INTRODUZIONE ALLA TEORIA DELLE MACCHINE CALCOLATRICI

DAVIDE TAMBUCHI

1. Cosa è una macchina calcolatrice

Una *macchina calcolatrice*, realizzata comunemente come un *elaboratore elettronico* è un dispositivo in grado di elaborare dei dati, ricevuti in *ingresso*, per fornire in *uscita* un opportuno risultato, frutto di una *computazione*, o *elaborazione*. Nella sua forma più semplice, un elaboratore può essere rappresentato come in figura 1; in essa è mostrata una macchina con *un solo ingresso* (indicato con la lettera `I`, in inglese *input*), ed *una sola uscita* (indicata con `O`, dall'inglese *output*).

Figura 1. Macchina calcolatrice con un solo ingresso ed una sola uscita

Come esempi di elaboratori, possiamo pensare a:

- Un personal computer,
- Una calcolatrice tascabile,
- Una organo elettronico,
- Un registratore di cassa.

Il più semplice personal computer è ad esempio costituito da una *unità elaboratrice*, da una tastiera (il dispositivo di ingresso) e da un video (il dispositivo di uscita). Mediante questi due dispositivi, è possibile *comunicare* tra l'uomo e la macchina; inviando un *flusso di dati* (in inglese: *stream*) dalla tastiera all'unità elaboratrice, la quale invia i dati elaborati sul *video*. È poi possibile aggiungere altri dispositivi di ingresso/uscita (in inglese input/output, abbreviato `I/O`), come ad esempio mouse, scanner, casse acustiche, eccetera. Nella maggior parte dei casi, i due dispositivi più importanti sono normalmente la tastiera ed il video, che per tali motivo sono detti *dispositivo standard di ingresso* (abbreviato `stdin`) e *dispositivo standard di uscita* (`stdout`). Insieme, questi due dispositivi sono indicati con `stdio` (dispositivi standard di input ed output). Occorre tuttavia sottolineare come in alcune applicazioni, questi due non siano i dispositivi principali; pensiamo ad esempio ad un organo elettronico: il dispositivo standard di ingresso è una tastiera simile a quella di un pianoforte, mentre il dispositivo standard di uscita è costituito dalle casse acustiche (esistono altri dispositivi di ingresso e di uscita, ad esempio l'ingresso microfonico, e l'uscita `MIDI`, utilizzata per il collegamento dello strumento musicale ad un personal computer; questi tuttavia non sono i due dispositivi principali, o standard).

2. Struttura di un personal computer

Analizziamo ora il più popolare elaboratore elettronico, il *personal computer*. La struttura che andiamo ad analizzare, detta *modello a bus*, è comune sia ai personal computer che ai grandi elaboratori utilizzati per la gestione delle reti, che ai computer superveloci utilizzati nei laboratori di ricerca. Essa è rappresentata in figura 2.

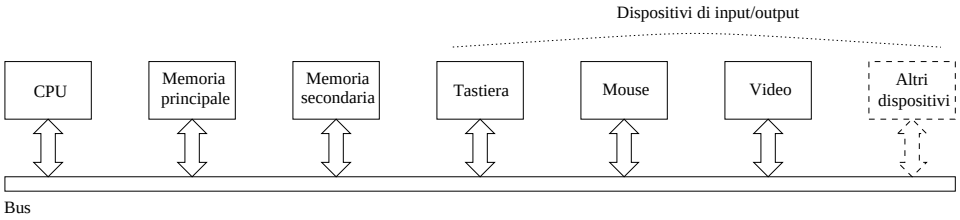


Figura 2. Modello a bus di un elaboratore

Esaminiamo ora questa struttura in dettaglio:

- La *CPU* (dall'inglese *central processing unit*, ovvero *unità centrale di processo*) è l'elemento principale di un computer. Essa ha i compiti di:
  - eseguire i calcoli e le istruzioni fornite dall'utente per mezzo di un *programma*,
  - controllare l'invio dei flussi di dati tra i vari dispositivi,
  - abilitare o disabilitare il funzionamento dei dispositivi collegati al bus.

  Come esempio di CPU, possiamo considerare un qualsiasi *microprocessore*. Occorre tuttavia notare come, nei computer utilizzati per la gestione di reti o per operazioni molto veloci, la CPU sia costituita da *più di un microprocessore*; in questo modo le operazioni da eseguire sono condivise da più di un microprocessore. Si parla in questo caso di un *sistema multiprocessore*.
- Il *bus* è una collezione di *collegamenti elettrici*, o *fili* che permettono di collegare tra di loro tutti i dispositivi rappresentati in figura 2; in particolare la presenza di un bus di collegamento permette di aggiungere facilmente nuovi dispositivi ad un elaboratore (si pensi ad esempio all'introduzione di un banco di memoria, di una scheda di collegamento con uno scanner, al collegamento di un modem). Tutti i dispositivi sono collegati sul bus, e su quest'ultimo transitano i segnali elettrici necessari per le comunicazioni tra i dispositivi stessi. Il transito delle informazioni è controllato dalla CPU, al fine di evitare conflitti. Ad esempio, se si sta inviando un flusso di dati dal mouse alla CPU, non è possibile far transitare simultaneamente altri dati dalla tastiera alla CPU; per tale motivo nel breve istante in cui vi è un flusso di dati proveniente dal mouse, la CPU stessa disabilita temporaneamente il flusso di dati proveniente dalla tastiera.
- La *memoria principale* (o memoria RAM) è un dispositivo elettronico capace di immagazzinare il programma che deve essere eseguito ed i dati usati dal programma (sia quelli provenienti dai dispositivi di input e da inviare ai dispositivi di output, che i dati intermedi necessari per l'elaborazione). I dati sono memorizzati in forma *binaria*, ovvero sotto forma di segnali elettrici che assumono i valori 0 ed 1. I dati contenuti nella memoria principale vengono persi all'atto dello spegnimento del computer. Il termine *memoria RAM* è una abbreviazione per *memoria ad accesso casuale* (Random Access Memory), e ciò significa che i dati possono essere letti e scritti in una qualunque locazione di memoria senza dovere accedere sequenzialmente ad

essi. Ad esempio, un nastro magnetico è una memoria sequenziale; per poter accedere ad un dato presente a metà del nastro occorre prima scorrere la parte iniziale del nastro stesso; mentre un foglio di carta a quadretti è l'equivalente di una memoria ad accesso casuale. Infatti, pensanso a ciascun quadretto come ad una cella (o locazione) di memoria, e pensando di leggere e scrivere dati sul foglio (con una matita, in modo da poterli anche cancellare quando non sono più necessari) sappiamo che possiamo scrivere (o leggere) un dato in un qualsiasi quadretto libero del foglio, senza dover prima analizzare tutti gli altri dati presenti sul foglio stesso. I dati presenti nella memoria RAM vengono persi quando viene tolta alimentazione all'elaboratore.

- La *memoria secondaria* è costituita da un insieme di dispositivi, capaci di memorizzare in modo permanente i dati (anche dopo lo spegnimento del computer). Sono dispositivi di memoria secondaria gli hard-disk, i floppy-disk, CD e DVD, eccetera. Occorre notare come il trasferimento tra la memoria secondaria e la CPU è in genere assai più lento che il trasferimento tra la memoria principale e la CPU; per tale motivo i risultati intermedi dell'elaborazione sono quasi sempre memorizzati nella memoria principale (a meno che la quantità di dati sia così grande da non poter esser contenuta in essa, si pensi ad esempio al problema di ordinamento in ordine alfabetico dell'elenco telefonico della città di Pechino). I dati presenti su memorie di tipo magnetico (nastri, hard-disk, floppy) o su dispositivi ottici (CD, DVC), o su schede memorizzatrici (ad esempio le memorie delle macchine fotografiche digitali) non vengono persi quando si toglie alimentazione al PC. In molti di questi dispositivi è possibile la cancellazione dei dati (esempio, CD riscrivibili, memorie delle macchine fotografiche, nastri magnetici e dischi magnetici), mentre altri dispositivi non possono piú essere cancellati una volta scritti i dati (esempio, CD non riscrivibili).
- I dispositivi di input/output, di cui abbiamo parlato in precedenza.

2.1. **La struttura interna di una CPU.** La figura 3 mostra schematicamente la struttura interna di una CPU; essa è costituita da:

- Una *unità di controllo*, che si occupa di:
  - *Controllare l'accesso al bus* dei dispositivi ad esso collegati (come discusso in precedenza),
  - *Prelevare una istruzione* per volta dal programma contenuto nella memoria principale, *interpretarne* il significato, *prelevare i dati* necessari all'elaborazione dai dispositivi di memoria, ed *eseguirla*. Ad esempio, una singola istruzione potrebbe essere quella di sommare due numeri, oppure sottrarli, oppure cambiare di segno ad un numero, eccetera.
- Una *unità aritmetico-logica* (abbreviata con `ALU`), che si occupa di eseguire le operazioni matematiche (somma, sottrazione, eccetera) e logiche (`OR`, `AND`, eccetera), secondo quanto comandatogli dall'unità di controllo.
- Una collezione di *registri*, ovvero di elemento di memoria con la quale è possibile scambiare dati in modo molto veloce; in essi sono contenuti gli operandi che devono essere elaborati (ad esempio, gli addendi di una somma), il risultato di un operazione appena eseguita, il numero progressivo della prossima istruzione da eseguire (che viene memorizzato in un registro detto *contatore di programma*, abbreviato `PC`), un numero che individua il *tipo di istruzione* da eseguire (una somma, un trasferimento dati, eccetera).

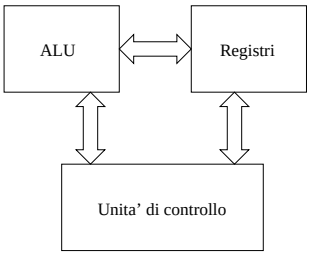


FIGURA 3. Struttura interna di una CPU

3. Cosa è un programma

Per capire cosa è un programma, occorre innanzitutto definire il concetto di *algoritmo*; vedremo successivamente che un *programma* è una *rappresentazione* di un algoritmo in un *linguaggio* interpretabile dalla macchina.

3.1. **Definizione di algoritmo.** Informalmente, un *algoritmo* è un insieme di regole che, elaborando in modo opportuno dei dati iniziali, permettono di ottenere la soluzione di un problema. Queste regole hanno la proprietà di scomporre il problema in una successione di sottoproblemi elementari (detti anche *passi*), che eseguiti uno dopo l'altro consentono di pervenire alla soluzione del problema. Semplificando al massimo, si pensi al problema di preparare una torta; i dati sono gli ingredienti, l'algoritmo è la successione di istruzioni elementari come: distendere la farina sul tavolo, aggiungervi le uova, impastare, eccetera, ed il risultato finale è la torta stessa.

Occorre tuttavia notare che un algoritmo è un concetto matematico; pertanto passiamo a darne una definizione più rigorosa:

Definizione 3.1. Un algoritmo è un insieme di operazioni elementari utilizzate per risolvere un problema ben specificato, avente le seguenti proprietà:

(1) *Finitezza.* Un algoritmo deve terminare dopo un numero finito di passi. (Una procedura che non termina dopo un numero finito di passi non è un algoritmo, ma è detta un *metodo computazionale*).
(2) *Precisione.* Ogni passo deve essere definito con precisione matematica, onde evitare ambiguità di interpretazione.
(3) *Possesso di un ingresso.* L'algoritmo deve avere uno o più dati di ingresso, che devono essere noti prima che l'algoritmo inizi.
(4) *Possesso di un uscita.* Un algoritmo deve fornire una o più uscite, frutto dell'elaborazione, i cui valori dipendono dai valori dei dati in ingresso.
(5) *Effettuabilità.* Tutte le operazioni (passi) di un algoritmo devono poter essere effettuate con precisione, in un arco di tempo ragionevolmente limitato, da una persona che utilizzi solo carta e matita. □

Notiamo come una ricetta di cucina non sia propriamente un algoritmo: ad esempio i passi per la preparazione di un piatto non sono descritti con rigore matematico.

