e, se non sono nell'ordine corretto, essi vengono scambiati. In questo modo, durante la prima fase il numero più grande niene posizionato nel contenitore $c[n-1]$ e, in generale, durante la fase i l'i-esimo elemento più grande viene posizionato nel contenitore $c[n-i]$. Possiamo, quindi, descrivere la fase i nel modo seguente: per $j = 0, \ldots, n-i-1$, confronta i numeri contenuti in $c[j]$ e in $c[j+1]$ e, se il numero contenuto in $c[j+1]$ è minore del numero contenuto in $c[j]$, scambiali.

L'algoritmo *Bubble Sort* è descritto formalmente in Tabella 2.

Dati:	i contenitori $c[0], c[1], ..., c[n-1]$, contenenti ciascuno un numero intero.
Risultato:	una permutazione dei contenuti dei contenitori, in modo tale che risulti: il numero contenuto nel contenitore $c[i]$ è maggiore o uguale al numero contenuto nel contenitore $c[i-1]$, per ogi $i = 1, \ldots, n-1$.
1	impostiamo $i = 1$;
2	iniziamo la fase i:
3	sotto-fase 1: scandiamo i contenitori $c[0], \ldots c[n-i]$ alla ricerca di quello che contiene il valore massimo e ricordiamo l'indice h di tale contenitore. In dettaglio:
4	impostiamo $h = 0$ e $j = 1$;
5	se il numero contenuto in $c[h]$ è più piccolo di quello contenuto in $c[j]$ poniamo $h = j$;
6	incrementiamo di 1 il valore di j (ossia, poniamo $j = j + 1$);
7	fine sotto-fase 1: ora sappiamo che l'i-esimo numero più grande è contenuto in $c[h]$;
8	sotto-fase 2: scambiamo il contenuto del contenitore $c[h]$ con quello del contenitore $c[n-i]$. Ora sappiamo che l'i-esimo numero più grande è contenuto in $c[n-i]$;
9	la fase i è terminata;
10	incrementiamo di 1 il valore di i (ossia, poni $i = i + 1$) e, se $i < n - 1$, iniziamo una nuova fase tornando al punto 2.

Tabella 1: Algoritmo `A:SelectionSort`.

Dati:	i contenitori $c[0], c[1], ..., c[n-1]$, contenenti ciascuno un numero intero.
Risultato:	una permutazione dei contenuti dei contenitori, in modo tale che risulti: il numero contenuto nel contenitore $c[i]$ è maggiore o uguale al numero contenuto nel contenitore $c[i-1]$, per ogi $i = 1, \ldots, n-1$.
1	impostiamo $i = 1$;
2	iniziamo la fase i:
3	impostiamo $j = 0$;
4	confrontiamo i numeri contenuti in $c[j]$ e in $c[j+1]$:
5	se il numero contenuto in $c[j+1]$ è più piccolo di quello contenuto in $c[j]$ allora li scambiamo di posto;
6	incrementiamo di 1 il valore di j (ossia, poni $j = j + 1$) e, se $j \leq n - i - 1$, torniamo al punto 4;
7	fine fase i: ora sappiamo che l'i-esimo numero più grande è contenuto in $c[n-i]$;
8	incrementiamo di 1 il valore di i (ossia, poniamo $i = i + 1$) e, se $i < n - 1$, iniziamo una nuova fase tornando al punto 2.

Tabella 2: Algoritmo `A:BubbleSort`.

2 Linguaggi e modelli di calcolo

Sino ad ora abbiamo descritto i nostri algoritmi utilizzando il linguaggio naturale e, quindi, assumendo che il loro esecutore fosse un essere umano. Ancora per un esecutore umano, è possibile utilizzare un altro tipo di linguaggio per descrivere gli algoritmi: il *linguaggio dei diagrammi di flusso*. Rispetto al linguaggio naturale, il linguaggio dei diagrammi di flusso mette in rilievo alcune caratteristiche che ciascun algoritmo possiede, e che sono rilevabili anche nei du algoritmi che abbiamo presentato: i passi elementari di cui ciascun algoritmo è costituito possono essere sempre classificati in una delle seguenti due categorie

- impostazioni di valori - come nelle istruzioni 1 e 3 dell'algoritmo in Tabella 2;
- scelte - come nell'istruzione 5 dell'algoritmo in Tabella 2.

Inoltre, mentre si assume che, *normalmente*, le istruzioni debbano essere eseguite una di seguito all'altra, nella sequenza specificata dal loro numero d'ordine, in alcuni casi, *se alcune condizioni si verificano*, è necessario *saltare* alcune istruzioni (ad esempio, se i due numeri contenuti in $c[j]$ e $c[j+1]$ si trovano già nell'ordine corretto all'istruzione 5, lo scambio non avviene) oppure *tornare indietro e ripetere* alcune istruzioni (come nelle istruzioni 6 e 8). In alcuni casi, cioè, è necessario alterare il normale flusso di esecuzione delle istruzioni. Infine, nel linguaggio dei grafi di flusso viene utilizzato il concetto di variabile, che corrisponde, in qualche modo, al concetto di unità di memoria di un calcolatore e a cui ci siamo riferiti fino ad ora con il termine "contenitore".

