

Capitolo 1

Complessità Computazionale

1.1 Introduzione

La complessità computazionale può essere considerata come la disciplina che si occupa dello studio degli algoritmi in relazione alla potenza e alle limitazioni dei calcolatori o dei sistemi di calcolo. Storicamente essa discende dagli studi fatti intorno alla prima metà del XX secolo e relativi allo studio della risolubilità dei problemi. In modo particolare una parte della logica matematica era stata votata allo studio di procedure automatiche per la dimostrazione di teoremi (tra l'altro molti anni prima che fossero anche solo progettati i primi elaboratori elettronici). La dimostrazione dell'esistenza di teoremi non dimostrabili (e quindi di problemi non risolubili), dovuta a Kurt Gödel, portò ad una specie di scissione nell'ambito di tali gruppi. Da una parte la *Teoria della Calcolabilità* si poneva l'obiettivo di studiare la risolubilità dei problemi e di classificare i problemi in base a tale parametro. Si può intuitivamente dire che risolvere un problema significa trovare un algoritmo che fornisca la soluzione a partire da un determinato insieme di valori assegnati ai parametri del problema. La Teoria della Calcolabilità consente quindi di effettuare una prima, approssimativa, classificazione dei problemi:

1. **Problemi decidibili:** risolubili per via algoritmica;
2. **Problemi non decidibili:** non risolubili per via algoritmica.

Nell'ambito dei problemi decidibili ha senso considerare una seconda classificazione che riguarda soprattutto l'efficienza nella loro risoluzione. Tale parametro viene studiato considerando la quantità di risorse (solitamente

tempo e spazio) che un algoritmo richiede per risolvere un determinato problema. Tale misura può quindi essere considerata un indice per quantificare il grado di difficoltà di un problema. In altre parole se tutti gli algoritmi per risolvere un determinato problema hanno una scarsa efficienza (cioè richiedono una quantità di risorse molto elevata) si può ragionevolmente affermare che ciò dipende dalla intrinseca difficoltà del problema.

Questa seconda classificazione, che riguarda solo i problemi decidibili, porterà ad individuare classi di problemi risolvibili in modo efficiente (da algoritmi che richiedono risorse limitate e che sono detti **Problemi trattabili**, e altre classi di problemi che richiedono una quantità irragionevole di risorse, detti **Problemi intrattabili**). La *Teoria della Complessità* si occupa proprio di questi ultimi aspetti che abbiamo richiamato. Possiamo riassumere il discorso fatto finora dicendo che la *Teoria della Calcolabilità* si interessa alla possibilità di risoluzione dei problemi, cioè se esista o meno un algoritmo in grado di risolvere un determinato problema, mentre la *Teoria della Complessità* studia l'efficienza degli algoritmi e altri problemi correlati. Per esempio ammesso che un algoritmo non sia efficiente quanto ciò dipende dal problema che stiamo risolvendo, ed inoltre esistono altri algoritmi più efficienti oppure il problema è così difficile che non è possibile trovarne migliori? In poche parole si tratta di misurare in qualche maniera il grado di difficoltà dei problemi. Un altro degli obiettivi della complessità computazionale è il classificare, nel modo più raffinato possibile, i problemi decidibili, raggruppandoli in classi in base al tipo e alla quantità di risorse necessarie per risolverli. Le risorse che si considerano sono principalmente il *Tempo* (cioè una misura della durata del tempo di calcolo) e lo *Spazio* (cioè la quantità di memoria richiesta dall'algoritmo). Come ultima osservazione si può dire che da un punto di vista pratico tra problemi non decidibili e problemi intrattabili la differenza è poca, infatti se un problema può essere risolto in un tempo non ragionevole è lo stesso che affermare che non può essere risolto. Lo studio della complessità di un algoritmo prescinde inoltre dalla definizione formale di una serie di concetti che sono intuitivamente noti. Per esempio il concetto stesso di problema: possiamo definirlo come una domanda la cui risposta dipende da un insieme di parametri e dal loro valore.

Introduciamo ora alcune definizioni formali riguardanti le funzioni di variabile intera. Sia \mathbb{N} l'insieme dei numeri naturali (interi non negativi), in teoria della complessità si considerano generalmente funzioni definite da \mathbb{N} a valori in \mathbb{N} . Anche quando il risultato di tali funzioni non è un numero intero, come per esempio \sqrt{n} oppure $\log^2 n$, il risultato sarà sempre considerato come un

numero intero. Quindi se $f : \mathbb{N} \rightarrow \mathbb{N}$ allora il valore $f(n)$ è

$$f(n) = \max\{\lceil f(n) \rceil, 0\}$$

dove $\lceil x \rceil$ denota la parte intera superiore del numero reale x cioè

$$\lceil x \rceil = \min \{n \in \mathbb{N} \mid n \geq x\}$$

Siano quindi $f, g : \mathbb{N} \rightarrow \mathbb{N}$, si scrive

$$f(n) = \mathcal{O}(g(n))$$

e si dice che $f(n)$ è \mathcal{O} -grande di $g(n)$ oppure f è dell'ordine di g se esistono due interi positivi c e n_0 tali che

$$f(n) \leq cg(n) \quad \forall n \geq n_0.$$

Formalmente significa che f cresce come g oppure più lentamente. Si scrive

$$f(n) = \Omega(g(n))$$

e si dice che $f(n)$ è Ω -grande di $g(n)$ se $g(n) = \mathcal{O}(f(n))$, ed infine

$$f(n) = \Theta(g(n))$$

e si dice che $f(n)$ è Θ -grande di $g(n)$ se, contemporaneamente, $f(n) = \mathcal{O}(g(n))$ e $f(n) = \Omega(g(n))$, cioè le due funzioni crescono esattamente allo stesso modo. Per esempio è facile verificare che se $p(n)$ è un polinomio in n di grado k risulta $p(n) = \Theta(n^k)$, cioè la crescita di un polinomio dipende essenzialmente dal termine di grado massimo. Vedremo che stabilire una relazione d'ordine tra le funzioni in base alla loro crescita è uno degli aspetti più importanti della Teoria della Complessità. Per esempio se c è un numero intero maggiore di 1 e $p(n)$ è un qualsiasi polinomio in n , allora

$$p(n) = \mathcal{O}(c^n),$$

ma non è vero che

$$p(n) = \Omega(c^n),$$

cioè, ogni polinomio cresce più lentamente di qualsiasi funzione esponenziale. Dalla stessa proprietà discende che

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

e, inoltre

$$\log^k n = \mathcal{O}(n),$$

per ogni potenza k . Si può verificare che se $p_k(n)$ è un polinomio di grado k a coefficienti interi

$$p_k(n) = \sum_{i=0}^k a_i n^i$$

allora

$$p_k(n) = \Theta(n^k).$$

Infatti

$$p_k(n) = \sum_{i=0}^k a_i n^i \leq \sum_{i=0}^k |a_i| n^i \leq \sum_{i=0}^k |a_i| n^k = c n^k$$

avendo posto

$$c = \sum_{i=0}^k |a_i|$$

e per $n \geq 1$. La maggiorazione inversa può essere dimostrata in modo più complesso nel caso generale, ma osseviamo che è immediata se tutti i coefficienti a_i sono interi positivi. Se non lo sono è sempre possibile trovare una costante c sufficientemente grande in modo tale che

$$n_k \leq c \sum_{i=0}^k a_i n^i \tag{1.1}$$

per n abbastanza grande. Per esempio considerando il polinomio

$$p_3(n) = n^3 - 3n^2 + 1$$

essendo

$$p_3(n) = n^2(n - 3) + 1$$

assume valori sicuramente positivi per $n \geq 3$ e scegliendo $c = 100$ la maggiorazione (1.1) è sicuramente verificata per tali valori.

Nella seguente tabella sono riportati i valori assunti da alcune funzioni al crescere di n :

Funzione	Valore approssimato		
n	10	100	1000
$n \log n$	33	664	9966
n^3	1000	1000000	10^9
$10^6 n^8$	10^{14}	10^{22}	10^{30}
2^n	1024	1.27×10^{30}	1.05×10^{301}
$n^{\log n}$	2099	1.93×10^{13}	7.89×10^{29}
$n!$	3628800	10^{158}	4×10^{2567}

Di seguito riportiamo una lista delle funzioni più comunemente incontrate nell'analisi della complessità degli algoritmi. Le funzioni elencate sono in ordine di grandezza crescente.