3.2. **Esempio di algoritmo.** Per meglio comprendere la precedente definizione di algoritmo, consideriamo il cosiddetto *algoritmo di Euclide*, utilizzato per trovare il massimo comun divisore tra due numeri interi positivi. Questo algoritmo, risalente a ben 2300 anni fa, è ancora utilizzato nelle moderne tecniche di crittografia utilizzati per comunicare informazioni segrete tra due personal computer, in particolar modo nella famosa tecnica di *crittografia a chiave pubblica* nota con la sigla RSA. Passiamo ora all'analisi dell'algoritmo:

Algoritmo E: (*Algoritmo di Euclide*). Dati due interi positivi m, n, trovare il loro *massimo comun divisore* d.

Passo 1: [Calcolo del resto] Dividere m per n, e calcolarne il resto r.

Passo 2: [Il resto è zero?] Se $r = 0$ vai al passo 5.

Passo 3: [Riduzione] Assegna ad m il valore n (abbreviato: $m \leftarrow n$), assegna ad n il valore r (cioè $n \leftarrow r$).

Passo 4: [Salto] Torna al passo 1.

Passo 5: [Risultato] Il m.c.d. è n. Fine algoritmo. ■

Notiamo che l'algoritmo ha un *nome simbolico*: Algoritmo E, ed un nome per esteso: si tratta dell'*algoritmo di Euclide*. Vi è poi una breve descrizione a parole dell'algoritmo. I passi dell'algoritmo sono tutte operazioni elementari, facilmente eseguibili con carta e penna, come mostreremo fra poco, e ciascun passo è rigorosamente definito. Vi è un ingresso, costituito dai due interi positivi m ed n, e una uscita, il loro massimo comun divisore. L'algoritmo termina, come vedremo tra poco, dopo un numero finito di passi; ciò è dovuto al fatto che, eseguendo un passo dopo l'altro, la successione di valori assunta da m ed n è descrescente, e dopo un numero finito di passi, il numero r deve necessariamente assumere il valore 0 (nel caso limite in cui m ed n sono primi tra loro, ciò accadrà quando n assumerà il valore 1, in questo caso infatti dal passo 1 si ottiene $r = 0$. Pertanto il procedimento di Euclide per il calcolo del m.c.d. soddisfa tutte le condizioni della definizione 3.1 per poter essere considerato un algoritmo. Notiamo infine che il termine della descrizione di un algoritmo è indicata con un rettangolino nero.

Proviamo ora l'algoritmo: supponiamo di voler calcolare il m.c.d. tra $m = 6$ ed $n = 4$. Concludiamo il paragrafo eseguento l'algoritmo passo passo:

Passo 1: Avendosi $m = 6$, $n = 4$, il resto è $r = 2$

Passo 2: Essendo $r \neq 0$, non saltiamo al passo 5, ma continuiamo con il passo 3.

Passo 3: Dopo questo passo, si ha $m = 4$ (il precedente valore di n) ed $n = 2$ (il valore di r calcolato con il passo 1).

Passo 4: Dobbiamo tornare al passo 1.

Passo 1: Ora, con $m = 4$ ed $n = 2$ il resto della divisione di m per n è $r = 0$.

Passo 2: Avendosi $r = 0$, saltiamo al passo 5.

Passo 5: Il massimo comun divisore è $n = 2$. L'algoritmo è terminato.

Il calcolo del m.c.d. è avvenuto in 7 passi. Il numero di passi dipende ovviamente dai valori di m e di n. Il lettore è invitato a provare l'algoritmo, ad esempio prendendo i valori iniziali $m = 40$ ed $n = 11$. In questo caso occorrono 19 passi per trovare il m.c.d. (che vale 1).

3.3. **La rappresentazione di un algoritmo.** L'algoritmo di Euclide è stato scritto in lingua italiana, al fine di essere facilmente interpretato dagli esseri umani. Tuttavia, affinchè esso possa essere interpretato ed eseguito da una macchina, è necessario utilizzare un *linguaggio di programmazione*, che è un linguaggio artificiale composto da un numero ristretto di parole, e che può essere facilmente decodificato ed eseguito da un elaboratore automatico. La scrittura di un algoritmo in un *linguaggio di programmazione* è detta *programma*. Più precisamente, un *programma* è una *rappresentazione* di un *algoritmo*, utilizzante un *linguaggio di programmazione*. Per meglio capire il concetto di rappresentazione, possiamo pensare ad un ritratto fotografico: il volto di una persona (corrispondente ad un algoritmo) è una cosa, la sua rappresentazione sotto forma di immagine fotografica (corrispondente ad un programma) un'altra. La rappresentazione fotografica del volto di una persona è necessaria ogniqualvolta si desidera mettere a disposizioni le informazioni

sulla sua fisiognomia ad altre persone (ad esempio, negli schedari degli uffici di polizia); analogamente la rappresentazione di un algoritmo sotto forma di programma è necessaria affinchè esso possa essere eseguito da un elaboratore elettronico.

Ad esempio, possiamo provare a scrivere l'algoritmo di Euclide in `Pascal`:

```
PROGRAM euclide (input, output);
(* Questo programma calcola il massimo comun divisore *)
(* tra due numeri interi positivi m ed n, utilizzando *)
(* l'algoritmo di Euclide                             *)
   VAR
       n, m, r : integer;
       (* Dichiarazione delle variabili *)
       (* (m, n, r sono numeri interi) *)
   LABEL 1, 5;
       (* Dichiarazione delle etichette *)
   BEGIN
       write ('Introduci il valore di m ');
       (* Scrive sul video la frase tra virgolette *)
       readln (m);
       (* Legge il valore di m da tastiera *)
       write ('Introduci il valore di n ');
       readln (n);
       (* Stessa cosa, per il numero n *)
1:     r := m MOD n;
       (* Passo 1: calcola il resto della divisione di m per n *)
       (* e lo assegna alla variabile r *)
       IF r = 0 THEN GOTO 5;
       (* Passo 2: Se il resto e' uguale a zero salta al punto 5 *)
       m := n; n := r;
       (* Passo 3: m <-- n, n <-- r *)
       GOTO 1;
       (* Passo 4: torna al punto 1 *)
5:     write ('Il m.c.d. vale: ', n);
       (* Passo 5: visualizza la frase tra virgolette *)
       (* ed il massimo comun divisore (il numero n)  *)
   END.
   (* Fine del programma *)
```

Notiamo come il programma abbia un nome (`euclide`), specificato a fianco della *parola riservata* `PROGRAM`. Sempre sulla prima riga, compaiono le parole `input` ed `output` che stanno a significare che si intendono utilizzare il dispositivo standard di ingresso (la tastiera) e quello standard di uscita (il video). Occorre poi dichiarare il nome delle variabili utilizzate (`m`, `n`, `r`), e che esse possono assumere solo valori interi (`integer`). Dopodichè i valori degli ingressi `m` ed `n` vengono letti da tastiera, e successivamente viene calcolato il resto della divisione, utilizzando la parola riservata `MOD`. Se (`IF`) il resto è uguale a zero, il programma salta (mediante l'istruzione `GOTO`) al punto 5 (individuato dall'*etichetta* `5:`), altrimenti prosegue ed assegna il valore di `n` ad `m` e quello di `r` ad `n`. L'operatore di assegnamento utilizzato in Pascal è il simbolo `:=` (nell'algoritmo è indicato con $\leftarrow$), da non confondersi con l'operatore di confronto `=`, utilizzato nell'istruzione precedente per testare se il resto è zero, oppure no. Il passo 4 consiste solo in un salto al passo 1, realizzato ancora con l'istruzione `GOTO`, ed infine il passo 5 consiste nella visualizzazione del risultato e nella terminazione del programma (che avviene raggiunta l'istruzione `END.`). Quanto è racchiuso tra `(*` e `*)` rappresenta un *commento*, utile agli esseri

umani per documentare il programma, ma ignorato dalla macchina. Notiamo come questo programma può essere scritto in modo migliore utilizzando un ciclo indefinito al posto dei salti con le istruzioni `GOTO`; ma qui non ci interessa tanto lo studio delle tecniche di programmazione in un determinato linguaggio quanto mettere in evidenza le analogie tra un algoritmo e la sua rappresentazione.

Consideriamo ora un altro linguaggio di programmazione, il `Fortran77`, e riscriviamo con esso un programma che rappresenta l'algoritmo di Euclide:

```
      PROGRAM euclid
C        Questo programma calcola il massimo comun divisore
C        tra due numeri interi positivi m ed n, utilizzando
C        l'algoritmo di Euclide
      INTEGER n, m, r
C        Dichiarazione delle variabili (m, n, r sono numeri interi)
      WRITE (FMT=*, UNIT=*) ' Introduci il valore di m '
C        Viene scritta su video la frase racchiusa tra
C        le virgolette
      READ (FMT=*, UNIT=*) m
C        Introduzione da tastiera del numero m
      WRITE (FMT=*, UNIT=*) ' Introduci il valore di n '
      READ (FMT=*, UNIT=*) n
C        Stessa cosa, per il numero n
 1    r = MOD(m, n)
C        Punto 1: Calcolo del resto della divisione di m per n
      IF (r .EQ. 0) GO TO 5
C        Punto 2: Se il resto e' uguale a zero salta al punto 5
      m = n
      n = r
C        Punto 3: assegna ad m il valore di n, ed ad n il
C        valore del resto r
      GO TO 1
C        Punto 4: salta al punto 1
 5    WRITE (FMT=*, UNIT=*) ' Il m.c.d. vale ', n
      STOP
C        Punto 5: visualizza la frase tra virgolette, il valore
C        del m.c.d. (cioe' n) e poi termina l'esecuzione
      END
C        Termine della scrittura del programma
```

Come si puó notare, esistono delle specifiche istruzioni (ad esempio `READ` e `WRITE`) per leggere dei dati da tastiera e per scriverle sul video, altre (`MOD`) per calcolare il resto della divisione, altre ancora (`STOP`) per comandare alla macchina di fermarsi. Inoltre esistono degli operatori di assegnamento (il simbolo `=`), di confronto (il simbolo `.EQ.` che significa: "è uguale a", e altri simboli tipici del linguaggio di programmazione scelto (ad esempio il simbolo `*` posto a fianco della parola `UNIT` indica che l'unità di input (o di output) scelta è quella standard. Cioè, lo scrivere `UNIT=*` all'interno delle parentesi poste a fianco di una istruzione `WRITE` significa che il dispositivo su cui scrivere è quello standard (il video); analogamente lo scrivere `UNIT=*` all'interno delle parentesi poste a fianco di una istruzione `READ` significa che il dispositivo di ingresso da cui leggere i dati è quello standard (la tastiera). Le righe che iniziano con la lettera `C` sono delle *righe di commento*. Notiamo altresì come il nome del programma (`euclid`) sia stato troncato, infatti il `FORTRAN77` ammette come lunghezza per i nomi del programma e delle variabili un massimo di 6 caratteri.

Non ci interessa in questo momento soffermarci ad analizzare in dettaglio il significato delle istruzioni e degli operatori del linguaggio `FORTRAN77`, ma solo notare come lo stesso algoritmo possa essere rappresentato (seppur in modo differente) con due differenti linguaggi di programmazione.

Una buona tecnica per imparare a programmare è quella di scrivere prima l'algoritmo (in lingua italiana, o nella propria lingua madre), e successivamente di passare alla scrittura del programma, in un linguaggio scelto a seconda delle esigenze. Infatti, esistono linguaggi particolarmente adatti per il calcolo scientifico, come il `FORTRAN77`, altri adatti per applicazioni in cui è richiesta la massima affidabilità (come l'`Ada95`), altri che si prestano bene ad essere utilizzati per sviluppare programmi gestionali (come il `COBOL`).