Il linguaggio dei diagrammi di flusso utilizza due componenti grafiche per distinguere i due tipi di istruzioni descritte precedentemente: le istruzioni che servono ad assegnare valori a variabili sono scritte all'interno di rettangoli, le istruzioni che impongono scelte vengono scritte all'interno di rombi. La sequenza in cui le istruzioni devono essere eseguite viene specificata mediante frecce che collegano le istruzioni stesse. In Figura 1 viene illustrato l'algoritmo Bubble Sort descritto mediante il linguaggio dei grafi di flusso.

Il linguaggio dei diagrammi di flusso è più vicino di quello naturale al modello di calcolatore di Von Neumann. Ricordiamo, brevemente, che tale modello consiste, essenzialmente, delle seguenti componenti

- una memoria ad accesso casuale, ossia, costituita da coppie $\langle cella, indirizzo \rangle$ in cui si può accedere (in lettura o in scrittura) ad una cella specificando il suo indirizzo;
- una unità di elaborazione, in grado di accedere alla memoria, di eseguire le operazioni matematiche e logiche di base e di eseguire i salti da un punto all'altro del programma.

Possiamo allora affermare che, nel liguaggio dei grafi di flusso, il concetto di passo elementare coincide con quello di istruzione della macchina di Von Neumann cui esso si riferisce. A questo proposito, osserviamo che, mentre nel linguaggio naturale lo scambio di due numeri contenuti in due contenitori diversi ($c[j]$ e $c[j+1]$) era considerata una singola istruzione, nel linguaggio dei grafi di flusso esso richiede tre istruzioni distinte e l'utilizzo di una variabile di appoggio t.

Se poi consideriamo come esecutore dei nostri algoritmi l'evoluzione del modello di Von Neumann costituita da un calcolatore in grado di eseguire programmi scritti in un qualche linguaggio di programmazione ad alto livello, il concetto di passo elementare coincide con quello di singola istruzione che tale *modello di calcolo* (ossia, coppia calcolatore-linguaggio) è in grado di eseguire. Dovrebbe, a questo punto, essere chiara la dipendenza del concetto di passo elementare da quello di modello di calcolo.

Questo corso si propone di studiare questioni del tipo

- quanto è efficiente l'algoritmo A nel risolvere il problema P?
- è possibile risolvere P con un algoritmo più efficiente di A?

dove il concetto di efficienza è strettamente correlato al numero di passi elementari eseguiti da un algoritmo. Per questa ragione, è opportuno parlare di efficienza di un algoritmo rispetto ad un modello di calcolo sufficientemente astratto da riuscire a catturare le caratteristiche comuni a tutti i moderni sistemi di calcolo. È in quest'ottica che ci occuperemo ora delle Macchine di Turing.

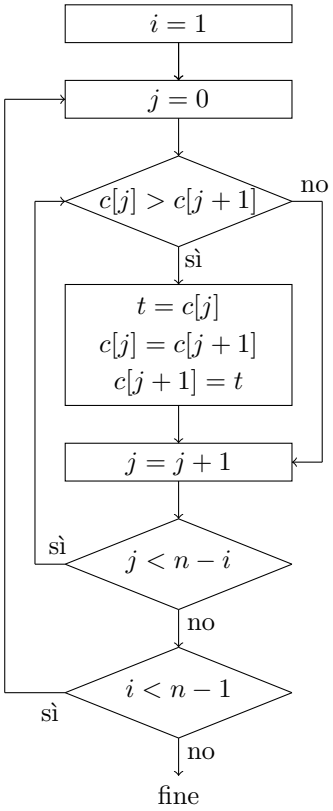

Figura 1: Algoritmo Bubble Sort descritto nel linguaggio dei grafi di flusso.

3 Macchine di Turing

Proviamo a pensare a come un qualunque esecutore eseguirebbe l'algoritmo Bubble Sort per ordinare i numeri contenuti in 6 contenitori. Supponiamo che i numeri siano scritti con il gesso su piccole lavagne contenute nei contenitori e che l'esecutore abbia una *memoria* (molto) limitata: esso è in grado di ricordare un solo numero alla volta. Inizialmente, l'esecutore andrebbe ad esaminare il numero contenuto nel primo contenitore e poi, *ricordando tale numero*, andrebbe ad esaminare il numero contenuto nel secondo contenitore: se tale numero fosse più piccolo di quello che *è nella sua memoria* allora:

- scriverebbe sulla lavagna il numero che si trova nella sua memoria, cancellandone e memorizzando, allo stesso tempo, il numero che vi era scritto in precedenza;
- tornerebbe al primo contenitore scrivendo sulla lavagna il numero che si trova nella sua memoria (cancellando il precedente contenuto);
- si riporterebbe in prossimità del secondo contenitore, leggendone (e memorizzando) il contenuto per poterlo confrontare con il contenuto del terzo contenitore e ripetere il procedimento.

Se, invece, il numero contenuto nel primo contenitore non fosse più grande di quello contenuto nel primo contenitore, allora l'esecutore *memorizzerebbe* il contenuto del secondo contenitore (*dimenticando quello che aveva memorizzato in precedenza*) e passerebbe ad esaminare il terzo contenitore. Una volta esaminato l'ultimo contenitore, il numero più grande avrebbe raggiunto tale ultimo contenitore e l'esecutore dovrebbe ricominciare ad esaminare i contenitori dall'inizio. L'esecuzione dell'algoritmo Bubble Sort su un input di 4 contenitori il cui contenuto è, inizialmente, nell'ordine $6, 2, 8, 1$ è illustrato nelle Figure 2 e 3.