Notazione	Nome
$\mathcal{O}(1)$	Costante
$\mathcal{O}(\log^* n)$	Logaritmica iterata (cioè $\log(\log n)$)
$\mathcal{O}(\log n)$	Logaritmica
$\mathcal{O}((\log n)^k)$	Polilogaritmica
$\mathcal{O}(kn)$	Sottolineare ($0 < k < 1$)
$\mathcal{O}(n)$	Lineare
$\mathcal{O}(n \log n)$	Sovralineare
$\mathcal{O}(n^2)$	Quadratica
$\mathcal{O}(n^k)$	Polinomiale
$\mathcal{O}(k^n)$	Esponenziale o Geometrica
$\mathcal{O}(n!)$	Fattoriale
$\mathcal{O}(n^n)$	Nessun nome esplicito

1.2 Tipi di problemi

I problemi possono essere suddivisi in tre tipi principali:

1. **Problemi di Decisione:** hanno come possibili risposte solo *SI* o *NO*, cioè riguardano la possibile esistenza o meno di un determinato oggetto (Rispondono a domande del tipo: Esiste un...);
2. **Problemi di Enumerazione:** hanno come risultato un insieme di soluzioni di un problema (Rispondono a domande del tipo: Elencare tutti...);

3. **Problemi di Ricerca:** hanno come risultato una particolare soluzione di un problema (Rispondono a domande del tipo: Trovare il... tale che...).

In generale si tende a ritenere che i problemi di decisione siano i più facili soprattutto però quando il problema ammette come soluzione il valore SI . I problemi di enumerazione sono i più complicati poichè implicano la necessità di trovare tutte le possibili soluzioni, problemi di questo tipo molto spesso possono essere risolti solo elencando tutte le possibili istanze di un problema e cercando tra queste quelle che verificano la proprietà voluta. Anche i problemi di ricerca sono complessi soprattutto quando si deve trovare una soluzione che soddisfi una particolare proprietà di minimo (o di massimo). Prima di dare una definizione formale di Problema Computazionale e di Algoritmo vediamo alcuni esempi.

Calcolo del determinante di una matrice

Come primo esempio consideriamo il calcolo del determinante di una matrice di ordine n . È noto che e $A = [a_{11}]$, matrice di ordine 1, si definisce *determinante di A* il numero

$$\det A = a_{11}.$$

Se la matrice A è quadrata di ordine n allora fissata una qualsiasi riga di A , la i -esima per esempio, allora applicando la cosiddetta *regola di Laplace* il determinante di A è:

$$\det A = \sum_{j=1}^n a_{ij} (-1)^{i+j} \det A_{ij} \quad (1.2)$$

dove A_{ij} è la matrice che si ottiene da A cancellando la i -esima riga e la j -esima colonna.

La regola di Laplace rappresenta sostanzialmente anche un modo per calcolare il determinante. Quindi supponiamo di voler valutare il costo di tale metodo. Considerando che le operazioni coinvolte sono soprattutto aritmetiche e ipotizzando che ogni operazione richieda lo stesso tempo (ipotesi per la verità non reale) possiamo concordare che il tempo richiesto da tale algoritmo Dalla (1.2) è evidente che per calcolare $\det A$ è necessario conoscere gli n determinanti di A_{ij} per poi effettuare n prodotti ed $n - 1$ somme (ipotizziamo che calcolare $(-1)^{i+j}$ non comporta alcuna operazione). Indichiamo

con $f(n)$ il numero di operazioni necessarie per calcolare il determinante di una matrice di ordine n . Quindi, dalla (1.2) è:

$$f(n) = nf(n-1) + 2n - 1$$

da cui, tralasciando gli ultimi due addendi si ha l'uguaglianza approssimata:

$$f(n) \simeq nf(n-1).$$

Tenendo presente che se $n = 2$ il determinante della matrice

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

è

$$\det A = a_{11}a_{22} - a_{12}a_{21}$$

il numero di operazioni necessarie per calcolarlo è 3, quindi

$$f(2) = 3.$$

In definitiva il numero di operazioni necessarie per calcolare il determinante di una matrice di ordine n può essere calcolato (con le opportune semplificazioni fatte) usando la seguente relazione di ricorrenza:

$$\begin{cases} f(n) = nf(n-1) \\ f(2) = 3. \end{cases}$$

Esplicitando tale relazione di ricorrenza si ottiene

$$\begin{aligned} f(n) &= nf(n-1) = n(n-1)f(n-2) = n(n-1)(n-2)f(n-3) \\ &= n(n-1)(n-2) \dots 3f(2) = n(n-1)(n-2) \dots 3 \cdot 3 = \frac{3}{2}n! \end{aligned}$$

Utilizzando la notazione introdotta in precedenza

$$f(n) = \mathcal{O}(n!).$$

Supponiamo di voler risolvere lo stesso problema utilizzando un diverso algoritmo. È noto che, sotto opportune ipotesi, una matrice quadrata A ammette fattorizzazione LU , cioè esistono due matrici L , triangolare inferiore con elementi diagonali uguali a 1, ed U triangolare superiore, tali che:

$$A = LU.$$

Supponendo di conoscere tali matrici il calcolo del determinante è piuttosto immediato, poichè:

$$\det A = \det L \det U = \det U.$$

Infatti il determinante di matrici triangolari è uguale al prodotto degli elementi diagonali quindi $\det L = 1$. Per calcolare $\det U$ sono necessari solo $n - 1$ prodotti.

Il costo computazionale della fattorizzazione è invece $n^3/3$, quindi indicato con $g(n)$ il numero di operazioni complessive richieste da questo secondo algoritmo per calcolare il determinante di A è

$$g(n) = \mathcal{O}(n^3)$$

quindi enormemente inferiore rispetto alla regola di Laplace.

Raggiungibilità dei Grafi

Per descrivere questo secondo esempio è necessario introdurre la definizione di grafo e diversi concetti riguardanti questi ultimi.

Un *grafo* è una coppia $G = (V, E)$ tale che

1. V è un insieme finito di *vertici* o *nodi*;
2. E è un insieme finito di *lati* (termine inglese *edges*), intendendo per lato una coppia di nodi connessi.

Per esempio considerando il grafo $G = (V, E)$ tale che:

$$V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$$

e

$$E = \{[v_1, v_2], [v_1, v_3], [v_1, v_5], [v_1, v_6], [v_2, v_4], [v_3, v_5], [v_4, v_5], [v_4, v_6], [v_5, v_6]\}$$

allora G può essere rappresentato graficamente come in Figura 1.1. Occasionalmente può essere utile considerare i cosiddetti *multigrafi*, cioè grafi con lati ripetuti.

Osserviamo che non essendo le coppie ordinate scrivere il lato $[v_i, v_j]$ oppure $[v_j, v_i]$ è sostanzialmente la stessa cosa.

Se $G = (V, E)$ è un grafo e $q = [v_i, v_j] \in E$, allora si dice che v_i è *adiacente* a v_j (e ovviamente viceversa) e che il lato q è *incidente* su v_i (e v_j). Si definisce

grado di un vertice v il numero di lati incidenti su v . Per esempio nel grafo in Figura 1.1 il grado di v_1 è 4.

Un *cammino in G* è una sequenza di vertici:

$$w = [v_{i_1}, v_{i_2}, \dots, v_{i_k}]$$

con $k \geq 1$, tali che $[v_{i_j}, v_{i_{j+1}}] \in E$, per ogni $j = 1, \dots, k-1$. Non è necessario che tutti i nodi di un cammino siano distinti.

Un cammino si dice *chiuso* se $k > 1$ e $v_{i_k} = v_{i_1}$. Un cammino senza nodi ripetuti si dice *percorso* (termine inglese *path*). Il valore $k-1$ si definisce *lunghezza del cammino*. Un cammino chiuso senza nodi ripetuti si dice *circuito* o *ciclo*. Se in G non ci sono cicli il grafo si dice *aciclico*. Se G è un grafo con n nodi si dice che G ha un *cammino di Hamilton* (o *cammino Hamiltoniano*) se esiste un cammino che tocca tutti i nodi.

Nel caso del grafo rappresentato nella Figura 1.1 si ha che:

- $[v_1, v_2, v_4, v_5, v_1, v_3, v_5]$ è un cammino di lunghezza 6;
- $[v_1, v_6, v_4, v_5, v_6, v_1]$ è un cammino chiuso;
- $[v_1, v_5, v_6, v_4, v_2]$ è un percorso;
- $[v_1, v_6, v_4, v_5, v_6, v_1]$ è un ciclo;
- $[v_3, v_5, v_1, v_6, v_4, v_2]$ è un cammino Hamiltoniano.