4. Approfondimenti

La definizione 3.1 di algoritmo può essere approfondita; infatti dalle cinque proprietà elencate nella definizione stessa, e dalle proprietà di rappresentazione di un algoritmo sotto forma di un programma, possiamo notare che:

- L'algoritmo è di lunghezza finita. Pertanto anche un qualsiasi programma che lo rappresenta deve essere di lunghezza finita.
- Ci deve essere un agente di calcolo capace di eseguire le istruzioni (un uomo con carta e penna per l'algoritmo, un elaboratore per un programma che lo rappresenta).
- L'agente di calcolo ha a disposizione una memoria dove immagazzinare i risultati intermedi (la mente umana ed eventualmente dei fogli di carta per l'uomo, la memoria principale ed eventualmente anche quella secondaria per un elaboratore), in modo da poterli utilizzare nelle fasi successive dell'elaborazione.
- Il calcolo deve avvenire per passi discreti, da eseguirsi secondo una successione rigorosamente descritta nell'algoritmo.
- Il calcolo è deterministico (il risultato del calcolo non è probabilistico, come lo è ad esempio il risultato del lancio di una moneta).
- Non ci deve essere alcun limite fisico nella lunghezza dei dati di ingresso di un algoritmo (l'algoritmo deve essere applicabile, in linea di principio, indipendentemente dalla grandezza fisica dei dati; ad esempio l'algoritmo di Euclide vale per qualsiasi coppia di interi positivi, grandi a piacere). Occorre tuttavia notare come tutti i linguaggi di programmazione pongono dei limiti fisici nella grandezza dei dati di ingresso; ad esempio in `Pascal` un numero intero deve essere compreso tra un valore minimo ed uno massimo (ad esempio, in alcuni compilatori commerciali, un *integer* è compreso tra -32768 e $+32767$).
- Non ci deve essere alcun limite fisico nella quantità di memoria disponibile. Ciò significa che i dati di un algoritmo devono poter essere memorizzati (sempre in linea di principio) su un numero arbitrariamente grande di fogli. Occorre tuttavia notare come la memoria fisica dei calcolatori è fisicamente limitata; ogni elaboratore è caratterizzato da una dimensione finita della sua memoria principale (la memoria `RAM`) e secondaria (capacità dell'hard-disk e degli altri dispositivi di memorizzazione).
- Deve esserci un limite finito al numero di operazioni ed alla loro complessità (sia per quanto riguarda l'esecuzione di un algoritmo, come specificato nella definizione), e sia per quanto riguarda il programma. Osserviamo inoltre che non possiamo costruire un algoritmo composto di istruzioni non elementari (ad esempio, non possiamo dire ad un elaboratore di "calcolare il massimo comun divisore" senza specificare ulteriormente i passi che deve

compiere in termini di istruzioni elementari che possano essere interpretate da esso. Per lo stesso motivo, non possiamo dire ad una persona di assemblare o di installare un elettrodomestico senza dargli precise istruzioni sulle fasi dell'assemblaggio, che deve avvenire come successione di passi elementari. Si pensi ad esempio ai manuali di istruzione per l'istallazione di un videoregistratore, che illustrano passo passo come collegare i cavi, come alimentare l'apparecchio, e come programmarne la data, l'ora e la lingua visualizzata sul display).

- Sono ammesse istruzioni con un numero di passi illimitato. Ad esempio, abbiamo visto che l'algoritmo E termina dopo soli 7 passi con $m = 6$ ed $n = 4$, mentre occorrono 19 passi con $m = 40$ ed $n = 11$. Il numero di passi per giungere alla soluzione dipende pertanto dai valori degli ingressi, e non deve essere limitato.
- Per quanto riguarda i *metodi computazionali* (e non gli algoritmi), sono ammesse esecuzioni con un numero di passi infinito, come segue dalle definizione stessa di metodo computazionale.

Prima di concludere questo paragrafo, mostriamo un esempio di metodo computazionale, per capire cosa sia. Il metodo computazionale P che ci apprestiamo a descrivere calcola tutte le potenze ad esponente positivo di un numero intero positivo n. Ad esempio, se $n = 3$, esso calcola $3^1 = 3$, $3^2 = 9$, $3^3 = 27$, e così via, senza mai arrestarsi.

Metodo computazionale M: (*Calcolo delle potenze di un numero intero*). Dati un interi positivo n, questo metodo calcola tutte le sue potenze ad esponente positivo.

Passo 1: [Inizializzazione dell'esponente] Assegna all'esponente p il valore 1 ($p \leftarrow 1$).

Passo 2: [Calcola la potenza] Calcola $m = n^p$.

Passo 3: [Incrementa l'esponente] Aggiungi 1 al valore di p ($p \leftarrow p+1$).

Passo 4: [Salto] Torna al passo 2. ■

Notiamo che il simbolo ■ è utilizzato per indicare la fine della *descrizione* di un algoritmo o di un metodo computazionale, e non per indicare la sua *terminazione* (che è indicata esplicitamente con le parole "Fine algoritmo", ovviamente non presenti nella descrizione di un metodo computazionale).

5. Conseguenze dei limiti fisici di memoria

La limitazione fisica della memoria disponibile in un elaboratore, pone ovviamente dei limiti alle sue possibilità di utilizzo. Come esempio, consideriamo un problema di Teoria dei Numeri risalente al 1742, in apparenza semplice, ma che risulta tuttora irrisolto, e noto come *congettura di Goldbach*. Il matematico C.F. Golbach, in una lettera a L. Eulero, congetturò che *ogni numero pari n maggiore di 2 può essere espresso come somma di due numeri primi p e q (cioè $n = p+q$)*. Per piccoli numeri, è facile verificare la validità di questa affermazione: ad esempio si ha $4 = 2 + 2$, $6 = 3 + 3$, $8 = 5 + 3$, $10 = 7 + 3$ e così via. Tuttavia, affermare che questa ciò valga per *tutti* gli interi pari maggiori di 2 è un problema non ancora risolto. Ci domandiamo in che modo il calcolatore possa aiutarci nel verificare questa congettura. Innanzitutto, il calcolatore *non* può verificare la *validità* della congettura per ogni intero pari maggiore di 2; infatti occorrerebbe fare la prova con infiniti numeri, e ciò richiederebbe un tempo infinito. Il calcolatore ci potrebbe aiutare nel dimostrare che questa congettura è *falsa*; se fosse infatti possibile trovare *un solo numero n* tale che esso *non* sia somma di due numeri primi, potremmo tranquillamente affermarne la sua falsità. Questo metodo è già stato applicato (anche utilizzando solamente carta e penna) in altri problemi dello stesso genere; ad esempio il matematico Pierre

de Fermat (1601-1605) congetturò che tutti i numeri $f_n = 2^{2^n} + 1$ (con n intero maggiore di zero) fossero primi, mentre Eulero enunciò nel 1739 la falsità di questa congettura, mostrando che per $n = 5$ il numero $f_5 = 2^{2^5} - 1$ è divisibile per 641. Potremmo allora pensare di usare il calcolatore per cercare un solo controesempio alla congettura di Goldbach, così come fece Eulero nei confronti della congettura di Fermat. Ciò è già stato tentato, con insuccesso, dovuto al fatto che la limitazione di memoria ci impedisce di immagazzinare numeri grandi all'interno dell'elaboratore, e pertanto sino ad oggi non è stato possibile trovare alcun controesempio che dimostrasse la falsità della congettura.

Anche se costruissimo un elaboratore gigantesco, utilizzante come memoria l'intera materia presente nell'universo, potremmo essere ugualmente condannati all'insuccesso. Infatti si stima che nell'universo conosciuto ci siano al più 10^{123} particelle elementari (protoni, neutroni, elettroni). Se pensassimo di utilizzare queste particelle per formare tanti atomi di idrogeno, ciascuno da utilizzare come elemento di memoria in grado di immagazzinare informazione sotto forma di 0 (atomo a riposo, con l'elettrone sull'orbitale 1s) o di 1 (mediante eccitazione dell'elettrone, che viene portato sull'orbitale 1p), il numero più grande che potremmo memorizzare sarebbe $b = 2^{10^{123}}$. Se pertanto per tutti i numeri pari compresi tra 2 e b la congettura fosse verificata, non avremo ugualmente alcuna informazione utile; potrebbe anche capitare che essa risulti falsa per il successivo numero pari, cioè per il numero $b+2$.

6. La macchina di Turing

E se disponessimo di un calcolatore avente memoria infinita? Ci sarebbero ancora limitazioni su ció che potremmo fare con una macchina calcolatrice? Per rispondere a questa domanda, il matematico inglese A.M. Turing (1912-1954) ideò un *modello matematico* in grado di rappresentare di un calcolatore ideale (cioè dotato di memoria infinita), costituito da:

- Un nastro di lunghezza infinita, sul quale vengono letti e scritti dei simboli di un alfabeto costituito da un insieme finito di simboli (ad esempio, i simboli binari 0 ed 1). Avendo lunghezza infinita, tale nastro corrisponde ad una memoria infinita.
- Una testina, in grado di leggere sul nastro un simbolo, scrivere un simbolo, spostarsi a destra ed a sinistra di una posizione per volta. La testina svolge il ruolo di dispositivo di input/output.
- Una unità di controllo, che dopo aver comandato alla testina di leggere un simbolo, si porta in un nuovo stato che dipende dal simbolo letto, ordina alla testina di scrivere un simbolo sul nastro, e comanda ad essa di spostarsi a destra o a sinistra di una posizione, oppure di fermarsi, arrestando la computazione. L'unità di controllo corrisponde al programma presente nella memoria di un calcolatore.

Lo schema di principio della macchina di Turing (abbreviazione: MT) è riportato in figura 4.

L'unità di controllo è dotata di una memoria interna avente dimensione finita, capace di memorizzare la "storia" della computazione, che dipende dai valori letti dalla testina sul nastro durante la computazione stessa. Questa memorizzazione è detta *stato* della macchina. Lo stato è indicato con una lettera, e dato che la macchina possiede più di uno stato, conviene indicare quest'ultimi con le lettere A, B, C, eccetera. Tutte le volte che la testina legge un simbolo, la macchina entra in un nuovo stato, dipendente dal simbolo letto, dopodichè la testina scrive su nastro un opportuno simbolo, e si sposta infine di un posto a destra o a sinistra.

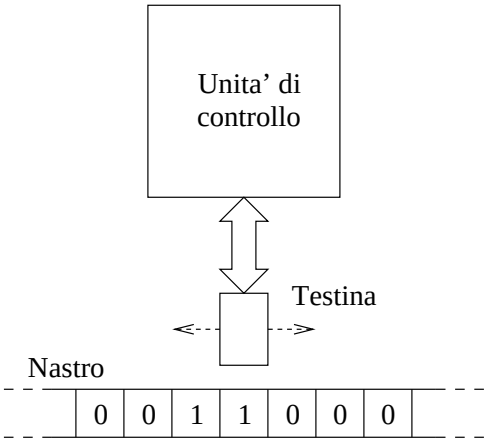


FIGURA 4. La macchina di Turing

6.1. **Esempio di computazione con una macchina di Turing.** A titolo di esempio, "programmiamo" una macchina di Turing al fine di compattare due blocchi di zeri, separati da un blocco di 1. Ciò che vogliamo ottenere è illustrato in figura 5, ove (a) rappresenta il nastro *prima* della computazione, e (b) lo stesso a computazione terminata.

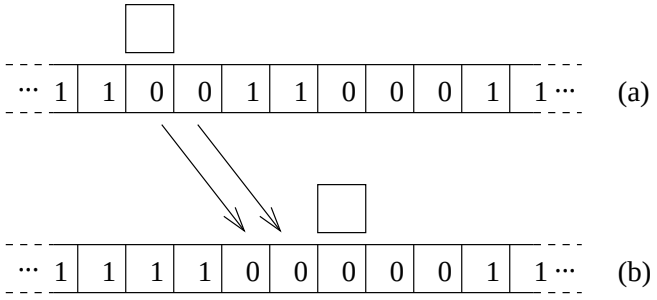


FIGURA 5. Compattazione di due blocchi di 0

Il "programma" consiste in una tabella, detta *tabella degli stati*, nella quale appaiono 3 colonne (vedi tabella 1).

	$x = 0$	$x = 1$
z_a	z_s, y, m	z_s, y, m
A	B, 1, d	–,–,–
B	B, 0, d	C, 0, d
C	Alt	D, 1, s
D	D, 0, s	A, 1, d

TABELLA 1. Tabella degli stati della macchina di Turing

Nella prima colonna si trova lo *stato attuale*, indicato con z_a, in cui si trova l'unità di controllo *prima* della lettura del simbolo sul nastro. Il simbolo letto dal nastro è indicato con x; nel nostro caso x può assumere solo i valori 0 ed 1. Nella seconda e terza colonna colonna troviamo indicati: lo stato successivo (indicato con z_s), in

cui si porta l'unità di controllo dopo la lettura del nastro, il simbolo y da scrivere sul nastro (0 oppure 1) nella stessa casella dove è stata effettuata l'ultima lettura, e una lettera (m) che vale 'd' oppure 's', e che indica la direzione di movimento della testina (destra o sinistra) dopo l'operazione di scrittura. La seconda colonna si riferisce al caso in cui venga letto il simbolo $x = 0$, la terza corrisponde alla lettura di $x = 1$. Ad esempio, se la macchina si trova nello stato attuale A, e viene letto il simbolo $x = 0$, la macchina si porta nel nuovo stato B, viene scritto 1 sul nastro, e la testina si sposta di una casella verso destra. Troviamo anche il simbolo 'Alt', che indica alla macchina di fermarsi. Ciò accade quando la macchina si trova nello stato z_s = C, e viene letto il simbolo $x = 0$. Queste informazioni possono anche non essere specificate qualora non necessario; ad esempio, vedremo che la macchina, quando si trova nello stato attuale A, non leggerà mai il simbolo $x = 1$ dal nastro; per questo motivo nella terza colonna della prima riga della tabella non compare nessuno stato successivo, nessun simbolo da scrivere, nessuno spostamento.