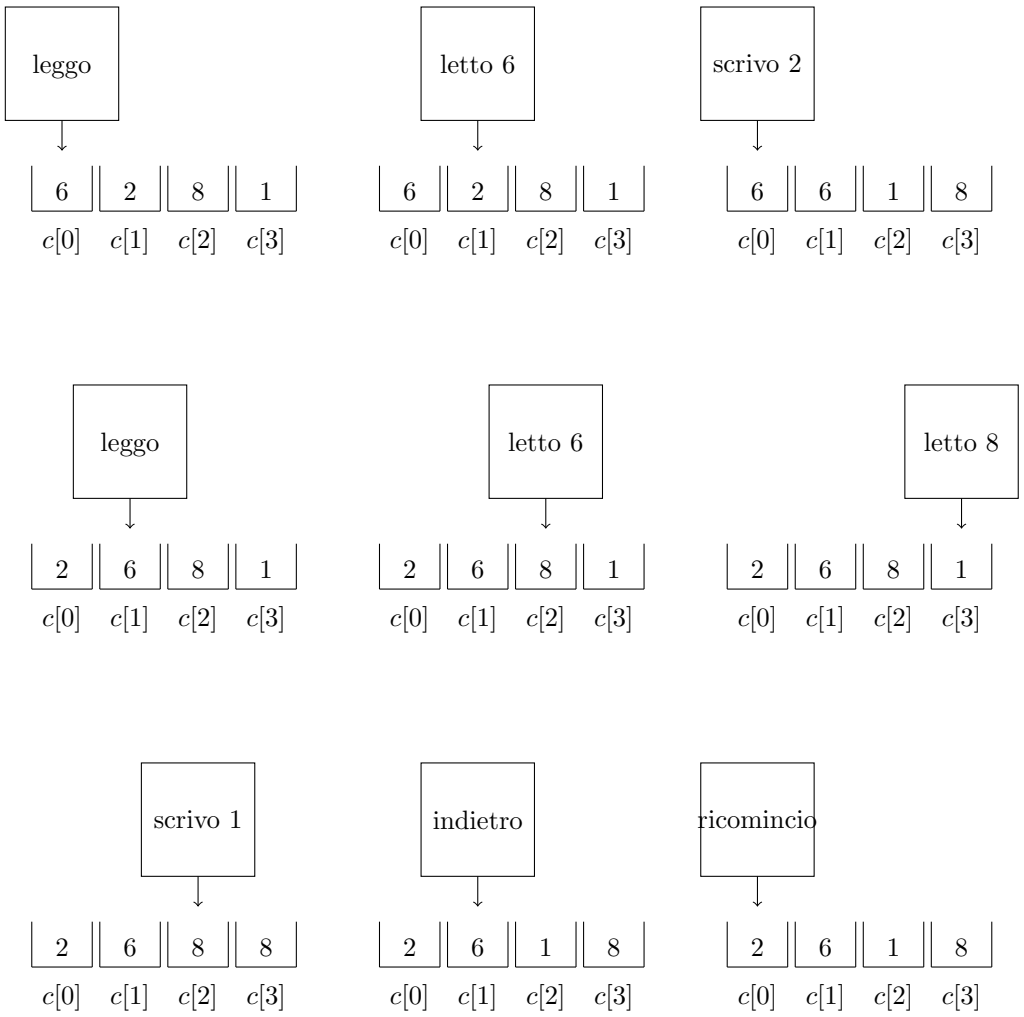

Figura 2: Esecuzione di una iterazione del loop esterno ($i = 0$) dell'algoritmo Bubble Sort con input 4 contenitori contenenti i numeri $6, 2, 8, 1$.

Le caratteristiche dell'esecutore che abbiamo appena descritto corrispondono (con qualche importante eccezione) al modello di calcolo *Macchina di Turing*: un dispositivo di calcolo dotato di una *memoria di lavoro ad accesso sequenziale* (nel nostro esempio, i contenitori), in grado di leggere e scrivere dati da/su tale memoria; ciò che viene scritto sulla memoria di lavoro dipende da ciò che vi era stato letto in precedenza (nel nostro esempio, ciò che l'esecutore ricordava). Abbiamo già osservato come il nostro esecutore avesse

una memoria (molto) limitata: anche la memoria di controllo della macchina di Turing, ossia la quantità di informazioni che gli occorrono per decidere quali azioni intraprendere, è limitata: come chiariremo fra breve, essa ha dimensione costante. Nel seguito definiremo più formalmente la macchina di Turing.