Il grafo si dice *diretto* (o *orientato*, o anche *digrafo*) se ai lati è assegnata una direzione. I lati vengono detti *archi*. Quindi se $G = (V, A)$ è un grafo diretto allora A è un insieme di coppie ordinate, cosicchè $A \subseteq V \times V$. Convenzionalmente i lati sono rappresentati da parentesi quadre mentre gli archi sono denotati da parentesi tonde. In un grafo diretto $G = (V, A)$ il *grado di ingresso* (*indegree*) di un nodo $v \in V$ è il numero di archi della forma $(u, v) \in A$, mentre *grado di uscita* (*outdegree*) di un nodo $v \in V$ è il numero di archi della forma $(v, u) \in A$. In modo simile ai grafi indiretti si possono definire i concetti relativi ai cammini. Quindi un *cammino diretto* è una sequenza di vertici non necessariamente distinti:

$$w = (v_{i_1}, v_{i_2}, \dots, v_{i_k})$$

con $k \geq 1$, tali che $(v_{i_j}, v_{i_{j+1}}) \in A$, per ogni $j = 1, \dots, k-1$. Un cammino diretto si dice *chiuso* se $k > 1$ e $v_{i_k} = v_{i_1}$. Un cammino senza nodi ripetuti si

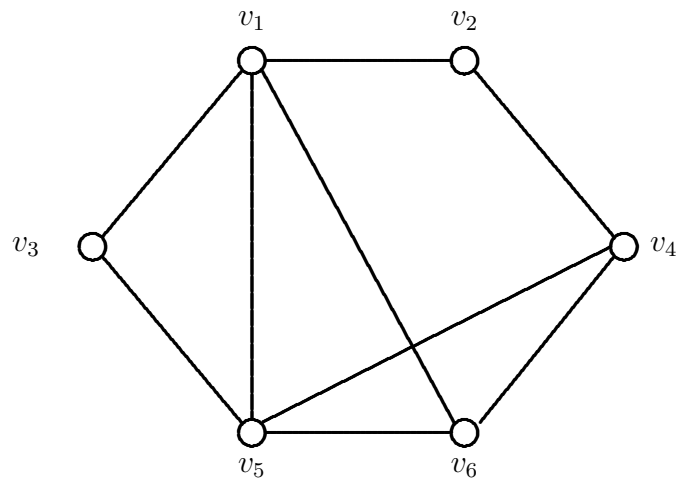


Figura 1.1: Rappresentazione visiva di un grafo.

dice *percorso diretto*. Un cammino diretto chiuso senza nodi ripetuti si dice *circuito diretto* o *ciclo diretto*. Per esempio il grafo diretto $G = (V, A)$ tale che:

$$V = \{v_1, v_2, v_3, v_4, v_5\}$$

e

$$A = \{(v_1, v_2), (v_1, v_4), (v_2, v_3), (v_3, v_5), (v_4, v_1), (v_4, v_5), (v_5, v_2), (v_5, v_4)\}$$

può essere rappresentato come mostrato in Figura 1.2. In tale esempio si ha che:

- $(v_1, v_4, v_5, v_4, v_1, v_2, v_3)$ è un cammino diretto di lunghezza 6;
- $(v_1, v_6, v_4, v_5, v_6, v_1)$ è un cammino chiuso;
- $(v_1, v_4, v_5, v_2, v_3)$ è un percorso diretto;
- (v_2, v_3, v_5, v_2) è un ciclo diretto.

Tipicamente sono numerosi i problemi tipici della teoria dei grafi, per esempio il cosiddetto *Problema della raggiungibilità* (*Graph Reachability*) è il seguente:

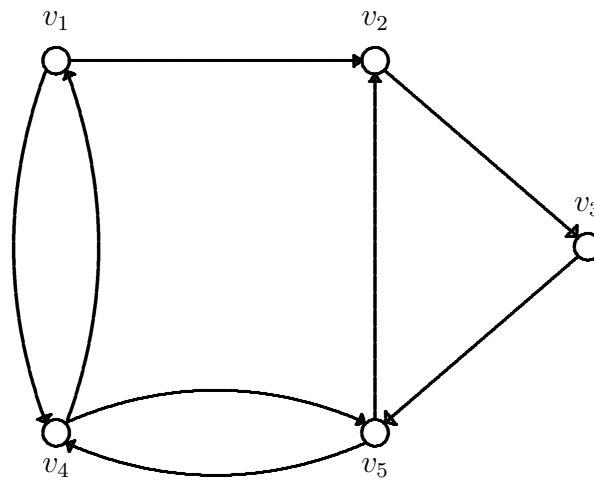


Figura 1.2: Rappresentazione visiva di un grafo diretto.

Assegnati due nodi $i, j \in V$ esiste un percorso che porta da i a j ? Consideriamo come esempio il grafo diretto $G = (V, A)$ dove

$$V = \{1, 2, 3, 4, 5, 6\}$$

e

$$A = \{(1, 2), (1, 3), (2, 5), (3, 4), (4, 6), (5, 4), (6, 2)\}.$$

La rappresentazione visiva del grafo è riportata in Figura 1.3.

In questo specifico caso poniamo il seguente quesito: esiste un cammino diretto dal nodo 1 al nodo 6?

In questo caso è facile accorgersi che esistono due cammini diretti che consentono di raggiungere il nodo 6 partendo dal nodo 1:

$$(1, 3, 4, 6) \quad \text{e} \quad (1, 2, 5, 4, 6)$$

quindi la risposta è SI. Osserviamo anche che invertendo la direzione dell'arco $(4, 6)$ la risposta sarebbe NO. Come visto in precedenza il problema di raggiungibilità dei grafi è un problema di decisione.

Descriviamo ora un semplice algoritmo di ricerca per risolvere tale problema. L'algoritmo richiede che venga memorizzato un sottoinsieme dei nodi denotato con \mathcal{S} , e inizialmente

$$\mathcal{S} = \{1\}$$

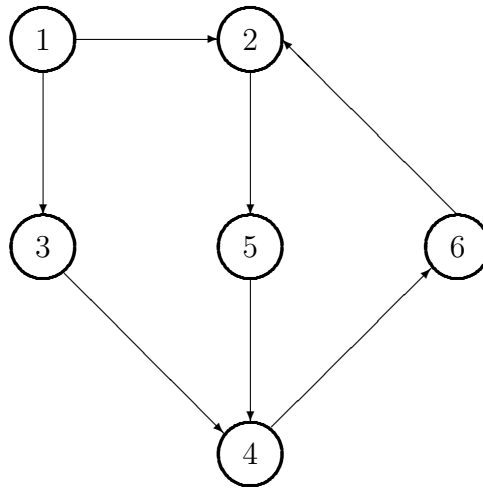


Figura 1.3:

cioè il nodo di partenza. Inoltre ad ogni passo verrà segnato uno o più nodi (cioè saranno inseriti in un secondo insieme \mathcal{M}). Se un nodo i è segnato vuol dire che i fa parte (o ha fatto parte) dell'insieme \mathcal{S} . L'algoritmo procede in questo modo.

Ad ogni iterazione si sceglie un nodo $i \in \mathcal{S}$, che viene rimosso da tale insieme. Quindi si considerano tutti i nodi k tali che $(i, k) \in A$. Se il nodo k non è segnato (cioè $k \notin \mathcal{M}$) allora viene segnato (cioè aggiunto all'insieme \mathcal{M}) e inserito in \mathcal{S} altrimenti non si fa niente e si sceglie con un altro nodo. Questo processo continua finché l'insieme \mathcal{S} non diviene vuoto. Appena termina l'algoritmo si risponde SI al problema di raggiungibilità se il nodo n è stato segnato (cioè $n \in \mathcal{M}$), altrimenti si risponde NO.

Applicando l'algoritmo appena descritto al grafo in Figura 1.3 possiamo

analizzare come funziona praticamente.