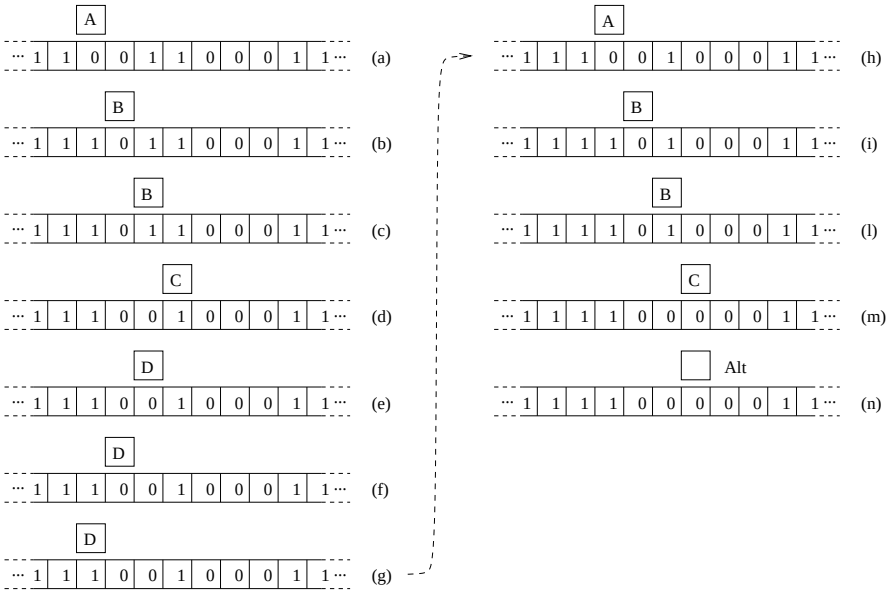


FIGURA 6. Compattazione dei blocchi di 0

Vediamo di eseguire, passo passo, questa procedura di compattazione. Per far ciò, supponiamo che la macchina si trovi inizialmente nello stato A, con la testina posizionata sotto lo zero più a sinistra del nastro (figura 6 (a)). Per comodità, disegnamo testina ed unità di controllo come un tutt'uno.

In questa posizione, la testina legge il simbolo $x = 0$, e di conseguenza la macchina passa nello stato successivo B, scrive $y = 1$ sul nastro, e si sposta a destra di una posizione (vedi figura 6 (b) e tabella 1).

Dopo lo spostamento della testina, il nuovo stato attuale della macchina è B. In questo stato, la testina legge il simbolo 0, e di conseguenza lo stato successivo rimane ancora B, la testina scrive ancora 0 sul nastro, e si sposta a destra di un'altra posizione, come illustrato in (c).

Ora, ci troviamo nello stato B, e la testina legge il simbolo 1. Dalla tabella 1 possiamo dedurre che il nuovo stato sarà C, che verrà scritto il simbolo 0 sul nastro, e che si avrà uno spostamento verso destra della testina, come illustrato in (d).

Osservando che la testina leggerà il simbolo 1, dalla tabella 1 ricaviamo che il nuovo stato sarà D, che verrà scritto 0 sul nastro, e che la testina si sposterà verso

sinistra, come illustrato in figura 6 (e). Qui avverrà la lettura del simbolo 0, e di conseguenza la testina si sposterà ancora a sinistra, dopo aver scritto 0 sul nastro. Il nuovo stato è ancora D (vedi figura 6 (f)).

A questo punto, viene letto ancora 0, e di conseguenza lo stato rimarrà D, la testina scriverà 0, e si sposterà di nuovo a sinistra (figura 6 (g)).

Siamo ora in presenza di un 1, che verrà letto, e conseguentemente la macchina si porterà nello stato A, scriverà 1 e sposterà la testina a destra (figura 6 (h)).

È importante notare come la posizione della testina in (h) sia del tutto simile alla posizione di partenza illustrata in (a): lo stato è quello iniziale (A), e la testina si trova sotto lo zero più a sinistra presente sul nastro. Abbiamo però compattato il primo zero; tra i due blocchi di zero ora rimane solo un 1.

I passi (i), (l), (m) sono di conseguenza simili ai passi (b), (c), (d) analizzati in precedenza. Arrivati nella situazione illustrata in (m), possiamo osservare che la macchina ha svolto con successo il suo compito: i due blocchi di 0 sono stati compattati. A questo punto, l'unità di controllo si trova nello stato C, la testina legge uno 0 e ciò arresta la macchina. La computazione è terminata.

Notiamo che la compattazione dei due blocchi di zeri può essere altresì descritta mediante un algoritmo. Se supponiamo di avere un foglio di carta con rappresentati i due blocchi di zeri, separati da un blocco di 1, e vogliamo eseguire la compattazione a mano, eliminando un 1 alla volta, possiamo procedere come illustrato nell'algoritmo C.

Algoritmo C: (*Compattazione di due blocchi di zeri*). Dati due blocchi di 0 separati da un blocco di 1, questo algoritmo provvede alla loro compattazione, eliminando un 1 alla volta. La penna si trova inizialmente sotto lo 0 più a sinistra.

Passo 1: [Cancellazione dello 0] Sostituirle lo 0 con un 1.

Passo 2: [Sostituzione del primo 1 trovato] Spostare la penna verso destra, fermandosi quando si è trovato il primo 1 e sostituirlo con uno 0.

Passo 3: [Ispezione] Spostare la penna di un posto verso destra. Se il simbolo trovato è uno 0, saltare al passo 6.

Passo 4: [Riposizionamento] Spostare la penna verso sinistra, sino a trovare un 1.

Passo 5: [Salto] Spostare la penna verso destra di un posto (si troverà così sotto lo zero più a sinistra) e tornare al passo 1.

Passo 6: [Fine] La compattazione è terminata. Fine. ■

6.2. **Una computazione infinita.** Una macchina di Turing potrebbe anche non arrestarsi mai. In questo caso, essa rappresenta un *metodo computazionale*.

	$x = 0$	$x = 1$
z_a	z_s, y, m	z_s, y, m
A	B, 0, d	B, 0, d
B	A, 1, d	A, 1, d

TABELLA 2. Rappresentazione di un metodo computazionale con una MT

Ad esempio, la macchina descritta in tabella 2 non si arresta mai, ma continua a scrivere coppie di 0 e di 1 sul nastro, spostando continuamente la testina verso destra, come illustrato in figura 7, indipendentemente dal contenuto iniziale del nastro.

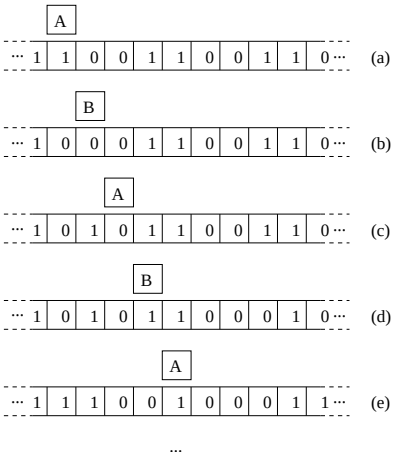


FIGURA 7. Esempio di macchina di Turing che non si arresta mai

Questo metodo poteva essere descritto a parole come segue:

Metodo computazionale S: (*Scrittura di coppie di 0 e di 1*). Dato un nastro di carta di lunghezza infinita, vengono scritte infinite coppie di 0 e di 1.

Passo 1: [Scrivi 0] Scrivi 0 sul nastro e sposta la penna a destra di un posto.

Passo 2: [Scrivi 1] Scrivi 1 sul nastro e sposta la penna a destra di un posto.

Passo 3: [Salto] Torna al passo 1. ■

Una macchina di Turing è pertanto in grado di rappresentare non solo algoritmi, ma anche metodi computazionali.

6.3. **Il problema della fermata.** Nel paragrafo 6.1, abbiamo visto un esempio di come un algoritmo (nel nostro caso la compattazione dei due blocchi di zeri) possa essere rappresentato mediante una macchina di Turing. Pertanto, la macchina di Turing è un efficace metodo di rappresentazione degli algoritmi, alternativo ai linguaggi di programmazione. La rappresentazione di un algoritmo mediante una MT è utile per considerazioni teoriche, quella mediante linguaggi di programmazione per l'esecuzione dello stesso utilizzando un elaboratore.

Torniamo al problema posto all'inizio del paragrafo 6, ovvero se la macchina di Turing presenta delle limitazioni in ciò che può elaborare, nonostante abbia memoria infinita.

Il matematico Alan Turing dimostrò che anche la MT presenta delle limitazioni; sussiste infatti il seguente risultato, noto come il *problema della fermata*:

Teorema 6.1. Non esiste alcun algoritmo capace di determinare se una arbitraria macchina di Turing in una configurazione iniziale arbitraria si fermerà oppure no.

Per questo motivo, il problema della fermata è detto *indecidibile*.

Cerchiamo di capire meglio il significato di questo teorema: esso ci dice che non è possibile scrivere un algoritmo, che abbia come ingressi lo stato iniziale di una arbitraria MT e la configurazione iniziale del suo nastro, e che determini se tale MT fermerà o no la sua computazione. Osservando che tale algoritmo (se esistesse) potrebbe essere rappresentato con una seconda macchina di Turing, segue che:

Corollario 6.1. Non esiste alcuna macchina di Turing capace di determinare se una arbitraria MT fermerà o no la sua computazione.

Le conseguenze sono assai importanti; infatti:

- Una MT ha una memoria *infinita*, un qualsiasi elaboratore ha una memoria *finita*.
- Pertanto una MT possiede capacità di calcolo superiori a *qualsiasi* elaboratore.
- Di conseguenza, ciò che una MT *non* può fare, *non* può nemmeno essere fatto da un qualsiasi elaboratore (il viceversa non è vero: su ciò che una MT può fare non possiamo dire nulla).
- E pertanto, *non* è possibile scrivere alcun programma su un calcolatore che sia in grado di stabilire se un altro elaboratore terminerà la sua computazione o no.

Ad esempio, immaginiamo di avere degli elaboratori collegati in rete, all'interno di un laboratorio di informatica. Uno studente, potrebbe scrivere ed eseguire, sul suo elaboratore, il seguente programma, che visualizza in continuazione, senza mai arrestarsi, la parola "Ciao":

```
PROGRAM infiniticiao (output);
   LABEL 1;
   BEGIN
      1: writeln('Ciao');
         GOTO 1;
   END.
```

Per quanto detto, l'*insegnante non sarà mai in grado* di scrivere un programma sul suo computer capace di stabilire se il programma che lo studente sta eseguendo sul suo terminale avrà termine o no.

L'importanza della macchina di Turing è quindi legata a problemi di Informatica Teorica; infatti mediante il modello MT è possibile stabilire se un problema è risolvibile mediante un algoritmo (e di conseguenza mediante un programma); ogniqualvolta si dimostra la *non esistenza di un algoritmo* per risolvere un dato problema (o della sua rappresentazione mediante MT), siamo certi che *nessun calcolatore* potrà mai risolvere quel problema.

Notiamo tuttavia come il problema della fermata, che non ammette soluzioni algoritmiche, è tuttavia risolubile utilizzando l'*ingenuità umana*. Ad esempio, se guardiamo la tabella 2 possiamo dire che quella MT non si fermerà mai, dato che in questa tabella non troviamo il simbolo 'Alt'. Analogamente, se analizziamo il programma `infiniticiao`, possiamo dire che esso non terminerà mai; infatti, dopo l'istruzione `writeln('Ciao')` che scrive la parola 'Ciao' su video, troviamo l'istruzione `GOTO 1` che salta al punto individuato dall'etichetta `1`, ovvero ritorna ad eseguire l'istruzione `writeln('Ciao')` senza che si raggiunga mai l'istruzione `END`, che termina il programma.