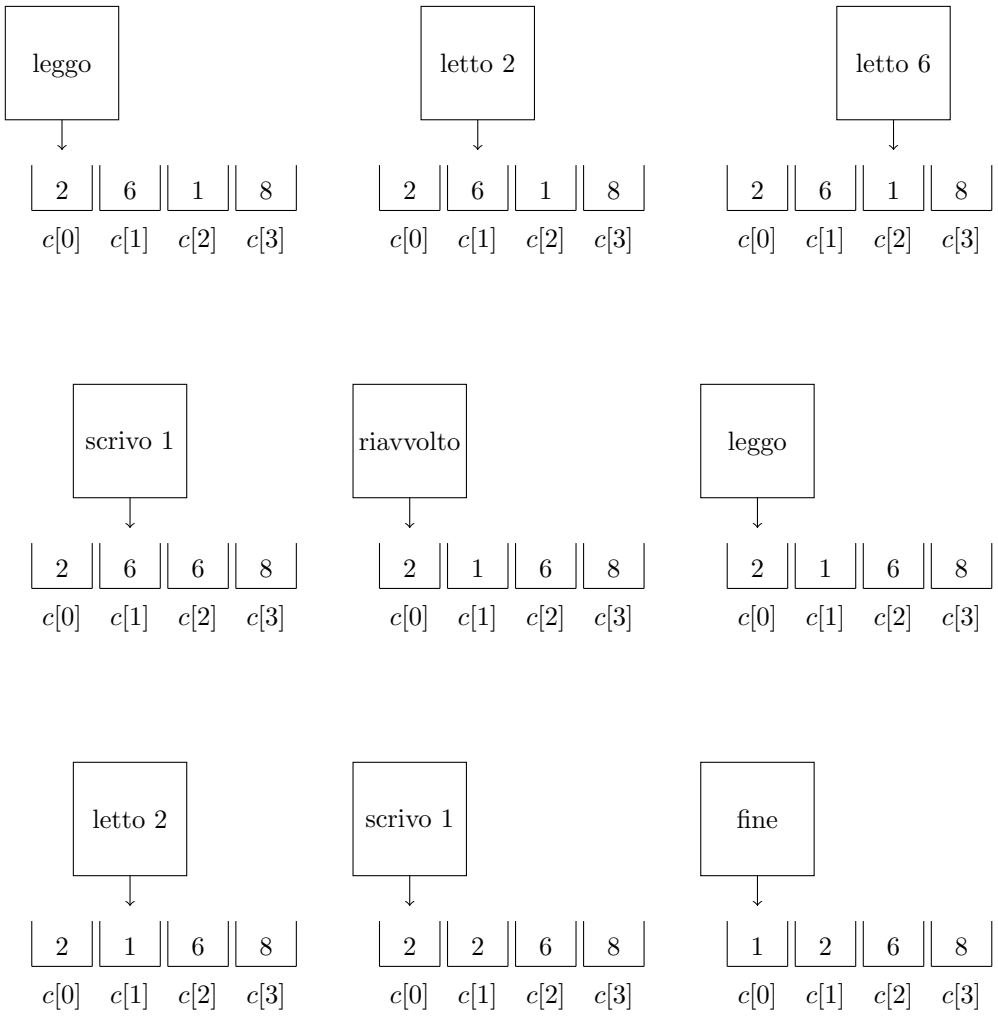

Figura 3: Continuazione dell'esecuzione dell'algoritmo Bubble Sort con input 4 contenitori contenenti i numeri $6, 2, 8, 1$.

Definizione 3 *Sia A un* alfabeto *finito e S un insieme di* stati *nel quale distinguiamo uno* stato iniziale *s_0 ed un insieme di* stati finali *S_F. Una* Macchina di Turing *sull'alfabeto A e sull'insieme di stati S è un dispositivo di calcolo dotato di*

- *una unità di controllo che, ad ogni istante della computazione, può trovarsi in uno qualsiasi degli stati in S,*

- *di un nastro di lettura/scrittura, di lunghezza infinita, suddiviso in celle indicizzate contenenti, ciascuna, un simbolo in A oppure il carattere $\square$ (cella vuota), e alle quali si può accedere sequenzialmente,*
- *di una testina di lettura/scrittura che, ad ogni istante della computazione, è posizionata su una cella del nastro.*

Ciascuna macchina di Turing sull'alfabeto A e sull'insieme di stati S è poi caratterizzata da un insieme Q che ne specifica il comportamento. Ogni elemento di Q è una quintupla del tipo

$$\langle s_1, a_1, a_2, m, s_2 \rangle$$

in cui $q_1 \in S - S_F$, $q_2 \in S$, $a_1, a_2 \in A$, e $m \in \{sinistra, destra, ferma\}$, e che ha il sigificato seguente: se la testina legge nella cella sulla quale si trova il simbolo a_1 e l'unità di controllo si trova nello stato s_1, allora la testina deve scrivere il simbolo a_2 nella cella sulla quale si trova, poi deve spostarsi di una cella nella direzione specificata da m; infine, lo stato interno dell'unità di controllo diventa s_2.

Per completare la definizione di una macchina di Turing restano da specificare le condizioni iniziali (ossia, le condizioni in cui si trova la macchina prima che la computazione abbia inizio) e le condizioni finali (ossia, le modalità di terminazione di una computazione).
Inizialmente, l'input è scritto sul nastro, un carattere per cella, a partire dalla cella di indice 0 e tutte le celle non occupate dall'input contengono il carattere $\square$; la testina è posizionata sulla cella di indice 0 e l'unità di controllo si trova nello stato s_0.
La computazione termina quando l'unità di controllo entra in uno stato $s \in S_F$.