Iterazione	Nodo i scelto	Nodi $(i, k) \in A$	Insieme \mathcal{S}	Insieme \mathcal{M}
0	===	===	{1}	{1}
1	$i = 1$	$k = 2, 3$	{2, 3}	{1, 2, 3}
2	$i = 2$	$k = 5$	{3, 5}	{1, 2, 3, 5}
3	$i = 3$	$k = 4$	{5, 4}	{1, 2, 3, 5, 4}
4	$i = 5$	Nessuno	{4}	{1, 2, 3, 5, 4}
5	$i = 4$	$k = 6$	{6}	{1, 2, 3, 5, 4, 6}
6	$i = 6$	Nessuno	\emptyset	{1, 2, 3, 5, 4, 6}
7	STOP			

Il primo aspetto da prendere in considerazione per la valutazione dell'algoritmo è che esso sicuramente termina perchè ad ogni passo viene tolto un elemento dall'insieme \mathcal{S} e non sempre ne viene inserito uno, quindi prima o poi l'insieme sarà vuoto. Inoltre l'algoritmo risolve certamente il problema. Infatti esso fornisce una risposta positiva all'esistenza del cammino se il nodo finale fa parte dell'insieme dei nodi segnati. Questa eventualità è legata al fatto che in questo insieme vengono inseriti tutti i nodi che possono essere raggiunti partendo dal nodo 1. Un altro aspetto è che si potrebbe porre come condizione di stop l'inserimento del nodo terminale nell'insieme \mathcal{M} , ma tale condizione sarebbe effettivamente valida solo se il problema decisionale ha risposta affermativa. Un aspetto secondario, ma comunque importante, è legato al fatto che un algoritmo deve prescindere dalla codifica delle istanze. In questo caso solitamente il grafo viene rappresentato dalla cosiddetta *matrice di adiacenza* (*Adjacency Matrix*). Tale matrice ha il numero di righe e di colonne uguali al numero di nodi e il suo elemento in posizione (i, j) vale 1 se l'arco (i, j) appartiene all'insieme A , 0 in caso contrario. Nel caso dell'esempio considerato la matrice di adiacenza è la seguente:

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Un altro problema dell'algoritmo è il seguente: come scegliere il nodo $i \in \mathcal{S}$ tra tutti gli elementi dell'insieme?

La strategia di scelta del nodo può influenzare ovviamente il comportamento dell'algoritmo. Si può scegliere per esempio una delle seguenti:

1. Gestire l'insieme \mathcal{S} come se fosse una coda, cioè scegliendo il nodo che appartiene all'insieme da più tempo;
2. Gestire l'insieme \mathcal{S} come uno stack, cioè scegliendo l'ultimo nodo che è stato inserito.

Indipendentemente dal modo di gestire \mathcal{S} notiamo che ogni elemento della matrice di adiacenza viene controllato (al più) una sola volta, quando il nodo corrispondente a tale riga viene scelto. Quindi considerando che il costo dell'algoritmo dipende essenzialmente dal numero di confronti di elementi della matrice di adiacenza si può ragionevolmente affermare che avremo un numero massimo di n^2 confronti (infatti il numero massimo di archi è proprio n^2).

Assumendo che le altre operazioni richieste (scelta di un elemento dell'insieme \mathcal{S} , controllo se un nodo è segnato ed eventuale segno del nodo) possono essere effettuati in un tempo costante allora concludiamo che l'algoritmo di ricerca che effettua il controllo sull'esistenza di un cammino diretto congiungente due nodi di un grafo diretto richiede un tempo proporzionale ad n^2 cioè ha un costo $\mathcal{O}(n^2)$.

Un secondo aspetto che si deve tenere presente nell'algoritmo di ricerca descritto è la quantità di memoria richiesta. L'algoritmo infatti richiede di memorizzare l'insieme S e l'insieme dei nodi segnati. Poiché ci possono essere al più n nodi segnati e la cardinalità massima di S può essere al più n , l'algoritmo richiede $\mathcal{O}(n)$ spazio.

Il Problema del Commesso Viaggiatore

Il Problema del Commesso Viaggiatore (in breve TSP, Traveling Salesman Problem) può essere formulato nel seguente modo: supponiamo di avere n città C_1, C_2, \dots, C_n , delle quali è nota la cosiddetta matrice delle distanze $D = \{d_{ij}\}$, dove d_{ij} indica la distanza tra le città C_i e C_j . La matrice D è simmetrica (la distanza della città C_i da C_j è la stessa di C_j da C_i) e con elementi diagonali nulli (la distanza di ogni città da se stessa è uguale a zero). Un percorso del commesso viaggiatore è un opportuno ordinamento delle città

$$C_{\pi_1} C_{\pi_2} \dots C_{\pi_{n-1}} C_{\pi_n}$$

dove $\Pi = (\pi_1, \pi_2, \dots, \pi_{n-1}, \pi_n)$ indica una permutazione dell'insieme di numeri interi $\{1, 2, \dots, n-1, n\}$. Il problema del commesso viaggiatore è quello di individuare un percorso che minimizzi la somma delle distanze dalle città, cioè

$$\min \sum_{i=1}^n d_{\pi_i, \pi_{i+1}}$$

richiedendo che risulti $\pi_{n+1} = \pi_1$ se si vuole che il commesso viaggiatore termini il percorso da dove è partito. Un problema di questo tipo è ovviamente un problema di ricerca in quanto si vuole determinare un tipo di percorso ottimale. Si potrebbe anche modificare il problema in uno di decisione, chiedendo di verificare se esiste un percorso di lunghezza minore o uguale ad un valore prefissato B , cioè tale che

$$\sum_{i=1}^n d_{\pi_i, \pi_{i+1}} \leq B.$$

Si può pensare, per esempio, che i valori della matrice D non indichino le distanze tra le città ma i costi per viaggio e che il valore B sia un limite superiore per il budget che il commesso viaggiatore ha a disposizione, pertanto il problema decisionale consiste nel sapere se tale budget sia sufficiente per visitare tutte le città. Per il problema del commesso viaggiatore non si conoscono (e presumibilmente non esistono) algoritmi di tipo polinomiale. Un algoritmo per risolverlo è il cosiddetto *metodo della Ricerca Esaustiva*, cioè un algoritmo che enumera tutti i possibili percorsi per poi fornire come soluzione quello che ha la lunghezza minore. Il numero di possibili percorsi è pari a $n!$, numero che è possibile ridurre a $(n-1)!$ fissando la città di partenza (e di arrivo) e dimezzando in considerazione del fatto che ogni percorso può essere fatto anche in modo inverso percorrendo esattamente la stessa distanza complessiva, quindi abbiamo $(n-1)!/2$ percorsi e quindi il costo dell'algoritmo di Ricerca Esaustiva è $\mathcal{O}((n-1)!)$. Un algoritmo molto molto costoso. In realtà questo è un problema che è considerato tra i più difficili da risolvere efficientemente, infatti non si conoscono algoritmi con complessità polinomiale in grado di risolverlo (e probabilmente non ne esistono).

Un modo di affrontare il problema del commesso viaggiatore è quello di rappresentarlo sotto forma di grafo cosiddetto *pesato*, in cui i nodi rappresentano le città, e sono tutti connessi, i lati i percorsi tra due città cui è associato un peso (cioè la distanza o il costo). Ovviamente il grafo non è diretto perchè

non c'è un verso nella distanza tra due città. Il problema è quello di trovare il ciclo Hamiltoniano con il minimo peso complessivo.

1.3 Problemi e algoritmi

1.3.1 Definizione formale di problema

Un problema computazionale può essere formalmente definito come un quesito di carattere generale al quale si deve rispondere e che dipende da vari parametri, o variabili libere, i cui valori non sono specificati.

Per descrivere in modo non ambiguo un problema computazionale è necessario fornire le specifiche di tutti i suoi parametri e delle proprietà che devono essere soddisfatte dalla soluzione.

Si definisce inoltre *istanza di un problema* un particolare insieme di valori assunti dalle variabili libere. Si definisce *dimensione di un'istanza* il numero di locazioni di memoria necessarie per memorizzare tutti i dati di un'istanza.

1.3.2 Definizione formale di algoritmo

In modo formale un algoritmo può essere definito come una procedura generale per risolvere un problema computazionale. Tale procedura è descritta sotto forma di sequenza finita e non ambigua di passi computazionali.

Un algoritmo è *corretto* se per ogni istanza di un problema computazionale produce la soluzione corrispondente, cioè risolve il problema.

Un algoritmo non corretto può non terminare in corrispondenza di determinate istanze, oppure terminare fornendo una soluzione sbagliata.