7. La bontà di un algoritmo

L'esecuzione di un programma (cioè di una *rappresentazione*) di un algoritmo mediante l'uso di un calcolatore richiede un utilizzo di *spaziali* e *temporali*, dipendenti dalla quantità di dati di ingresso che devono essere elaborati. Per risorse spaziali si intende la quantità di memoria utilizzata, nonchè il numero dei dispositivi di calcolo e di memorizzazione impegnati. Ad esempio, un ordinamento di una grande quantità di dati presente su un disco fisso coinvolge l'utilizzo di una certa quantità di memoria RAM, di una certa porzione di disco fisso, di un certo numero di registri del microprocessore. Per calcoli scientifici o elaborazioni grafiche di immagini sono a volte necessari più elaboratori collegati in rete, oppure sistemi multiprocessore; in questo caso interessa anche sapere quale è il numero di elaboratori utilizzati e

quanti microprocessori sono impegnati nel calcolo. Per risorse temporali, si intende il tempo di elaborazione, di solito dipendente dalla quantità e dal tipo di dati di ingresso. La bontà di un algoritmo è legata alla quantità di risorse impegnate per la sua esecuzione, e dipende anche dal linguaggio di programmazione utilizzato. Ad esempio, l'utilizzo di linguaggi *interpretati* come il `LISP`, in cui l'elaboratore decodifica (cioè trasforma in una successione di 0 e 1) ed esegue un comando alla volta, presentato una lentezza di esecuzione molto superiore rispetto ai linguaggi *compilabili* (come ad esempio il `Pascal` o il C), che vengono tradotti *prima* dell'esecuzione in una sequenza di 0 e 1 rapidamente interpretabili dalla macchina. In quest'ultimo caso, l'operazione di traduzione (detta *compilazione*) del programma in un codice binario è fatta una sola volta, *prima* di eseguire il programma, mentre nel caso di linguaggi interpretati ogni singola esecuzione del programma è sempre associata ad una interpretazione dei comandi.

Occorre notare che un consumo eccessivo di risorse può rendere inutilizzabile un algoritmo. Si consideri ad esempio il metodo di Cramer per il calcolo delle soluzioni di un sistema lineare di n equazioni in n incognite. Se il sistema é di piccole dimensioni (cioé con 2, 3 o al piú 4) incognite, puó essere risolto sia manualmente che con l'ausilio di un calcolatore con il *metodo di Cramer* (vedi [15]). Osserviamo che utilizzando questo metodo servono $N = (n + 1)n!$ operazioni per risolvere il sistema. Questo numero cresce assai rapidamente con il numero di equazioni; ad esempio su un elaboratore in grado di eseguire 10^9 operazioni in virgola mobile (cioé con numeri reali) al secondo, occorrerebbe circa un decimo di secondo per risolvere un sistema di 10 equazioni, circa quindici ore per un sistema di 15 equazioni, e piú di 4000 anni per un sistema di 20 equazioni. E' allora importante utilizzare altri metodi che consentano la risoluzione del sistema in un tempo ragionevole. Uno di questi é il *metodo di eliminazione di Gauss*, per il quale si rimanda ancora a [15]. Notiamo che anche utilizzando un elaboratore 1000 volte più veloce, non cambia la sostanza del problema: occorrerebbero sempre 4 anni per risolvere un sistema di sole 20 equazioni. E dato che nella tecnica si presentano sistemi con centinaia di incognite, questo metodo risulta inutilizzabile per il calcolo delle soluzioni di un sistema lineare di grandi dimensioni. Cioè, *la bontà di un algoritmo non dipende dalla velocità dell'elaboratore*, ma è legata a quanto velocemente cresce il tempo di calcolo (oppure la quantità di memoria utilizzata) al crescere della dimensione dei dati di input (in questo caso, al crescere del numero delle equazioni).

In questo caso, solo la sostituzione di un algoritmo con un altro permette di risolvere il problema del calcolo del calcolo delle soluzioni in un tempo ragionevole.

7.1. **La misura della complessità di un algoritmo.** Per valutare la *bontà* di un algoritmo, si definiscono i seguenti parametri:

Definizione 7.1. Sia x l'input di un algoritmo A. Il tempo di calcolo $T_A(x)$ è definito come il tempo di esecuzione dell'algoritmo A. Lo spazio occupato $S_A(x)$ è la quantità di memoria utilizzata dall'algoritmo A, sempre con ingresso x.

Notiamo alcune cose:

- Il tempo di calcolo e lo spazio occupato dipendono sia dall'algoritmo A che dall'input scelto x. Ad esempio, se A è un algoritmo di ordinamento di numeri, il tempo e lo spazio occupato dipendono dalla dimensione dell'input (un conto è ordinare 10 numeri, un'altro ordinarne 1000), e dalle caratteristiche dei dati stessi (le operazioni di confronto tra numeri reali avvengono in un tempo differente rispetto a quelle effettuate su numeri interi; inoltre lo spazio di memoria occupato da numeri reali è diverso da quello occupato da numeri interi).

- Se consideriamo con I l'insieme di tutti gli input possibili, esisterà un particolare input $x_{pT} \in I$ per il quale $T_A(x_p)$ è massimo, analogamente esisterà un altro input x_{pS} (che può essere anche distinto da x_{pT}) che occuperà il maggior spazio in memoria. Definiamo in questo caso la complessità (temporale e spaziale) nel *caso peggiore* come:

$$T_{pA}(x) = \max_{x \in I} T_A(x), \quad S_{pA}(x) = \max_{x \in I} S_A(x) \tag{1}$$

- Una delle difficoltà nel calcolo di T_{pA} ed S_{pA} consiste nel fatto che è pressochè impossibile considerare tutti i possibili input appartenenti ad I. Per questo motivo, indicata con n la dimensione dei dati di input, si può calcolare la complessità di un algoritmo per ogni input x avente dimensione n. Ad esempio, se l'algoritmo mette in ordine delle stringhe di caratteri, n è il numero di stringhe da ordinare; se l'algoritmo numera un cruciverba, n è il numero di caselle del cruciverba stesso; se l'algoritmo genera i primi n numeri perfetti, si puó considerare n come la sua dimensione. Indicheremo con $|x|$ la dimensione dell'input x. In questo caso, T_{pA} ed S_{pA} possono essere calcolati, per n fissato, considerando tutti gli input $x \in I$ di dimensione n, cioé tali che $|x| = n$. Inoltre, essendo n un numero naturale, conviene considerare anche T_{pA} ed S_{pA} come numeri naturali.

Dalle considerazioni precedenti, possiamo definire le funzioni T_{pA} ed S_{pA} nel modo seguente:

Definizione 7.2. La complessità temporale in caso peggiore di un algoritmo A è una funzione $T_{pA} : \mathbb{N} \longrightarrow \mathbb{N}$ tale che, per ogni $n \in \mathbb{N}$, si ha:

$$T_{pA}(n) = \max_{x \in I, |x|=n} T_A(x) \tag{2}$$

Definizione 7.3. La complessità spaziale in caso peggiore di un algoritmo A è una funzione $S_{pA} : \mathbb{N} \longrightarrow \mathbb{N}$ tale che, per ogni $n \in \mathbb{N}$, si ha:

$$S_{pA}(n) = \max_{x \in I, |x|=n} S_A(x) \tag{3}$$

Possiamo inoltre effettuare una media su tutti i possibili input di dimensione n, determinando così una complessità media, definita nel modo seguente:

Definizione 7.4. Sia $I_n \subseteq I$ il sottoinsieme di I costituito dagli input di dimensione n, e sia $|I_n|$ il numero di elementi di I_n. La complessità temporale media di un algoritmo A è una funzione $T_{mA} : \mathbb{N} \longrightarrow \mathbb{N}$ tale che, per ogni $n \in \mathbb{N}$, si ha:

$$T_{mA}(n) = \sum_{x \in I, |x|=n} \frac{T_A(x)}{|I_n|} \tag{4}$$

Definizione 7.5. La complessità spaziale media di un algoritmo A è una funzione $S_{mA} : \mathbb{N} \longrightarrow \mathbb{N}$ tale che, per ogni $n \in \mathbb{N}$, si ha:

$$S_{pA}(n) = \sum_{x \in I, |x|=n} \frac{S_A(x)}{|I_n|} \tag{5}$$

La valutazione del caso peggiore da molte volte una valutazione troppo pessimistica delle prestazioni di un algoritmo, per questo motivo in alcuni casi risulta più opportuno considerare la valutazione media. Quest'ultima può tuttavia essere inefficiente in quanto il verificarsi del caso peggiore può dare delle prestazioni assai inaccettabili rispetto alla stima media.

Quello che interessa, è comunque una valutazione per grandi valori di n, ovvero per input di grande dimensione. Interessa cioè lo studio della crescita della complessità (soprattutto quella temporale) al crescere di n, cioè al tendere di n a $+\infty$.

7.2. **Cenni sulla complessità computazionale temporale.** Per renderci conto dell'andamento della complessità temporale al crescere di n, consideriamo 6 algoritmi, di complessità temporale rispettivamente pari ad n, $n \log_2 n$, n^2, n^3, 2^n, 3^n, e supponiamo che l'elaboratore esegua una singola istruzione elementare in un nanosecondo (cioè 10^{-6} secondi). Possiamo riportare in tabella 3 i tempi di esecuzione degli algoritmi in funzione di n[1]. In questa tabella, abbiamo indicato i minuti con min, le ore con h, i giorni con g, gli anni con a, i secoli con c, e con ∞ un tempo superiore al milione di anni.

Tabella 3. Complessitá computazionale temporale

$T_A(n)$	$n = 10$	$n = 25$	$n = 50$	$n = 100$	$n = 1000$	$n = 10^4$	$n = 10^5$	$n = 10^6$
n	10 μs	25 μs	50 μs	100 μs	1 ms	10 ms	0,1 s	1 s
$n \log_2 n$	33,2 μs	116,2 μs	0,28 ms	0,6 ms	9,9 ms	0,1 s	1,6 s	19,9 s
n^2	0,1 ms	0,63 ms	2,5 ms	10 ms	1 s	100 s	2,7 h	11,5 g
n^3	1 ms	15,6 ms	125 ms	1 s	16,6 min	11,5 g	31,7 a	317 c
2^n	1 ms	33,6 s	35,7 a	$4 \cdot 10^{14}$ c	∞	∞	∞	∞
3^n	59 ms	9,8 g	$2{,}3 \cdot 10^8$ c	∞	∞	∞	∞	∞

Notiamo come gli algoritmi con complessità lineare (cioè proporzionale ad n), o al più di tipo $n \log_2 n$ siano utilizzabili efficientemente; questa efficienza già decade per algoritmi con complessità quadratica (cioè proporzionale ad n^2), si vede infatti che per $n = 10^6$ occorrono circa 11,5 giorni per l'elaborazione. Il risultato, è tuttavia ancora accettabile. Le prestazioni peggiorano per una complessità proporzionale ad n^3, per diventaer inaccettabili in caso di complessità esponenziale (cioè di tipo 2^n, 3^n o casi peggiori). In questo caso, l'elaborazione di un numero limitato di dati (esempio, $n = 50$) richiede già anni se non secoli di tempo, come si vede dalle ultime due righe della tabella.

È altrettanto interessante chiederci quale è la massima dimensione di un input che può essere elaborato in un'ora; il risultato è riportato in tabella 4.

Tabella 4. Dimensione dell'input computabile in un'ora

Complessità in tempo $T_A(n)$	Massima dimensione dell'input n
n	$3{,}6 \cdot 10^9$
$n \log_2 n$	$1{,}3 \cdot 10^8$
n^2	$6 \cdot 10^4$
n^3	$1{,}5 \cdot 10^3$
2^n	31
3^n	20

Supponiamo ora di utilizzare un elaboratore 10^6 volte piú veloce del precedente, e ripetiamo i calcoli sulla massima dimensione dell'input. I dati sono riportati in tabella 5; possiamo notare come un aumento enorme della velocità di calcolo influisce assai poco sulla possibilità di utilizzare algoritmi con capacità di calcolo esponenziale.