Esempio. Ricordiamo che una parola è *palindroma* se rimane invariata leggendola da sinistra a destra o viceversa: ad esempio, le parole onorarono e ingegno sono palindrome.
Sia $A = \{a, b\}$. Vogliamo definire una macchina di Turing che *decida* se una parola data in input è palindroma. Tale decisione verrà codificata negli stati finali: vogliamo, cioè, che la nostra macchina di Turing termini la computazione nello stato finale s_{pal} se la parola in input è palindroma, termini nello stato $s_{non-pal}$ se la parola in input non è palindroma. Pertanto, definiamo $S_F = \{s_{pal}, s_{non-pal}\}$.
La nostra macchina opera nel modo seguente: a partire dallo stato iniziale, legge il primo carattere della parola, lo cancella (sovrascrivendolo con un $\square$) e, *ricordandolo* (ossia, entrando in s_a se ha letto a, in s_b se ha letto s_b), sposta la testina a destra fino a raggiungere l'ultimo carattere della parola (ossia, raggiunge il primo $\square$ a destra e torna indietro di una posizione). Se l'ultimo carattere è uguale a quello che sta ricordando (ossia, se è nello stato s_a e legge a oppure se è nello stato s_b e legge b) allora torna sul primo carattere a sinistra che non ha ancora cancellato e ricomincia il procedimento, altrimenti conclude che la parola non è palindroma. Se riesce a cancellare tutte le coppie di caratteri della parola, allora può concludere che la parola è palindroma. In dettaglio, le quintuple che realizzano questo compito sono le seguenti:

$$\begin{array}{rr}
\langle s_0, a, \square, s_a, destra \rangle & \langle s_0, b, \square, s_b, destra \rangle \\
\langle s_a, a, a, s_a, destra \rangle & \langle s_b, a, a, s_b, destra \rangle \\
\langle s_a, b, b, s_a, destra \rangle & \langle s_b, b, b, s_b, destra \rangle \\
\langle s_a, \square, \square, s_a^1, sinistra \rangle & \langle s_b, \square, \square, s_b^1, sinistra \rangle \\
\langle s_a^1, a, \square, s_{indietro}, sinistra \rangle & \langle s_b^1, b, \square, s_{indietro}, sinistra \rangle \\
\langle s_a^1, b, b, s_{non-pal}, ferma \rangle & \langle s_b^1, a, a, s_{non-pal}, ferma \rangle \\
\langle s_a^1, \square, \square, s_{non-pal}, ferma \rangle & \langle s_b^1, \square, \square, s_{non-pal}, ferma \rangle \\
\langle s_{indietro}, a, a, s_{indietro}, sinistra \rangle & \langle s_{indietro}, b, b, s_{indietro}, sinistra \\
\multicolumn{2}{c}{\langle s_{indietro}, \square, \square, s_0, destra \rangle} \\
\multicolumn{2}{c}{\langle s_0, \square, \square, s_{pal}, ferma \rangle.}
\end{array}$$

Attività proposte. Progettare macchine di Turing in grado di:
riconoscere parole palindrome su un alfabeto di più di due caratteri, calcolare il numero pari successivo ad uno dato rappresentato in binario, calcolare il bit di parità di un numero dato (0 se il numero è pari, 1 se il numero è dispari) rappresentato in binario.
La computazione descritta nelle Figure 2 e 3 non corrisponde ad una macchina di Turing: perché?

4 Equivalenza linguaggi di programmazione e macchine di Turing e Tesi di Church

Nel precedente paragrafo abbiamo parlato di calcolabilità di funzioni e di decidibilità di linguaggi sempre in relazione alle macchine di Turing. Potremmo, a questo punto, domandarci se, utilizzando modelli di calcolo più potenti non possa essere possibile calcolare anche funzioni non calcolabili dalla semplice macchina di Turing. In effetti, nella teoria della calcolabilità è stata proposta un'ampia varietà di modelli di calcolo. A titolo di esempio, ricordiamo

- le grammatiche di tipo 0
- il modello di Kleene basato sulle equazioni funzionali,
- il λ-calcolo di Church
- la logica combinatoria
- gli algoritmi normali di Markov
- i sistemi combinatori di Post
- le macchine a registri elementari
- i comuni linguaggi di programmazione, sia imperativi che funzionali.

Per quanto riguarda il potere computazionale di *tutti* i modelli di calcolo elencati sopra, è stato dimostrato che essi sono Turing equivalenti, ossia, qualunque funzione calcolabile mediante uno di tali modelli è calcolabile anche mediante una macchina di Turing.

Questa osservazione ha condotto i due logici Alonzo Church ed Alan Turing ad enunciare la tesi seguente: se un problema può essere ridotto in una serie finita di passi elementari, allora esisterà una macchina di Turing in grado di risolverlo. O, in altri termini,

è calcolabile tutto (e solo) ciò che può essere calcolato da una macchina di Turing.

La tesi di Church afferma che non esiste un formalismo più potente della macchina di Turing in termini computazionali. Quindi, tutto ciò che non è calcolabile da una macchina di Turing non può essere calcolato da qualche altro formalismo. La tesi di Church-Turing è ormai universalmente accettata, ma non può essere dimostrata.

Osserviamo che, comunque, sono stati definiti esempi di modelli di calcolo meno potenti delle macchine di Turing. Ricordiamo, fra essi, le grammatiche regolari, gli automi a stati finiti e le macchine che terminano sempre.

Nel seguito di questo corso ci riferiremo più frequentemente a programmi scritti in uno specifico linguaggio che non a macchine di Turing. In particolare, considereremo il linguaggio di programmazione che utilizza, oltre al concetto di variabile e di collezioni di variabili (che, nel seguito, denoteremo sempre con il nome generico di array), le istruzioni seguenti:

- l'istruzione di assegnazione "$x \leftarrow y$;" ove y può essere una variabile, un valore costante, oppure una espressione con operatori aritmetici contenente variabili e valori costanti;
- l'istruzione condizionale "**if** (condizione) **then begin** ⟨sequenza di istruzioni⟩ **end else begin** ⟨sequenza di istruzioni⟩ **end**" in cui la parte **else** può essere assente;
- l'istruzione di loop "**while** (condizione) **then begin** ⟨sequenza di istruzioni⟩ **end**";
- l'istruzione di output, che deve essere l'ultima istruzione del programma e che consente di comunicare all'esterno il valore di una variabile oppure un valore costante: "**Output:** ⟨nomeDiVariabile⟩", oppure, **Output:** ⟨valoreCostante ⟩.