L'algoritmo può essere visto come una funzione che associa ad un'istanza la soluzione del problema stesso, infatti spesso si usa il termine equivalente di *funzione algoritmica*:

$$\text{Algoritmo} : \{\mathcal{I}\text{stanze}\} \longrightarrow \{\mathcal{S}\text{oluzioni}\}.$$

Un algoritmo deve avere le seguenti proprietà:

1. L'insieme delle istruzioni che definisce l'algoritmo è finito.
2. L'insieme delle informazioni che rappresentano il problema e i requisiti richiesti alla sua soluzione hanno una descrizione finita.

3. Esiste un agente di calcolo in grado di eseguire le istruzioni.
4. Il procedimento di calcolo, o *computazione*, è suddiviso in passi discreti e non fa uso di dispositivi analogici.
5. La computazione è deterministica, cioè la sequenza dei passi computazionali è determinata senza alcuna ambiguità.

In base alle proprietà appena esposte la soluzione associata a un determinato insieme di parametri è sempre la stessa, cioè un algoritmo fornisce lo stesso risultato ogni volta che riceve in ingresso gli stessi dati. In generale, dato un problema, si è interessati a trovare l'algoritmo di risoluzione più efficiente rispetto all'uso delle risorse di calcolo. Il primo problema che si pone è quello di scegliere la risorsa della quale si vuole che gli algoritmi facciano il migliore uso possibile. Le risorse devono esprimere consumi della macchina ed il loro utilizzo economico deve portare dei vantaggi rilevanti. Molte sono le scelte possibili, tuttavia quelle classiche, che abbiamo già accennato, sono il *tempo* e lo *spazio*.

I vantaggi derivati economizzando memoria e durata di calcolo sono evidenti: usare al meglio la memoria significa essere in grado di rappresentare problemi di dimensioni sempre maggiori e soprattutto di memorizzare grandi quantità di dati intermedi, necessari al fine di procedere nelle elaborazioni; completare computazioni nel minor tempo possibile è cruciale per molte applicazioni e rende possibile la soluzione di problemi di dimensioni sempre maggiori.

Come abbiamo visto negli esempi dei precedenti paragrafi il costo computazionale di un algoritmo si esprime normalmente in funzione di un'unica variabile, che rappresenta la *dimensione* del problema. Abbiamo detto che un problema dipende da parametri che, una volta specificati, ne definiscono un'istanza.

Nell'analisi degli algoritmi attraverso i diversi modelli di calcolo che vedremo nei paragrafi seguenti vengono fatte sempre alcune semplificazioni che sono tuttavia molto realistiche. Si suppone che l'algoritmo riceve in ingresso k valori interi e restituisce come risultato j valori interi, con $k, j \geq 1$. L'algoritmo viene visto come una funzione

$$f : \mathbb{N}^k \rightarrow \mathbb{N}^j, \quad (y_1, y_2, \dots, y_j) = f(x_1, x_2, \dots, x_k).$$

La necessità di introdurre determinati modelli di calcolo sta nel fatto di dover valutare la complessità di algoritmi molto diversi tra loro usando una

descrizione basata su una codifica delle istruzioni che segue un determinato insieme di regole.

1.4 Modelli di Calcolo

Ricordiamo brevemente i modelli di calcolo più noti che sono stati introdotti successivamente rinviando una trattazione più complessa che esula dagli scopi di questo corso:

1. Le Funzioni Ricorsive (J. Herbrand, K. Gödel, anni '20-'30);
2. Le Macchine di Turing (A. Turing, 1936)
3. λ -calcolo (A. Church, S. Kleene, 1936)
4. I sistemi di Post (1943)
5. Le catene di Markov (1954)
6. Le Macchine ad Accesso Casuale (RAM) (Shepherdson, Sturgis, 1963);
7. Le Macchine a Registri Elementari (M. Minsky, anni '70).

Tutte queste caratterizzazioni, pur essendo molto diverse nella formulazione, condividono alcuni elementi comuni:

- Tentano di fornire una formale definizione di algoritmo (o di funzione algoritmica);
- Formulano la nozione di funzione parziale calcolabile da un algoritmo.

La definizione formale di algoritmo dipende strettamente dal tipo di modello utilizzato e dal modo con cui sono organizzati i passi elementari del calcolo che costituiscono la computazione. Per quello che riguarda i modelli appena citati va detto che alcuni forniscono una rappresentazione degli algoritmi fortemente mutuata da concetti matematici, altri sono delle astrazioni più o meno simili alla realtà, degli elaboratori elettronici. Nei seguenti paragrafi accenniamo brevemente ad alcuni modelli di calcolo interessanti soprattutto per motivi di carattere storico e culturale.

1.4.1 Le Funzioni Ricorsive Primitive

Visto che un algoritmo può essere interpretato come una funzione il primo tentativo fatto nel passato di rappresentare univocamente tutti gli algoritmi mediante un modello di calcolo è stato quello di utilizzare un modello di carattere matematico, quello delle Funzioni Ricorsive Primitive.

Definizione 1.4.1 *L'insieme \mathcal{C} delle Funzioni Ricorsive Primitive è la più piccola classe di funzioni tali che*

1. le funzioni costanti appartengono a \mathcal{C} ;
2. la funzione successore $S(x) = x + 1$ appartiene a \mathcal{C} ;
3. le funzioni identità, cioè:

$$I_j(x_1, x_2, \dots, x_n) = x_j, \quad j = 1, \dots, n,$$

appartengono a \mathcal{C} ;

4. se f è una funzione di k variabili e g_1, \dots, g_k sono funzioni in m variabili in \mathcal{C} allora la funzione;

$$h(x_1, x_2, \dots, x_m) = f(g_1(x_1, x_2, \dots, x_m), \dots, g_k(x_1, x_2, \dots, x_m))$$

appartiene a \mathcal{C} (**Operazione di Composizione**);

5. se g ed h sono funzioni in \mathcal{C} rispettivamente in $k - 1$ e $k + 1$ variabili allora l'unica funzione f in k variabili per cui

$$f(0, x_2, \dots, x_k) = g(x_2, \dots, x_k)$$

$$f(y + 1, x_2, \dots, x_k) = h(y, f(y, x_2, \dots, x_k), x_2, \dots, x_k)$$

appartiene a \mathcal{C} (**Operazione di Ricorsione Primitiva**);

Visto che sia le funzioni di base che gli operatori sono definiti in modo costruttivo, la definizione della classe delle funzioni ricorsive primitive indica come tali funzioni possono essere calcolate. In particolare, una funzione f è ricorsiva primitiva se e solo esiste una sequenza finita di funzioni f_1, f_2, \dots, f_n tali che $f_n = f$ e, per ogni $j \leq n$:

- $f_j \in \mathcal{C}$ per una tra le proprietà 1., 2. o 3., oppure

• f_j si può ricavare direttamente dalle funzioni f_i , $i < j$, per mezzo delle operazioni di composizione o ricorsione primitiva.

Una descrizione di tale sequenza, insieme alla specifica di come sia possibile ottenere ciascuna f_i , $i \leq n$, è detta *derivazione di f* .

Esempio 1.4.1 *Derivazione della somma come funzione ricorsiva primitiva. Definiamo la seguente sequenza*

$$\begin{aligned} f_1(x) &= x \\ f_2(x) &= x + 1 \\ f_3(x_1, x_2, x_3) &= x_2 \\ f_4(x_1, x_2, x_3) &= f_2(f_3(x_1, x_2, x_3)) \\ \begin{cases} f_5(0, x_2) &= f_1(x_2) \\ f_5(y + 1, x_2) &= f_4(y, f_5(y, x_2), x_2) \end{cases} \end{aligned}$$

Giusto per esempio verifichiamo che $f_5(2, 3) = 2 + 3 = 5$:

$$\begin{aligned} f(2, 3) &= f_5(2, 3) = f_4(1, f_5(1, 3), 3) = f_4(1, f_4(0, f_5(0, 3), 3), 3) = \\ &= f_4(1, f_4(0, f_1(3), 3), 3) = f_4(1, f_4(0, 3, 3), 3) = \\ &= f_4(1, f_2(f_3(0, 3, 3)), 3) = f_4(1, f_2(3), 3) = f_4(1, 4, 3) = \\ &= f_2(f_3(1, 4, 3)) = f_2(4) = 5. \end{aligned}$$

Come esempio di funzione che non appartiene alla classe delle funzioni ricorsive primitive viene citata la cosiddetta *Funzione di Ackermann*:

$$\begin{aligned} A(0, x, y) &= x + 1 \\ A(1, x, y) &= x + y \\ A(2, x, y) &= xy \\ n \geq 2 : A(n + 1, x, y) &= \begin{cases} 1 & y = 0 \\ A(n, x, A(n + 1, x, y - 1)) & y > 0 \end{cases} \end{aligned}$$

Avendo dimostrato che l'insieme delle funzioni ricorsive primitive non è in grado di definire tutte le funzioni e quindi non può essere utilizzato per formalizzare il concetto di funzione algoritmica, intorno agli anni '30 si rese necessario estendere tali concetti e cambiando il modello di calcolo.