Come si vede, l'aumento di velocità del computer di 10^6 volte (ciò che si è verificato in circa 40 anni, passando dai personal computer con clock di qualche centinaio di kHz agli attuali PC con clock di qualche GHz) non basta assolutamente a rendere applicabili algoritmi con complessitá di calcolo esponenziale.

[1]per semplicità, supponiamo che essi dipendano solo da n e non da x, così che tutti gli input con $|x| = n$ abbiano la stessa complessità temporale; in questo modo la complessità media e quella del caso peggiore coincidono. Notiamo che la trattazione non cambia considerando il caso peggiore, oppure il caso medio.

TABELLA 5. Dimensione dell'input computabile in un'ora con una velocità di elaborazione 10^6 volte superiore

Complessità in tempo $T_A(n)$	Massima dimensione dell'input n
n	$3{,}6 \cdot 10^{15}$
$n \log_2 n$	$7{,}4 \cdot 10^{13}$
n^2	$6 \cdot 10^7$
n^3	$1{,}5 \cdot 10^5$
2^n	51
3^n	32

Ad esempio, supponiamo che la complessità di calcolo sia $T_A(n) = 2^n$. Se indichiamo con t_0 il tempo necessario all'esecuzione di una operazione elementare (nel caso del primo elaboratore da noi considerato, si ha $t_0 = 1\,\mu$s), e se indichiamo con M il fattore che indica il rapporto di velocità di esecuzione tra un nuovo elaboratore e il vecchio (nel nostro caso, avevamo scelto un aumento di velocità di 10^6 volte, quindi $M = 10^6$), possiamo dire che:

- Con il primo calcolatore, la dimensione massima n dell'input elaborabile entro un'ora (cioè 3600 secondi) deve soddisfare la relazione $2^n \cdot t_0 = 3600$, da cui si ottiene: $n = \log_2(3600/t_0)$.
- Con il secondo calcolatore, la dimensione massima n dell'input soddisfa la relazione: $(2^n \cdot t_0)/M = 3600$, da cui si ricava $n = \log_2(3600 \cdot M/t_0) = \log_2(3600/t_0) + \log_2(M)$ e come si vede, la dimensione dell'input è aumentata solo di $\log_2(M)$. Nel caso in esame, con $M = 10^6$, essa è aumentata solo di $\log_2(10^6) = 19{,}9 \approx 20$ volte. Tale aumento non dipende dal tempo imposto per l'elaborazione; infatti il termine temporale (3600 secondi) non compare nel termine aggiuntivo $\log_2(M)$.

La tabella 6 riporta l'accrescimento della dimensione dell'input calcolabile in un tempo fissato.

TABELLA 6. Effetto di un aumento di velocità di un fattore M sulla dimensione dell'input

Complessità in tempo $T_A(n)$	Dimensione dell'input n prima dell'aumento di velocità	Dimensione dopo l'aumento di un fattore M
n	n_1	$M \cdot s1$
$n \log_2 n$	n_2	$\approx M \cdot n_2$, per n_2 molto grande
n^2	n_3	$\sqrt{M} \cdot n_3$
n^3	n_4	$M^{1/3} \cdot n_4$
2^n	n_5	$n_5 + \log_2 M$
3^n	n_6	$n_5 + \log_3 M$

8. MODELLI DI CALCOLO

Passiamo ora all'analisi di diversi modelli matematici per il calcolo della complessità computazionale di un algoritmo. Analizzaremo anche alcune importanti relazioni tra questi modelli.

8.1. **La macchina RAM.** La macchina RAM, o *Macchina ad accesso casuale* (Random Access Machine) è individuata dallo stesso acronimo utilizzato per la memoria ad accesso casuale; si tratta tuttavia di due oggetti completamente differenti tra loro.

La macchina RAM è un modello matematico di un calcolatore, in parte simile alla macchina di Turing, ma più adatta allo studio della complessità computazionale degli algoritmi, in quanto l'algoritmo elaborato non è descritto in termini di stati, ma mediante l'esecuzione di un programma in un linguaggio di basso livello, che chiameremo *assembler RAM*, abbreviato con ASMRAM.

Essa è rappresentata in figura 8; consiste di una unità centrale, contenente il programma, di due nastri di lunghezza semiinfinita (cioè illimitati a destra), di un numero infinito di locazioni di memoria, dette registri (indicati con R_0, R_1, ..., e di un registro detto *contatore di programma* (abbreviato con c_p) che individia la riga di programma da eseguire. La macchina puó utilizzare il nastro 1 solo in lettura, ed il nastro 2 solo in scrittura. Ogniqualvolta viene letto un simbolo dal nastro 1, la corrispondente testina si sposta di una casella verso destra. Lo stesso avviene quando la testina di scrittura ha scritto un dato sul nastro 2. A differenza della macchina di Turing, le testine si muovono solo verso destra. Pertanto, è possibile leggere un dato solo una volta, e quando un dato viene scritto sul nastro 2, non è più possibile cambiarlo. Il contenuto dei nastri è costituito da numeri interi (anche negativi), oppure da caselle bianche. Prima dell'esecuzione del programma, il nastro di scrittura è bianco, cioè non contiene dati. Il nastro di lettura può contenere un numero arbitrario di dati, ed anche delle caselle vuote. Prima dell'esecuzione del programma, le testine si trovano sotto le caselle iniziali dei nastri (le caselle numerate con 0). Indicheremo una casella vuota (sia per il nastro 1 che per il nastro 2) come un *blank*, abbreviato con b_k. L'unità centrale è in grado di contenere programmi di lunghezza arbitrariamente grande.

Il programma è contentuto nell'unità centrale, separata dai registri che costituiscono la memoria della macchina. Il programma può modificare il contenuto dei registri, ma non se stesso. Tutte le operazioni aritmetiche, logiche e di confronto avvengono utilizzando il dato contenuto nel registro R_0, detto *accumulatore*. I registri possono contenere solo numeri interi. Prima dell'esecuzione del programma, tutti i registri contengono il numero 0.

Il linguaggio di programmazione ASMRAM è composto di 13 istruzioni, e tranne l'ultima (l'istruzione `HALT`) le rimanenti 12 sono costituite da due parti: un *codice operativo* e da un *operando*, che è detto *etichetta* per le operazioni di salto (`JUMP`, `JGTZ`, `JZERO`, `JBLANK`). La tabella 7 contiene le istruzioni della macchina RAM

Tabella 7. Tabella delle istruzioni della macchina RAM

Numero istruzione	Codice operativo	Indirizzo
1	`LOAD`	operando
2	`STORE`	operando
3	`ADD`	operando
4	`SUB`	operando
5	`MULT`	operando
6	`DIV`	operando
7	`READ`	operando
8	`WRITE`	operando
9	`JUMP`	etichetta
10	`JGTZ`	etichetta
11	`JZERO`	etichetta
12	`JBLANK`	etichetta
13	`HALT`	

Un operando può essere di tre tipi:

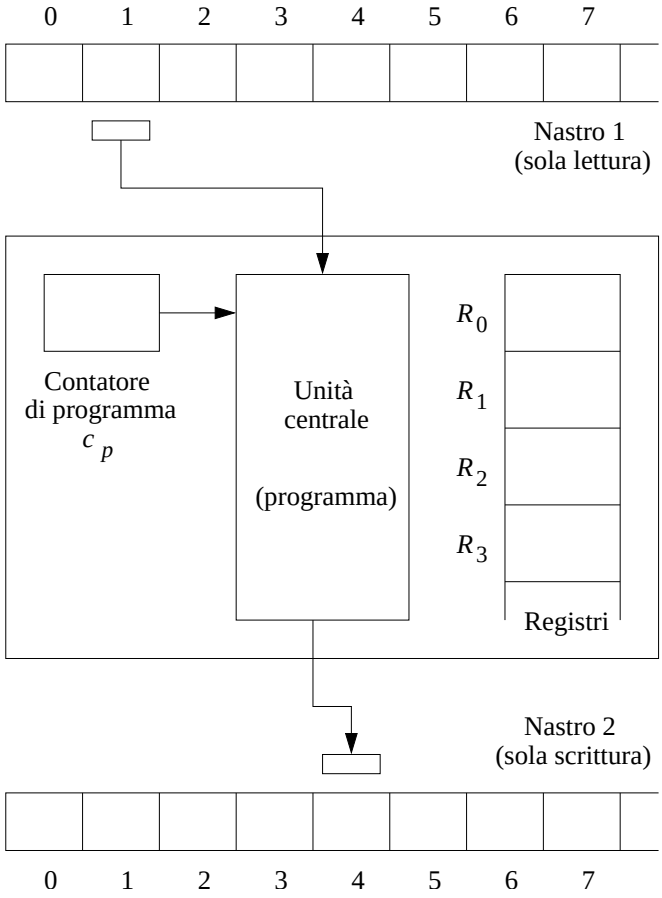


FIGURA 8. La macchina RAM

- $= i$, che indica il numero intero i,
- i, (con i intero maggiore o uguale a zero), e che indica il contenuto del registro R_i. Se $i < 0$ la macchina si arresta.
- $\star i$ (con i intero maggiore o uguale a zero), che indica il contenuto del registro R_j, ove j é l'intero (maggiore o uguale a zero) contenuto nel registro R_i. Nel caso in cui $j < 0$ oppure $i < 0$, la macchina si arresta.

Ad esempio, supponiamo che il contenuto dei registri sia quello rappresentato in figura 9.

	Registri
R_0	4
R_1	−2
R_2	6
R_3	5
R_4	−22
R_5	−71

FIGURA 9. Esempio di contenuti dei registri della macchina RAM

Sia $i = 3$. Allora:

- L'operando $= i$ (cioè $= 3$) indica l'intero i stesso, cioè il numero 3.
- L'operando i (cioè 3) indica il contenuto di R_i, cioè di R_3, cioè il numero 5.
- L'operando $\star i$ (cioè $\star 3$) individua il numero -71. Infatti il contenuto di $R_i = R_3$ è $j = 5$, ed il contenuto di $R_j = R_5$ è il numero -71.

Indicando con $c(R_i)$ il numero intero contenuto nel registro R_i, possiamo definire il significato degli operatori in modo matematico, introducendo il *valore* $v(a)$ di un operatore a, nel modo seguente:

- $v(= i) = i$
- $v(i) = c(R_i)$ (se $i < 0$ si ha che $v(i)$ è indeterminato, e la macchina si arresta)
- $v(\star i) = c(c(R_i))$ (se $i < 0$, oppure se $j < 0$ si ha che $v(i)$ è indeterminato, e la macchina si arresta)

Se $v(i)$ è indeterminato, lo indicheremo con il simbolo $\perp$, e scriveremo $v(i) = \perp$. In presenza di una qualsiasi indeterminazione, la macchina RAM si ferma.

Con riferimento al contenuto dei registri di figura 9, si ha:

$$v(= 3) = 3, \quad v(2) = 6, \quad v(-3) = \perp, \quad v(0) = 4, \quad v(\star 4) = \perp$$

Una etichetta è invece una descrizione simbolica (un numero, oppure una stringa di caratteri) associata ad una istruzione, in modo da consentire i salti (si pensi alle `label` del linguaggio `Pascal`.

Il significato delle istruzioni è illustrato sinteticamente nella tabella 8. In questa tabella, ove compare il simbolo a è possibile utilizzare tutti e tre gli operandi ($= i$, i, $\star i$), mentre nei casi in cui uno o più di essi non siano utilizzabili, vengono specificati a fianco dell'istruzione stessa. Indichiamo con $X_r(n)$ il simbolo letto dal nastro 1 dalla testina nella n-esima casella, e con $X_w(n)$ il simbolo scritto sul nastro 2 nella n-esima casella.