Input:	coppia di interi memorizzati nelle variabili n e m
1	$k \leftarrow 2$;
2	**if** $(n > m)$ **then**
3	$\quad p \leftarrow n$;
4	**else**
5	$\quad p \leftarrow m$;
6	**while** $(p > 2)$ **do begin**
7	$\quad p \leftarrow p - k$;
8	**end**
9	**Output:** p

Tabella 3: Algoritmo che calcola se il massimo fra due interi è pari o dispari.

Assumiamo anche che, se le sequenze di istruzioni sono istruzioni singole, si possa omettere il **begin-end**. Assumiamo, infine, che l'input venga comunicato al programma prima che le istruzioni del programma abbiano inizio mediante una sorta di istruzione denotata dalla parola "**Input:**".

Nel seguito denotermo il linguaggio appena descritto con il termine **PascalMinimo**. Per poterlo usare in sostituzione delle macchine di Turing, dobbiamo dimostrare che il potere computazionale del linguaggio **PascalMinimo** non differisce da quello delle macchine di Turing.

Iniziamo dimostrando (o meglio, accennando alla dimostrazione) che la tesi di Church-Turing rimane valida per il nostro linguaggio.

Teorema 1 *Per ogni programma scritto in accordo con il linguaggio di programmazione* **PascalMinimo**, *esiste una macchina di Turing T di tipo trasduttore che scrive sul nastro di output lo stesso valore fornito in output dal programma.*

Schema della dimostrazione: Sia $\mathcal{P}$ un programma scritto in accordo con le regole del linguaggio di programmazione **PascalMinimo**. Allora, la macchina di Turing T è definita in accordo a quanto di seguito descritto.

- T utilizza, oltre ai nastri input ed output, un nastro per ciascuna variabile e per ciascun valore costante che compare in una condizione. Ad esempio, se in $\mathcal{P}$ compare l'istruzione **if** $(a = 2)$ **then** $b = 1$, allora T utilizza una nastro per la variabile a, un nastro per la variabile b ed un nastro per il valore costante b. I contenuti dei nastri corrispondenti a valori costanti non vengono mai modificati durante le computazioni di T.
- I contenuti dei nastri sono codificati in binario.
- Ad ogni istruzione di assegnazione in $\mathcal{P}$ corrisponde uno stato q_i, $i > 0$, in T.
- Ad ogni istruzione condizionale in $\mathcal{P}$ corrisponde uno stato q_i oppure una coppia di stati q_i, q_j, $i, j > 0$, in T, rispettivamente, nel caso in cui sia assente oppure presente la parte **else**.
- Ad ogni istruzione di loop in $\mathcal{P}$ corrisponde uno stato q_i, $i > 0$, in T.
- Lo stato iniziale di T è q_0.

Illustriamo, ora, come costruire una quintupla a partire da una istruzione in $\mathcal{P}$. A questo scopo, per semplicità, assumiamo di scrivere una singola istruzione in ciascuna linea di codice, come nel programma esempio in Tabella 3, in modo tale da avere una corrispondenza fra linee di codice e stati della macchina (ovviamente, alle linee contenenti **end** non corrisponde alcuno stato in T).

1. Ad ogni assegnazione di un valore costante ad una variabile (linea 1 in Tabella 3) corrisponde una sequenza di scritture sul nastro corrispondente alla variabile assegnanda; tale sequenza è indipendente

da ciò che viene letto sui vari nastri. La sequenza termina con la macchina che entra nello stato corrispondente all'istruzione eseguita (nell'esempio, prima di eseguire l'istruzione alla linea 1 la macchina si trova nello stato q_0 e dopo averla eseguita entra nello stato q_1).

2. Ad ogni assegnazione di un valore variabile ad una variabile (linee 3 e 5 in Tabella 3) corrisponde una copia del nastro corrispondente alla variabile che compare a destra dell'assegnazione sul nastro corrispondente alla variabile che compare a sinistra dell'assegnazione; tale sequenza è indipendente da ciò che viene letto sugli altri nastri. La sequenza termina con la macchina che entra nello stato corrispondente all'istruzione eseguita (nell'esempio, prima di eseguire l'istruzione alla linea 3 la macchina si trova nello stato q_2 e dopo averla eseguita entra nello stato q_3).

3. Ad ogni assegnazione di una espressione ad una variabile (linea 7 in Tabella 3) corrisponde una sequenza di quintuple che eseguono quella espressione (analoghe alle quintuple che eseguono la somma di due valori descritte nella Dispensa 1) e che terminano con una scrittura sul nastro corrispondente alla variabile assegnanda del valore calcolato.La sequenza termina con la macchina che entra nello stato corrispondente all'istruzione eseguita (nell'esempio, prima di eseguire l'istruzione alla linea 7 la macchina si trova nello stato q_6 e dopo averla eseguita entra nello stato q_7).