1.4.2 La Macchine di Turing

In maniera del tutto informale possiamo dire che una *Macchina di Turing* è un dispositivo composto da:

1. un'unità di controllo;
2. un dispositivo mobile di lettura e scrittura detto *testina*;
3. un *nastro* quale dispositivo di Input/Output.

Sebbene la macchina di Turing sia stata teorizzata nel 1936 essa rappresenta in un modo modo fedele i primi calcolatori elettronici che sono stati progettati e realizzati diversi anni dopo. Il nastro della macchina è finito a sinistra e infinito a destra ed è diviso in celle ognuna delle quali può contenere un simbolo appartenente ad un determinato alfabeto. È possibile definire macchine di Turing a k nastri, con $k \geq 3$, in cui il primo nastro contiene l'input, l'ultimo contiene l'output della macchina e tutti gli altri sono nastri di lavoro. Un passo computazionale di una macchina di Turing consiste nelle seguenti operazioni:

- L'unità di controllo modifica il proprio stato in funzione dello stato attuale e del simbolo letto;
- La testina può scrivere un simbolo sul nastro e poi spostarsi a destra, a sinistra oppure restare ferma.

In modo formale una macchina di Turing è una quadrupla

$$M = (K, \Sigma, \delta, s)$$

dove K è un insieme finito di stati, $s \in K$ è lo stato iniziale, Σ è un insieme finito di simboli che prendono il nome di *alfabeto di M* , e δ è la cosiddetta *funzione di transizione*. Si assume che gli insiemi Σ e K siano disgiunti. L'alfabeto Σ contiene sempre due simboli speciali:

- che indica un blank, una cella vuota;
- ▷ che indica l'inizio del nastro.

La funzione δ è definita:

$$\delta : K \times \Sigma \longrightarrow K \cup \{h, yes, no\} \times \Sigma \times \{\rightarrow, \leftarrow, -\}$$

dove h è lo stato di halt, yes è lo stato di accettazione, no è lo stato di rifiuto. L'ultimo valore della funzione δ indica la direzione in cui spostare la testina:

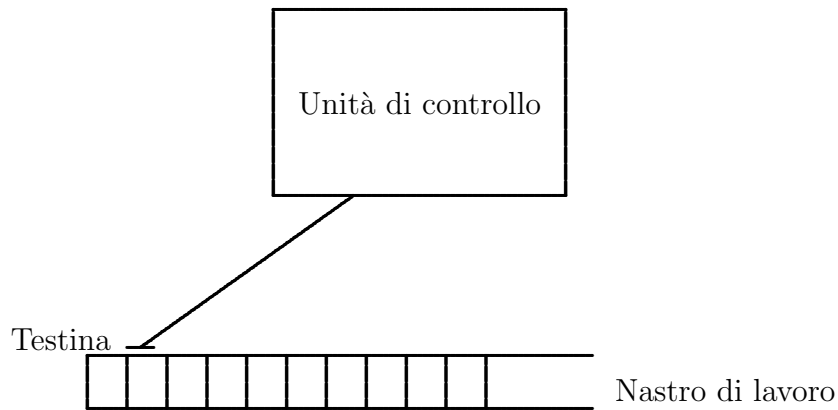


Figura 1.4: Una Macchina di Turing a un nastro.

- ← La testina si sposta sulla cella a sinistra;
- La testina si sposta sulla cella a destra;
- La testina resta ferma.

La funzione δ è il programma della macchina: essa specifica, per ogni possibile combinazione dello stato corrente $q \in K$ con il simbolo inserito nella cella puntata dalla testina $\sigma \in \Sigma$ una tripla

$$\delta(q, \sigma) = (p, \alpha, D)$$

dove

- p e' lo stato successivo;
- α è il simbolo che viene scritto nella cella;
- $D \in \{\rightarrow, \leftarrow, -\}$ è la direzione in cui la testina deve spostarsi.

È bene specificare che spesso non tutte le transizioni sono possibili, infatti nella definizione della funzione δ si trascrivono solo quelle che possono avvenire. La situazione della macchina di Turing prima dell'inizio della computazione è la seguente: la testina punta al simbolo iniziale del nastro \triangleright , lo stato dell'unità di controllo è s e a destra della testina si trova la stringa di input

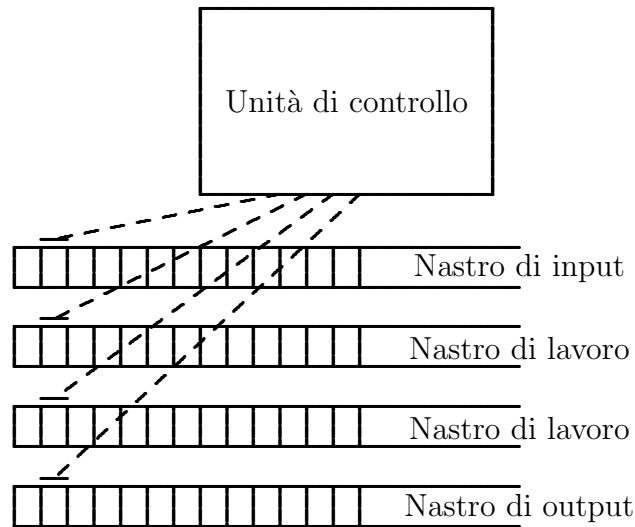


Figura 1.5: Esempio di Macchina di Turing a 4 nastri.

$x \in (\Sigma - \{\sqcup, \triangleright\})^*$. In base alla definizione della funzione δ e partendo dalla configurazione iniziale, si cambia lo stato, si scrive un simbolo sul nastro e si muove la testina. Il primo passo definito dalla funzione di transizione deve essere tale da non cancellare il simbolo di inizio del nastro e permettere alla testina di muoversi:

$$\delta(s, \triangleright) = (p, \triangleright, \rightarrow), \quad p \in K$$

in cui eventualmente p può coincidere con s . Benchè la testina non si possa muovere verso sinistra essa può spostarsi indefinitamente verso destra, si infatti suppone che a destra della stringa x di input ci siano infiniti \sqcup . Nelle computazioni può capitare che la stringa di input diventi più lunga. Poichè la funzione di transizione è completamente specificata la macchina di Turing si ferma quando viene raggiunto uno dei tre stati di stop:

- h La computazione si ferma
- yes La computazione si ferma e la stringa di input viene accettata
- no La computazione si ferma e la stringa di input viene rifiutata.

Se una macchina di Turing raggiunge uno stato di *alt* per un input x allora l'output si indica con $M(x)$. Il valore di $M(x)$ può essere

$$M(x) = \begin{cases} \textit{yes} & \text{La stringa } x \text{ è stata accettata dalla macchina;} \\ \textit{no} & \text{La stringa } x \text{ è stata rifiutata dalla macchina;} \\ y & \text{Risultato della computazione per l'input } x. \end{cases}$$

In quest'ultimo caso, cioè se

$$y = M(x)$$

la situazione finale della macchina prevede che il nastro contenga all'inizio il simbolo \triangleright seguito dalla stringa y e da infiniti blank \sqcup . La posizione della testina è indifferente. Naturalmente è possibile che la macchina di Turing non raggiunga uno stato di *alt* per una stringa di input x , in questo caso si scrive

$$M(x) = \nearrow$$

cioè la macchina diverge. Far divergere una macchina di Turing è molto semplice, infatti è sufficiente che nella tabella di definizione della funzione di transizione ci sia:

$$\delta(p, \alpha) = (p, \alpha, -)$$

cosicchè lo stato resta sempre invariato e la testina ferma.