Tabella 8. Significato delle istruzioni della macchina RAM

Numero	Istruzione	Significato
1	`LOAD` a	$c(R_0) \leftarrow v(a)$
2	`STORE` i	$c(R_i) \leftarrow c(R_0)$
3	`STORE` $\star i$	$c(c(i)) \leftarrow c(R_0)$
4	`ADD` a	$c(R_0) \leftarrow c(R_0) + v(a)$
5	`SUB` a	$c(R_0) \leftarrow c(R_0) - v(a)$
6	`MULT` a	$c(R_0) \leftarrow c(R_0) \cdot v(a)$
7	`DIV` a	$c(R_0) \leftarrow c(R_0)/v(a)$
8	`READ` i	$c(R_i) \leftarrow X_r(n), \quad n \leftarrow n + 1$
9	`READ` $\star i$	$c(c(R_i)) \leftarrow X_r(n), \quad n \leftarrow n + 1$
10	`WRITE` a	$X_w(n) \leftarrow v(a), \quad n \leftarrow n + 1$
11	`JUMP` e	Salta alla riga di programma etichettata con e
12	`JGTZ` e	Se $c(R_0) > 0$ salta alla riga etichettata con e
13	`JZERO` e	Se $c(R_0) = 0$ salta alla riga etichettata con e
14	`JBLANK` e	Se $X_r(n) = b_k$ salta alla riga etichettata con e
15	`HALT`	Ferma la macchina

L'istruzione `JUMP` è detta di *salto incondizionato*, perchè salta in ogni caso alla riga di programma individuata dall'etichetta. Le rimanenti istruzioni di salto eseguono il salto solo se la condizione specificata in tabella è vera. In caso contrario,

il programma prosegue eseguendo l'istruzione successiva. Notiamo che l'istruzione `JBLANK` testa sul nastro 1 la presenza di un blank, ma *non* sposta la testina di lettura.

Analizziamo ora in maggiore dettaglio il linguaggio della macchina: incominciamo dall'istruzione `LOAD`, la quale può avere 3 operandi differenti:

- `LOAD` $= i$. L'istruzione copia nel registro R_0 il numero intero i. Ad esempio, l'istruzione `LOAD` $= -3$ copia in R_0 il numero intero -3.
- `LOAD` i. Copia in R_0 il contenuto del registro R_i. Con riferimento alla figura 9, l'istruzione `LOAD` 2 copia in R_0 il contenuto di R_2. Dopo la sua esecuzione, sia R_0 che R_2 conterranno il numero 6.
- `LOAD` $\star i$. Copia in R_0 il contenuto di R_j, ove j è il contenuto di R_i (con $i, j \geq 0$). Se prima dell'esecuzione il contenuto dei registri è quello della figura 9, l'istruzione `LOAD` $\star 3$ copia il numero -71 in R_0. Infatti essa prende prima il contenuto di R_3 (il numero $j = 5$), poi prende il contenuto di $R_j = R_5 = -71$ e lo copia in R_0.

Notiamo che l'istruzione `LOAD` sposta il contenuto di un qualsiasi registro in R_0. Per ottenere l'operazione inversa, cioè la copia da R_0 in un altro registro, si utilizza l'istruzione `STORE`.

- `STORE` i. Copia il contenuto di R_0 in R_i. Ad esempio, riferendoci alla solita figura, l'istruzione `STORE` 2 copia il numero 4 in R_2. Dopo l'esecuzione, sia R_0 che R_2 conterranno il valore 4.
- `STORE` $\star i$. Copia il contenuto di R_0 in R_j, ove j è il contenuto di R_i. Ad esempio, `STORE` $\star 3$ copia il numero 4 da R_0 in $R_j = R_5$. Infatti, il registro $R_i = R_3$ contiene il valore 5.

Analoghe considerazioni valgono per le operazioni aritmetiche, ad esempio, riferendoci sempre alla figura 9 che illustra la situazione prima dell'esecuzione di qualsiasi operazione elencata di seguito, si ha:

- `ADD` 3 aggiunge il contenuto di R_3 in R_0, cosí che alla fine dell'operazione, R_0 contiene il numero 9.
- `SUB` $\star 3$ sottrae al contenuto di R_0 il contenuto di $R_j = R_5$, cioè il numero -71. Al termine dell'operazione, il contenuto di R_0 è $4 - (-71) = 75$.
- `MULT` $= 2$. Moltiplica per il numero 2 il contenuto di R_0, che varrà $4 \cdot 20 = 8$.
- `DIV` $= 3$. Divide per il numero 3 il contenuto di R_0. Al termine, R_0 conterrà il *quoziente* della divisione tra gli interi 4 e 3, cioè il numero 1. Il resto della divisione non viene calcolato.

L'istruzione `READ` legge un dato dal nastro 1, e lo mette nel registro R_i (se l'istruzione è `READ` i), oppure nel registro R_j, con j contenuto di R_i, se l'istruzione è `READ` $\star i$. In entrambi i casi, la testina viene spostata verso destra di una casella dopo la lettura.

L'istruzione `WRITE` scrive sul nastro 2 un dato, e poi sposta la testina a destra di un posto. Ad esempio, l'istruzione `WRITE` $= 322$ scrive il numero 322 su nastro. Invece, l'istruzione `WRITE` 3 scrive il contenuto di R_3 sul nastro. Con riferimento alla solita figura, essa scrive 5. Infine, l'istruzione `WRITE` $\star 3$ scrive il contenuto di $R_j = R_5$ (cioè il numero -71) su nastro (sempre con riferimento alla figura 9, si ha $j = c(R_3) = 5$, da cui $c(R_5) = -71$).

Per le rimanenti operazioni, si rimanda alla tabella 8.

8.2. **Esempi di programmi RAM.** Scriviamo un programma che legge una successione di numeri interi positivi presenti sul nastro 1, fermandosi quando trova un blank, e che scriva sul nastro 2 il maggiore tra di essi. Utilizzeremo il registro R_0 per contenere il dato letto, ed il registro R_1 per contenere il massimo tra i numeri

letti. Ricordiamo che prima dell'esecuzione del programma, tutti i registri contengono il numero 0. Pertanto, già dalla lettura del primo numero è possibile fare il confronto tra il contenuto di R_1 ed il dato letto dal nastro (e posto in R_0). Al lettore è lasciata l'estensione del programma nel caso in cui i numeri presenti sul nastro 1 possano anche essere negativi. Scriviamo dapprima un algoritmo, e poi traduciamolo in linguaggio ASMRAM.

Algoritmo M: (*Ricerca del massimo tra un insieme finito di numeri interi positivi*). Data una successione $n_0, n_1, \ldots, n_k$ di numeri interi maggiori di zero, scritti su un nastro, l'algoritmo calcola il massimo m tra di essi. Nel caso in cui la successione sia vuota (cioè non esista alcun numero in input), l'algoritmo si ferma. Inizialmente, si ha $R_0 = R_1 = 0$.

Passo 1: [Finito?] Se la casella del nastro contiene un blank vai al passo 5.

Passo 2: [Leggi] Leggi il dato da nastro e copialo in R_0.

Passo 3: [Nuovo massimo?] Se $c(R_0) > c(R_1)$ allora $c(R_1) \leftarrow c(R_0)$.

Passo 4: [Salto] Torna al passo 1.

Passo 5: [Risultato] Il massimo è in R_1. Fine algoritmo. ■

Scriviamo ora il programma in linguaggio RAM. Notiamo come la difficoltà maggiore sia data dal passo 3 dell'algoritmo M; infatti per poter fare il confronto tra il contenuto di R_1 e quello di R_0 non disponiamo, in ASMRAM, dell'operatore $>$, ma possiamo soltanto testare se il contenuto di R_0 è uguale o diverso da zero, mediante le istruzioni di salto condizionato `JZERO` e `JGTZ`.

Nastro 1

FIGURA 10. Contenuto del nastro per il programma di calcolo del massimo

Pertanto, domandarsi se $c(R_0) > c(R_1)$ equivale a domandarsi se $c(R_0) - c(R_1) > 0$. Per far ciò, occorre sottrarre al contenuto di R_0 il contenuto di R_1, e poi utilizzare opportunamente l'istruzione `JGTZ`. Ma occorre anche salvare temporaneamente il contenuto di R_0 (che potrebbe essere il nuovo massimo), e per far ciò copieremo il suo contenuto in R_2. Inoltre, quando si è terminato di leggere il nastro, bisogna analizzare il contenuto di R_1 (che viene riportato in R_0; se esso è 0 allora nessun numero era presente su nastro, ed il programma non fa nulla; se invece esso è un numero diverso da zero (e necessariamente maggiore di zero per ipotesi sui dati di input), allora lo scrive sul nastro 2. Il programma è riportato in tabella 9; invece in figura 10 sono riportati anche i valori contenuti sul nastro 1. I corrispondenti valori contenuti, passo dopo passo, nei registri interessati, quando sul nastro 1 sono presenti i valori indicati nella figura stessa, sono riportati in tabella 10.

Notiamo come la *rappresentazione* di un algoritmo possa richiedere un numero di passi superiore rispetto a quelli dell'algoritmo stesso; nonchè delle maggiori difficoltà di scrittura (cioè di implementazione). Ciò è dovuto al fatto che l'algoritmo è scritto in un linguaggio naturale (la lingua italiana), molto piú ricca di vocaboli, espressiva e permissiva rispetto alle limitazioni di un linguaggio artificiale adatto alla programmazione.

Vediamo ora l'esecuzione passo dopo passo del programma (tabella 10), analizzando il contenuto dei registri quando l'input è riportato in figura 10. Per semplicità, si è indicato il contenuto dei registri R_0, R_1, R_2 con una terna di numeri del tipo (x, y, z); ad esempio la terna $(7, 2, 9)$ significa che il registro R_0 contiene il valore 7, R_1 contiene 2 ed R_2 contiene 9.

TABELLA 9. Programma RAM che rappresenta l'algoritmo M

N. istr.	Etichetta	Istruzione	Commento
1	*start*	`JBLANK` *fine*	Salta alla fine se trova un blank sul nastro 1
2		`READ` 0	Legge il dato dal nastro 1
3		`STORE` 2	Lo copia in R_2
4		`SUB` 1	Sottrae al contenuto di R_0 il contenuto di R_1
5		`JGTZ` *newm*	Salta sel risultato è maggiore di zero
6		`JUMP` *start*	$c(R_0)$ non è il nuovo massimo
7	*newm*	`LOAD` 2	Riporta il nuovo massimo in R_0
9		`STORE` 1	E lo salva in R_1
10		`JUMP` *start*	E ritorna alla prima istruzione
11	*fine*	`LOAD` 1	Copia il contenuto di R_1 in R_0
12		`JZERO` *stop*	Se $c(R_0) = 0$ esce senza scrivere nulla
13		`WRITE` 0	Scrive il massimo sul nastro 2
14	*stop*	`HALT`	Ferma la macchina

TABELLA 10. Esecuzione passo passo del programma

N. istr.	Etichetta	Istruzione	Passo 1	Passo 2	Passo 3	Passo 4
1	*start*	`JBLANK` *fine*	Non salta	Non salta	Non salta	Salta
2		`READ` 0	$(7, 0, 0)$	$(2, 7, 7)$	$(9, 7, 2)$	
3		`STORE` 2	$(7, 0, 7)$	$(2, 7, 2)$	$(9, 7, 9)$	
4		`SUB` 1	$(7, 0, 7)$	$(-5, 7, 2)$	$(2, 7, 9)$	
5		`JGTZ` *newm*	Salta	Non salta	Salta	
6		`JUMP` *start*		Salta		
7	*newm*	`LOAD` 2	$(7, 0, 7)$		$(9, 7, 9)$	
9		`STORE` 1	$(7, 7, 7)$		$(9, 9, 9)$	
10		`JUMP` *start*	Salta		Salta	
11	*fine*	`LOAD` 1				$(9, 9, 9)$
12		`JZERO` *stop*				Non salta
13		`WRITE` 0				Scrive 9 sul nastro 2
14	*stop*	`HALT`				Ferma la macchina

8.3. **L'importanza dell'indirizzamento indiretto.** L'operatore $\star a$, è detto operatore di indirizzamento *indiretto*, così come l'operatore a è detto indirizzamento diretto e $= a$ è detto indirizzamento immediato. Mentre l'uso degli operatori di indirizzamento diretto e immediato è facilmente comprensibile (ad esempio `MULT` $= -1$ cambia di segno il contenuto di R_0, e `ADD` 1 moltiplica il contenuto di R_0 per quello di R_1), a prima vista appare difficile capire quale sia la necessità di un indirizzamento indiretto. Consideriamo tuttavia un programma che legge un numero finito ma arbitrario di interi sul nastro 1 (terminando la lettura in presenza di un blank), e li scrive, in ordine inverso, sul nastro 2. Non sapendo quanti sono gli interi presenti sul nastro 1, è necessario tenerne il conto utilizzando un registro (ad esempio R_1 contenente il numero (o locazione) del primo registro libero in cui poter trasferire l'intero letto da nastro). Gli interi vengono posti nei registri R_2, R_3, ... Così, all'inizio della computazione, il primo registro libero è R_2, e pertanto R_1 viene inizializzato a 2. Man mano che viene letto un intero, viene trasferito nel registro libero R_j, ove j è il contenuto di R_1 ($j = 2, 3, \dots$), e viene incrementato il contenuto j di R_1 per individuare il prossimo registro libero. Per il trasferimento del dato nel registro libero, occorre utilizzare l'istruzione `STORE` $\star 1$, in cui si vede la

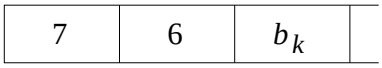
necessità di utilizzare l'indirizzamento indiretto[2]. Al termine della lettura, si può iniziare la fase di scrittura sul secondo nastro, decrementando tutte le volte di 1 il contenuto di R_1, e poi scrivendo su nastro il contenuto di R_j, con $j = c(R_1)$, fermandosi quando anche il contenuto di R_2 è stato visualizzato.