4. Ogni condizione viene valutata confrontando i contenuti dei due nastri interessati: ad esempio, nella linea 2 in Tabella 3, vengono confrontati i contenuti dei nastri corrispondenti alle variabili n ed m, mentre nella linea 6 vengono confrontati i contenuti dei nastri corrispondenti alla variabile p ed al valore 2. Dopo aver valutato la condizione, la macchina entra in un nuovo stato che dipende, oltre che dal valore della condizione, dal tipo di istruzione in cui la condizione è utilizzata.

   (a) In una istruzione **if-then-else**, se la condizione è vera allora la macchina entra in uno stato che permette di eseguire le istruzioni della parte **if**, se la condizione è falsa allora la macchina entra in uno stato che permette di eseguire le istruzioni della parte **else**. Ad esempio, possiamo trascrivere le linee 2-5 del codice in Tabella 3 in un insieme di pseudo-quintuple, nelle quali viene mostrato il solo contenuto dei nastri di interesse in quelle istruzioni, il confronto $n > m$ non viene trascritto in una sequenza di contolli di coppie di caratteri binari sui nastri n ed m e non viene specificato il movimento delle testine, nella maniera seguente:

$$\langle q_1, n < m, (\ldots, n, \ldots, m, \ldots), q_2, \cdot \rangle$$

$$\langle q_1, n \geq m, (\ldots, n, \ldots, m, \ldots), q_4, \cdot \rangle$$

$$\langle q_2, p \leftarrow n, (\ldots), q_3, \cdot \rangle$$

$$\langle q_4, p \leftarrow m, (\ldots), q_5, \cdot \rangle.$$

   (b) In una istruzione **while**, se la condizione è vera allora la macchina entra in uno stato che permette di eseguire la prima istruzione del corpo del loop, altrimenti entra in uno stato che permette di eseguire la prima istruzione successiva a quelle che costituiscono il corpo del loop. Di nuovo, a titolo di esempio, trascriviamo in pseudo-quintuple le linee 6-8 del codice in Tabella 3:

$$\langle q_5, p > 2, (\ldots, p, \ldots, 2, \ldots), q_6, \cdot \rangle$$

$$\langle q_5, p \leq 2, (\ldots, p, \ldots, 2, \ldots), q_8, \cdot \rangle$$

$$\langle q_6, p \leftarrow p - k, (\ldots), q_7, \cdot \rangle.$$

5. Resta da chiarire come vengano collegate le quintuple le une con le altre. Da quanto sopra esposto, si intuisce che *lo stato con il quale la macchina termina una istruzione è lo stato che le consente di iniziare l'istruzione successiva*: sempre riferendoci all'esempio in Tabella 3 ed alle pseudo-quintuple descritte sino ad ora, osserviamo che lo stato q_1, con il quale termina l'istruzione alla linea 1, è lo stato con il quale inizia l'istruzione **if** alla linea 2. Questa regola è valida con due sole eccezioni:

   (a) lo stato con il quale la macchina termina l'esecuzione dell'ultima istruzione della sequenza che costituisce un blocco **if** deve passare il controllo alla prima istruzione successiva al blocco **else**: nell'esempio, a partire dallo stato q_3 dovrà essere possibile eseguire l'istruzione **while**, e solo essa (dunque, lo stato q_3 è equivalente allo stato q_5);

(b) lo stato con il quale la macchina termina l'esecuzione dell'ultima istruzione della sequenza che costituisce il corpo di un **while** deve passare il controllo all'istruzione che testa la condizione del loop: nell'esempio, a partire dallo stato q_7 dovrà essere possibile testare la condizione alla linea 6 per decidere se ripetere nuovamente il corpo del loop oppure uscire dal loop; dunque, lo stato q_7 è equivalente allo stato q_5.

Non possiamo, in questa sede, formalizzare maggiormente la codifica della macchina di Turing (ad esempio, esplicitandone le quintuple) perché essa dipende dal numero di nastri utilizzato. Per tale motivo, non procediamo neppure oltre con la prova formale della correttezza della trasformazione. Lasciamo come (utile) esercizio la progettazione delle quintuple corrispondenti all'algoritmo in Tabella 3. □

Il Teorema 1 mostra che le funzioni che possono essere calcolate da un programma scritto nel linguaggio **PascalMinimo** possono essere calcolate anche da una macchina di Turing, ossia, che il linguaggio **PascalMinimo** non è un sistema di calcolo più potente delle macchine di Turing. Il prossimo teorema prova la proposizione inversa, ossia, che le macchine di Turing non sono un sistema di calcolo più potente del linguaggio **PascalMinimo**.

Teorema 2 *Per ogni macchina di Turing deterministica T di tipo riconoscitore ad un nastro esiste un programma $\mathcal{P}$ scritto in accordo alle regole del linguaggio* **PascalMinimo** *tale che, per ogni stringa x, se $T(x)$ termina in uno stato finale q_F (di accettazione o di rigetto) allora $\mathcal{P}$ con input x restituisce q_f in output.*

Schema della dimostrazione: Siano $\langle q_{1_1}, s_{1_1}, s_{1_2}, q_{1_2}, m_1\rangle$, $\langle q_{2_1}, s_{2_1}, s_{2_2}, q_{2_2}, m_2\rangle$, ..., $\langle q_{k_1}, s_{k_1}, s_{k_2}, q_{k_2}, m_k\rangle$, le quintuple che descrivono la macchina T ed indichiamo con Q_F l'insieme degli stati finali di T.