Vediamo ora di descrivere un esempio di semplice macchina di Turing. Sia

$$M = (K, \Sigma, \delta, s)$$

dove

$$K = \{s, q\}, \quad \Sigma = \{0, 1, \sqcup, \triangleright\}$$

e la cui funzione di transizione è fornita dalla seguente tabella:

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

Tale macchina di Turing prende in input una stringa di 0 e 1 e fornisce in output una stringa in cui i simboli sono stati invertiti (se la stringa di input è $x = 01001$ allora quella di output è $y = 10110$). Per avere conferma di questo è sufficiente simulare la macchina a partire dalla stringa di input, ottenendo i seguenti passaggi:

Passo	Stato	Nastro	
1	s	$\triangleright 01001 \sqcup$	
2	s	$\triangleright 01001 \sqcup$	
3	s	$\triangleright 01001 \sqcup$	
4	s	$\triangleright 01001 \sqcup$	
5	s	$\triangleright 01001 \sqcup$	
6	s	$\triangleright 01001 \sqcup$	
7	s	$\triangleright 01001 \sqcup$	(1.3)
8	q	$\triangleright 01001 \sqcup$	
9	q	$\triangleright 01000 \sqcup$	
10	q	$\triangleright 01010 \sqcup$	
11	q	$\triangleright 01110 \sqcup$	
12	q	$\triangleright 00110 \sqcup$	
13	q	$\triangleright 10110 \sqcup$	
14	h	$\triangleright 10110 \sqcup$	

Il funzionamento della macchina merita un certo approfondimento, innanzitutto osserviamo che la testina scorre prima tutta la stringa di input, e questo è evidenziato dal fatto che la macchina permane nello stato iniziale s . Una volta che viene incontrato il simbolo \sqcup questo indica che la stringa di input è terminata e infatti la macchina passa nello stato q . In questo stato i simboli 0 e 1 della stringa vengono cambiati finchè non viene incontrato il simbolo di inizio nastro \triangleright , cosicchè la macchina interpreta che la stringa di input è terminata e quindi passa nello stato di halt h . In questo caso la macchina M calcola una stringa di output, che è quella che si trova a destra del simbolo \triangleright alla fine della computazione. Quindi in questo esempio:

$$M(01001) = 10110.$$

Un altro tipo di macchina di Turing è quella che, anzichè calcolare un output, verifica se una stringa possiede una determinata proprietà. L'esempio che consideriamo è quello della macchina di Turing che riconosce le stringhe palindrome (cioè quelle che sono uguali se lette indifferentemente da sinistra a destra o viceversa), composte dai soli simboli 0 e 1.

stato $p \in K$	simbolo $\varepsilon \in \Sigma$	$\delta(p, \varepsilon)$
s	0	$(q_0, \triangleright, \rightarrow)$
s	1	$(q_1, \triangleright, \rightarrow)$
s	\triangleright	$(s, \triangleright, \rightarrow)$
s	\sqcup	$(yes, \sqcup, -)$
q_0	0	$(q_0, 0, \rightarrow)$
q_0	1	$(q_0, 1, \rightarrow)$
q_0	\sqcup	$(q'_0, \sqcup, \leftarrow)$
q_1	0	$(q_1, 0, \rightarrow)$
q_1	1	$(q_1, 1, \rightarrow)$
q_1	\sqcup	$(q'_1, \sqcup, \leftarrow)$
q'_0	0	(q, \sqcup, \leftarrow)
q'_0	1	$(no, 1, -)$
q'_0	\triangleright	$(yes, \sqcup, \rightarrow)$
q'_1	0	$(no, 1, -)$
q'_1	1	(q, \sqcup, \leftarrow)
q'_1	\triangleright	$(yes, \triangleright, \rightarrow)$
q	0	$(q, 0, \leftarrow)$
q	1	$(q, 1, \leftarrow)$
q	\triangleright	$(s, \triangleright, \rightarrow)$

Anche in questo caso prima di vedere un esempio dobbiamo fare alcune considerazioni sulla strategia di funzionamento della macchina. Infatti quando la macchina di Turing si trova nello stato s trova il primo simbolo della stringa di input, lo trasforma in \triangleright (cosicché rende più corta la stringa), e lo memorizza nello stato, infatti la macchina passa nello stato q_0 se il simbolo trovato è 0, in q_1 se invece è 1. Ora la macchina si muove fino alla fine della stringa, cioè finché non incontra il simbolo \sqcup e quindi si sposta a sinistra di una cella, quindi passa nello stato simbolo q'_0 oppure q'_1 sempre avendo in memoria il primo simbolo. Se l'ultimo simbolo della stringa coincide con quello memorizzato allora viene sostituito con \sqcup cosicché la stringa si accorcia ulteriormente e la macchina riparte andando a ritroso verso l'inizio della stringa ripetendo le stesse operazioni. Se invece l'ultimo simbolo della stringa è diverso dal primo allora la macchina termina la computazione nello stato no concludendo che la stringa non è palindroma. Se la macchina arriva alla stringa vuota, oppure non riesce a trovare l'ultimo simbolo, cosa che si verifica se la stringa è composta da un singolo simbolo, allora la computazione termina nello stato yes e la stringa viene riconosciuta come palindroma. Ve-

diamo il funzionamento di tale macchina quando applicata alla stringa di input $x = 1001$:

Passo	Stato	Nastro
1	s	$\triangleright 1001 \sqcup$
2	s	$\triangleright \underline{1}001 \sqcup$
3	q_1	$\triangleright \triangleright \underline{0}01 \sqcup$
4	q_1	$\triangleright \triangleright \underline{0}01 \sqcup$
5	q_1	$\triangleright \triangleright 00\underline{1} \sqcup$
6	q_1	$\triangleright \triangleright 001\underline{\sqcup} \sqcup$
7	q'_1	$\triangleright \triangleright 001 \sqcup$
8	q	$\triangleright \triangleright \underline{00} \sqcup \sqcup$
9	q	$\triangleright \triangleright \underline{00} \sqcup \sqcup$
10	q	$\triangleright \triangleright \underline{00} \sqcup \sqcup$
11	s	$\triangleright \triangleright \underline{00} \sqcup \sqcup$
12	q_0	$\triangleright \triangleright \triangleright \underline{0} \sqcup \sqcup$
13	q_0	$\triangleright \triangleright \triangleright 0 \underline{\sqcup} \sqcup$
14	q'_0	$\triangleright \triangleright \triangleright \underline{0} \sqcup \sqcup$
15	q	$\triangleright \triangleright \triangleright \underline{\sqcup} \sqcup \sqcup$
16	s	$\triangleright \triangleright \triangleright \underline{\sqcup} \sqcup \sqcup$
17	yes	$\triangleright \triangleright \triangleright \underline{\sqcup} \sqcup \sqcup$

Macchine di Turing a k nastri

Vediamo ora cosa cambia quando la macchina di Turing ha k nastri. Essa continua a essere definita come una quadrupla solo che la funzione di transizione δ deve essere definita in funzione di uno stato (che resta unico perchè l'unità di controllo resta una) e dei k simboli letti dalle k testine sui k nastri. L'alfabeto Σ è unico per tutti i nastri. La funzione δ deve avere come uscita lo stato dell'unità di controllo e in più k simboli, ognuno dei quali viene trascritto su un nastro, e k movimenti della testina, poichè ogni testina si può muovere su un nastro indipendentemente dalle altre. Quindi

$$\delta : K \times \Sigma^k \longrightarrow K \cup \{h, yes, no\} \times (\Sigma \times \{\leftarrow, \rightarrow, -\})^k .$$

1.4.3 Complessità delle Macchine di Turing

Complessità in Tempo

È possibile definire. Introduciamo ora in questa sezione le nozioni che ci permettono di definire la complessità di una macchina di Turing. Si può definire una singola operazione di una Macchina di Turing introducendo il concetto di *configurazione*.

Definizione 1.4.2 Una configurazione della macchina M è una tripla (q, w, u) dove $q \in K$ è uno stato mentre u e w sono stringhe in Σ^* .

In dettaglio w è la stringa a sinistra della testina compreso il simbolo evidenziato dalla testina e u è la stringa a destra della testina, anche il solo carattere vuoto mentre q è lo stato corrente. Vedendo l'esempio della macchina descritta in (1.3), al passo 9 la macchina M ha la configurazione

$$(q, \triangleright 0100, 0\sqcup).$$

Definizione 1.4.3 Se M è una macchina di Turing si dice che la configurazione (q, w, u) produce la configurazione (q', w', u') in un passo, e si scrive

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

se in un singola computazione della macchina si passa dallo stato (q, w, u) a quello (q', w', u') .

Se σ è l'ultimo simbolo di w e $\delta(q, \sigma) = (p, \gamma, D)$ allora deve essere necessariamente $p = q'$.