Il programma è rappresentato, insieme a brevi commenti, in tabella 11.

TABELLA 11. Scrittura all'inverso dei numeri letti

N. istr.	Etichetta	Istruzione	Commento
1	*start*	`LOAD` $=2$	Carica il numero 2 in R_0
2		`STORE` 1	E lo copia in R_1
3	*blk*	`JBLANK` *ciclo*	Salta se c'è un blank sul nastro
4		`READ` $\star 1$	Mette il dato letto in R_j, con $j = c(R_1)$
5		`ADD` $= 1$	Incrementa R_0 di 1
6		`STORE` 1	E lo copia in R_1
7		`JUMP` *blk*	Salta indietro a *blk*
8	*ciclo*	`SUB` $= 1$	Decrementa il contenuto di R_0 di 1
9		`STORE` 1	E lo copia in R_1
10		`SUB` $= 1$	Sottrae 1 al contenuto di R_0
11		`JZERO` *hlt*	Se $c(R_0) = 0$ ha finito
12		`WRITE` $\star 1$	Scrive $c(R_j)$ sul nastro 2, con $j = c(R_1)$
13		`LOAD` 1	Copia il contenuto di R_1 in R_0
14		`JUMP` *ciclo*	Ritorna all'inizio del ciclo di scrittura
15	*hlt*	`HALT`	Ferma la macchina

Supponendo che il contenuto del nastro 1 sia quello di figura 11, in tabella 12 è rappresentata l'esecuzione passo passo del programma, ove la quaterna (x, y, z, t) rappresenta il contenuto dei registri R_0, R_1, R_2, R_3.



Nastro 1

FIGURA 11. Contenuto del nastro 1 per il programma di scrittura alla rovescia

9. Esempi di calcolo di complessità con il modello RAM

Consideriamo ora due criteri di calcolo della complessità di un algoritmo; il primo di essi, che conta solo il *numero* di istruzioni eseguite ed il numero di registri occupati, è detto *criterio di costo uniforme*, mentre il secondo metodo, che considera tempi differenti a seconda del tipo di istruzione utilizzata e delle dimensioni dei dati di input, è detto *criterio di costo logaritmico*.

9.1. Il criterio di costo uniforme.

Cominciamo subito con una definizione:

Definizione 9.1. Per il criterio di costo uniforme, il tempo impiegato per l'esecuzione di un algoritmo è dato dal numero di istruzioni eseguite. Lo spazio impiegato è invece il numero di registri occupati dal programma.

[2]Come vedremo in seguito, se una macchina non dispone di tale possibilità, risulta impossibile scrivere un tale programma, a meno di permettere al programma di modificare se stesso, come avviene nel linguaggio della macchina RASP.

TABELLA 12. Scrittura all'inverso dei numeri letti

N.	Etich.	Istruzione	Passo 1	Passo 2	Passo 3	Passo 4	Passo 5
1	*start*	LOAD =*2*	$(2,0,0,0)$				
2		STORE 1	$(2,2,0,0)$				
3	*blk*	JBLANK *ciclo*	Non salta	Non salta	Salta		
4		READ $\star$1	$(2,2,7,2)$	$(3,3,7,6)$			
5		ADD = 1	$(3,2,7,2)$	$(4,3,7,6)$			
6		STORE 1	$(3,3,7,2)$	$(4,4,7,6)$			
7		JUMP *blk*	Salta	Salta			
8	*ciclo*	SUB = 1			$(3,4,7,6)$	$(2,3,7,6)$	$(1,2,7,6)$
9		STORE 1			$(3,3,7,6)$	$(2,2,7,6)$	$(1,2,7,6)$
10		SUB = 1			$(2,3,7,6)$	$(1,2,7,6)$	$(0,2,7,6)$
11		JZERO *hlt*			Non salta	Non salta	Salta
12		WRITE $\star$1			Scrive 6	Scrive 7	
13		LOAD 1			$(3,3,7,6)$	$(2,2,7,6)$	
14		JUMP *ciclo*			Salta	Salta	
15	*hlt*	HALT					Termina

Consideriamo ad esempio il seguente programma, che legge due numeri interi n, m da nastro (con $n < m$), e che scrive tutti i numeri pari compresi tra n ed m (ad esempio, se $n = 2$ ed $m = 7$ scrive i numeri 2, 4, 6, mentre se $n = 4$ ed $n = 10$ allora scrive 4, 6, 8, 10). Per semplicità, il programma è scritto senza ricorrere alla notazione in corsivo per gli operandi. Sempre per semplicità, scriveremo senza apici e pedici, cosí `c(R0)` significherà $c(R_0)$. Supponiamo infine che sul nastro di input siano presenti i numeri $n = 7$ ed $m = 12$.

```
                    COMMENTO
 1      READ 1      legge il numero n e lo mette in R1
 2      READ 2      legge il numero m e lo mette in R2
 3      LOAD 1      copia il contenuto di R1 in R0
 4      DIV =2      lo divide per 2 (test per vedere se e' pari)
 5      MULT =2     e lo moltiplica per 2
 6      SUB 1       se c(R0) non e' zero il numero e' dispari
 7      JZERO scr   salta se il numero e' pari (cioe' c(R0)=0)
 8      LOAD 1      ricopia c(R1) in R0
 9      ADD =1      e se e' dispari gli aggiunge 1
10      STORE 1     riportando il risultato in R1
11 scr  LOAD 1      inizio del ciclo di scrittura
12      WRITE 1     scrive c(R1) sul nastro
13      ADD =2      aggiunge 2 a c(R0)
14      STORE 1     e copia il risultato in R1
15      SUB 2       gli sottrae c(R2)
16      JGTZ fine   salta se c(R1) > c(R2) alla fine
17      JUMP scr    altrimenti rimane nel ciclo di scrittura
18 fine HALT        ferma la macchina
```

Eseguendo il programma passo passo, si trovano nei registri i valori indicati nella descrizione seguente: per quanto riguarda il ciclo di scrittura compreso tra le istruzioni 11 e 17 i valori nei registri dopo la prima esecuzione delle istruzioni sono indicati senza parentesi, quelli della seconda iterazione tra parentesi tonde, quelli della terza tra parentesi quadre.

```
                    R0          R1          R2
 1      READ 1      0           7           0
 2      READ 2      0           7          12
 3      LOAD 0      7           7          12
 4      DIV =2      3           7          12
```

```
 5      MULT =2      6                7                12
 6      SUB 1       -1                7                12
 7      JZERO scr   -1                7                12
 8      LOAD 1       7                7                12
 9      ADD =1       8                7                12
10      STORE 1      8                8                12
11 scr  LOAD 1       8 (10) [12]      8 (10) [12]      12 (12) [12]
12      WRITE 1      8 (10) [12]      8 (10) [12]      12 (12) [12]
13      ADD =2      10 (12) [14]      8 (10) [12]      12 (12) [12]
14      STORE 1     10 (12) [14]     10 (12) [14]      12 (12) [12]
15      SUB 2       -2  (0)  [2]     10 (12) [14]      12 (12) [12]
16      JGTZ fine   -2  (0)  [2]     10 (12) [14]      12 (12) [14]
17      JUMP scr    -2  (0)          10 (12)           12 (12)
18 fine HALT                 [2]              [14]              [14]
```

Notiamo che lo *spazio* occupato, secondo il criterio di costo uniforme, è pari a 3, dato che sono interessati esattamente 3 registri della macchina. Per quanto riguarda il tempo di esecuzione, con i dati scelti ($n = 7$, $m = 12$) le istruzioni da `1` a `10` sono state eseguite una sola volta ciascuna, quelle numerate tra `11` e `16` tre volte ciascuna, la `17` due volte, e la `18` una sola volta. Il tempo di calcolo è allora: $t = 10 \cdot 1 + 6 \cdot 3 + 1 \cdot 2 + 1 \cdot 1 = 31$. L'unità di misura del tempo non viene indicata. Se disponessimo di un personal computer in grado di simulare una macchina RAM, il tempo effettivamente impiegato potrebbe essere facilmente calcolato moltiplicando t per il tempo necessario all'esecuzione di una singola istruzione RAM.

Più difficile è determinare la complessità computazionale per n ed m interi generici. Sia $p = m - n$. Nel caso in cui m ed n siano entrambi pari (il caso peggiore), occorre scrivere esattamente $\frac{p}{2}+1$ interi. Per p molto grande, il tempo di esecuzione è condizionato dal ciclo di scrittura (essendo trascurabile il tempo impiegato dalle prime dieci istruzioni), che viene eseguito $\frac{p}{2}+1$ volte. Piú esattamente, le istruzioni da `11` a `16` vengono eseguite $\frac{p}{2}+1$ volte ciascuna, mentre la `17` una volta meno. Il numero totale di istruzioni eseguite all'interno del ciclo è allora $6 \cdot (\frac{p}{2}+1) + \frac{p}{2} = \frac{7p}{2}+6$, e quindi risulta direttamente proporzionale a p.

10. Avvertenza

Riferimenti bibliografici

[1] A.V. Aho, J.D. Ullman. *Foundations of Computer Science*, W.H. Freeman and Company, New York, (1992).
[2] A.V. Aho, J.E. Hopcroft, J.D. Ullman. *The Design and Analysis of Computer Algorithms*, Bell Telephone Laboratories, Philippines (1974).
[3] A.V. Aho, J.E. Hopcroft, J.D. Ullman. *Data Structures and Algorithms*, Addyson-Wesley, Reading, Massachusetts, (1987).
[4] G.E. Andrews. *Number Theory*, Dover Publications, New York, (1994).
[5] M. Curzio, P. Longobardi, M. Maj. *Lezioni di Algebra*, Liguori Editore, Napoli, (1994).
[6] M. Davis. *Computability and Unsolvability*, Dover Publications, New York, (1982).
[7] M.R. Garey, D.S. Johnson. *Computers and Intractability*, Bell Telephone Laboratories, W.H. Freeman and Company, New York, (1979).
[8] J.E. Hopcroft, J.D. Ullman. *Formal Languages and their Relation to Automata*, Addison-Wesley, Reading, (1969).

[9] Z. Kohavi. *Switching and Finite Automata Theory, Second Edition*, McGraw-Hill, New York, (1978).
[10] D.E. Knuth. *The Art of Computer Programming, Volume I, Third Edition*, Addison-Wesley, New York, (1997).
[11] V.J. LeVeque. *Foundamentals of Number Theory*, Dover Publications, New York, (1996).
[12] G. Pelagatti. *Sistemi di Elaborazione, Architetture Hardware e Software*, McGraw-Hill, Milano, (1990).
[13] M. Somalvico. *Complementi di Programmazione, Volume 1, Teoria della Programmazione* CLUP, Milano, (1981).
[14] D. Tambuchi. *Guida Rapida al* `FORTRAN77`, dispensa, (2001).
[15] D. Tambuchi *Il metodo di eliminazione di Gauss*, dispensa, (2003).
[16] A.S. Tanenbaum. *Structured Computer Organization, Third Edition*, Prentice-Hall, Englewood Cliff, (1990).

Typeset by LaTeX 2ε *under* `LINUX`