Nel programma $\mathcal{P}$ vengono utilizzate le seguenti variabili:

- q, che descrive lo stato attuale della macchina;
- s, un array che descrive il contenuto del nastro ad ogni istante della computazione; la dimensione di s è pari alla porzione di nastro utilizzata dalla
- t, che è la posizione della testina sul nastro di T;
- *primaCella*, che, ad ogni istante, memorizza l'indirizzo della cella del nastro più a sinistra utilizzata fino ad allora dalla computazione $T(x)$;
- *ultimaCella*, che, ad ogni istante, memorizza l'indirizzo della cella del nastro più a destra utilizzata fino ad allora dalla computazione $T(x)$;
- i e j sono variabili di iterazione;
- *trovata* è una variabile booleana utilizzata per testimoniare dell'avvenuto ritrovamento della corretta quintupla da eseguire.

Il programma $\mathcal{P}$ è mostrato in Tabella 4. Si noti che esso utilizza gli array $\overline{q_1} = (q_{1_1}, q_{1_2}, \ldots, q_{1_k})$, $\overline{s_1} = (s_{1_1}, s_{1_2}, \ldots, s_{1_k})$, $\overline{s_2} = (s_{2_1}, s_{2_2}, \ldots, s_{2_k})$, $\overline{q_2} = (q_{2_1}, q_{2_2}, \ldots, q_{2_k})$ e $\overline{m} = (m_1, m_2, \ldots, m_k)$ i cui elementi denotano, rispettivamente, lo stato di partenza, il simbolo letto, il simbolo scritto, lo stato di arrivo e il movimento della testina di ciascuna delle k quintuple di T: tali array, comunque, non sono elencati fra le variabili in quanto sono, per così dire, cablati nel programma, ossia, sono valori costanti, in quanto il programma $\mathcal{P}$ simula le computazioni della sola macchina T.

Intuitivamente, il programma in Tabella 4 cosiste in un unico loop (linee 10-35) nel quale, fissati lo stato attuale q e la posizione t della testina sul nastro (e, conseguentemente, il simbolo letto $s[t]$), cerca la quintupla che inizia con la coppia $(q, s[t])$ e, una volta trovata (se esiste), la esegue. Nell'eseguire la quintupla è possibile che la testina venga spostata a sinistra o a destra della porzione di nastro utilizzata fino a quel momento (questo accade, rispettivamente, quando $t < primaCella$ e quando $t > ultimaCella$): in tal caso,

viene aggiornato il valore di *primaCella* o di *ultimaCella* e viene inizializzato a $\square$ il valore di $s[t]$. Si osservi che non è detto che la quintupla che inizia con $(q, s[t])$ esista: in tal caso, T terminerebbe in nello stato q mentre, se $q \notin Q_F$, l'esecuzione di $\mathcal{P}$ con input x non avrebbe termine.

Seppure intuitivo, non dimostriamo formalmente, in questa sede, che se $T(x)$ termina in uno stato finale allora l'esecuzione di $\mathcal{P}$ con input x restituisce in output lo stato finale in cui $T(x)$ ha terminato. $\square$

I due Teoremi 1 e 2 ci consentono di utilizzare, ai fini del nostro studio, il formalismo del linguaggio **PascalMinimo** al posto di quello, meno immediato, delle macchine di Turing.

Osserviamo, infine, che è possibile dimostrare che l'espressività del linguaggio **PascalMinimo** non cambia introducendo l'uso di funzioni, ossia di sottoprogrammi che possono essere utilizzati come parti destre nelle istruzioni di assegnazione. Una funzione viene definita mediante una *intestazione*, che ne specifica il nome ed i parametri, ossia, i valori letti da opportune variabili del corpo del programma principale specificate al momento delle sua invocazione, ed un *corpo*, costituito da una sequenza di istruzioni del linguaggio **PascalMinimo**. Il corpo di una funzione termina sempre con l'istruzione "**Output:**" che specifica il valore che deve essere scritto nella variabile che compare a sinistra dell'assegnazione in cui viene invocata. Nel seguito di questo corso, utilizzeremo frequentemente le funzioni insieme con la loro variante in cui non viene restituito alcun valore in output ma che modificano i valori delle variabili che ricevono come parametri: ci riferiremo a tale variante delle funzioni con il termine *procedure*.

```
Input: stringa x_1 x_2 ... x_n
1   q ← q_0;
2   primaCella ← 1;
3   ultimaCella ← n;
4   i ← 1;
5   while (i < n) do begin
6           s[i] ← x_i;
7           i ← i + 1;
8   end
9   t ← 1;
10  while (q ∉ Q_F) do begin
11          j ← 1;
12          trovata ← falso;
13          while (j ≤ k ∧ trovato = falso) do begin
14                  if (q = q_{j_1} ∧ s[t] = s_{j_1}) then begin
15                          s[t] ← s_{j_2};
16                          q ← q_{j_2};
17                          if (m_j = sinistra) then begin
18                                  t ← t − 1;
19                                  if (t < primaCella) then begin
20                                          primaCella ← t;
21                                          s[t] ← □;
22                                  end
23                          end
24                          else if (m_j = destra) then begin
25                                  t ← t + 1;
26                                  if (t > ultimaCella) then begin
27                                          ultimaCella ← t;
28                                          s[t] ← □;
29                                  end
30                          end
31                          trovata ← vero;
32                  end
33                  else j ← j + 1;
34          end
35  end
36  Output: q
```

Tabella 4: Algoritmo corrispondente ad una Macchina di Turing deterministica ad un nastro T.