Se $D = \rightarrow$ allora w' è w con l'ultimo simbolo, cioè σ , sostituito da γ e con il primo simbolo di u aggiunto alla fine, invece u' è u privata del primo simbolo a sinistra, se u è la stringa vuota allora anche u' è la stringa vuota.

Se $D = \leftarrow$ allora w' è w ma senza il simbolo σ alla sua fine, invece u' è u con l'aggiunta di γ alla sua sinistra.

Se $D = -$ allora w' è w con il simbolo σ sostituito da γ e $u' = u$.

Si può generalizzare il concetto dicendo che la configurazione (q, w, u) produce la configurazione (q', w', u') in k passi, e si scrive

$$(q, w, u) \xrightarrow{M^k} (q', w', u'), \quad k \geq 0$$

se esistono $k + 1$ configurazioni (q_i, w_i, u_i) , $i = 1, \dots, k + 1$, tali che

$$(q_i, w_i, u_i) \xrightarrow{M} (q_{i+1}, w_{i+1}, u_{i+1}), \quad i = 1, \dots, k,$$

e, inoltre

$$(q_1, w_1, u_1) = (q, w, u), \quad (q_{k+1}, w_{k+1}, u_{k+1}) = (q', w', u').$$

Infine si dice che la configurazione (q, w, u) produce la configurazione (q', w', u') e si scrive

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

se esiste $k \geq 0$, k intero, tale che

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

Quindi si può dare la seguente definizione di tempo di computazione per una macchina di Turing.

Definizione 1.4.4 *Se per una macchina di Turing M con input x si ha:*

$$(s, \triangleright, x) \xrightarrow{M^t} (H, w, u)$$

con $H \in \{h, \text{yes}, \text{no}\}$ allora il tempo richiesto da M per l'input x è t .

Quindi il tempo richiesto da una macchina può essere considerato semplicemente come il numero di passi richiesto dalla macchina per fermarsi. Se

$$M(x) = \nearrow$$

allora il tempo richiesto da M per l'input x è posto uguale a ∞ . Dobbiamo ricordare tuttavia che le prestazioni di un algoritmo devono essere valutate calcolando il tempo e lo spazio richiesto in funzione della dimensione di un'istanza n . Per le macchine di Turing è naturale considerare tali valori in funzione della lunghezza della stringa di input x . Si dice quindi che la macchina M esegue le operazioni entro un tempo $f(n)$ se, per ogni stringa di input x , il tempo richiesto da M su x è al più $f(|x|)$ (con $|x|$ si indica la lunghezza della stringa x). La funzione è il limite di tempo per M .

Considerando nuovamente la macchina di Turing per il riconoscimento delle stringhe palindrome, posto n la lunghezza della stringa di input, si può facilmente valutare che la macchina opera in $\lceil n/2 \rceil$ fasi. Nella prima fase, in $n+1$ passi la macchina scorre tutti i caratteri della stringa di input e confronta il primo e l'ultimo. Successivamente, in n passi la macchina si posiziona sul secondo carattere della stringa ed è pronta a ripetere la procedura. Complessivamente in $2n+1$ passi la macchina confronta i caratteri alle estremità

della stringa e si riposiziona all'inizio della stessa cosicchè il procedimento è ripetuto con una stringa di lunghezza $n - 2$, quindi con una di lunghezza $n - 4$ e così via. Il numero totale di passi è

$$(2n + 1) + (2n - 3) + (2n - 7) + (2n - 11) + \dots$$

che, supponendo per semplicità che n sia un numero pari, equivale a calcolare

$$\begin{aligned} \sum_{k=0}^{n/2} (4k + 1) &= 4 \sum_{k=0}^{n/2} k + \sum_{k=0}^{n/2} 1 = 4 \frac{n}{2} \left(\frac{n}{2} + 1 \right) \frac{1}{2} + \frac{n}{2} + 1 = \\ &= n \left(\frac{n}{2} + 1 \right) + \frac{n}{2} + 1 = \left(\frac{n}{2} + 1 \right) (n + 1), \end{aligned}$$

cosicchè si ottiene

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

ovvero

$$f(n) = \mathcal{O}(n^2).$$

Naturalmente questa stima è notevolmente pessimistica, poichè se la stringa non è palindroma il riconoscimento può avvenire in molti meno passi (se il primo e l'ultimo simbolo della stringa sono diversi i passi sono al più $n + 3$). Tuttavia si deve considerare che l'analisi della complessità degli algoritmi va effettuata sempre nel caso peggiore (che accade quando effettivamente una stringa è palindroma e quindi tutti i suoi simboli vanno verificati).

Complessità in Spazio

Intuitivamente si può definire la complessità in spazio di una macchina di Turing come il numero di celle del nastro che vengono utilizzate da M per terminare la computazione. Per quello che riguarda il numero da associare a tale concetto intuitivo sono possibili diverse alternative, tutte plausibili, e che comunque differiscono di poco dal punto di vista sostanziale. Considerando tutte le configurazioni assunte da M nel corso della computazione, (q, u_i, v_i) una prima scelta è quella di considerare il massimo numero di caratteri da cui è composta l'unione delle stringhe u_i, v_i , cioè, detta $g(n)$ la funzione che misura la complessità in spazio della macchina M applicata ad una stringa x di lunghezza n , poniamo

$$g(n) = \max_i |u_i v_i|.$$

Un secondo approccio consiste nel considerare la somma delle lunghezze di tali stringhe al variare di tutte le configurazioni assunte durante il corso della computazione:

$$g(n) = \sum_i |u_i v_i|.$$

Esiste anche un terzo approccio, che in verità è applicato soprattutto nel caso di Macchine di Turing multinastro e nell'ipotesi che si eviti la possibilità che la stringa di input possa diventare più corta durante la computazione. Infatti se (H, u_t, v_t) è la configurazione finale, cioè $H \in \{h, yes, no\}$ allora si pone

$$g(n) = |u_t v_t|$$

cioè il numero di celle occupate dalla configurazione finale.

1.4.4 Random Access Machine (RAM)

Le Macchine di Turing, a dispetto della loro semplice struttura, sono strumenti abbastanza potenti, perciò ha senso chiedersi se esse sono in grado di descrivere tutti gli algoritmi. La risposta affermativa a tale quesito è fornita dalla cosiddetta *Tesi di Church-Turing* che afferma che se un problema può essere risolto allora esisterà una macchina di Turing in grado di risolverlo, o, più in generale l'insieme delle funzioni calcolabili coincide con quello delle funzioni calcolabili da una macchina di Turing. Tale tesi, universalmente ritenuta vera, non è stata tuttavia provata. In realtà ciò che è stato dimostrato negli ultimi anni sono le seguenti proprietà:

1. Tutti i modelli di calcolo introdotti negli ultimi anni hanno la stessa potenza delle macchine di Turing;
2. Il costo richiesto da un algoritmo per la trasformazione di un algoritmo in un qualsiasi modello di calcolo in una macchina di Turing è di ordine polinomiale.

Quest'ultimo risultato riguarda anche il modello di macchina di Turing a k nastri che può essere simulato con una macchina di Turing a un singolo nastro utilizzando un algoritmo di costo $\mathcal{O}(n^2)$.

Una **RAM** (*Random Access Machine*, cioè *Macchina ad Accesso Casuale*) è un dispositivo di calcolo che consiste in un programma che agisce su una struttura di dati. Tale struttura è composta da un numero infinito di registri,

ognuno dei quali può contenere un numero intero arbitrariamente grande positivo o negativo. La macchina può operare su tali registri attraverso la seguente serie di istruzioni:

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

dove:

- j è un numero intero;
- r_j è l'attuale contenuto del Registro j ;
- i_j è l' i -esimo input;
- il valore dell'operando j è r_j , quello di $\uparrow j$ è r_{r_j} ;
- κ è il cosiddetto Program Counter;
- Tutte le istruzioni incrementano di 1 il Program Counter, a meno che l'istruzione non specifichi l'assegnazione di un valore (istruzioni di salto).

Formalmente una macchina RAM è in grado di eseguire un programma

$$\Pi = (\pi_1, \pi_2, \dots, \pi_m)$$

in cui ogni π_i è una delle istruzioni definite precedentemente. I registri vengono indicati con numeri interi positivi. Una particolare importanza ha il registro 0, detto *accumulatore* perchè in esso avvengono tutte le computazioni aritmetiche e logiche. Ad un certo punto dell'esecuzione del programma Π ogni registro i contiene il valore intero r_i e viene eseguita l'istruzione π_κ . Il valore κ è il cosiddetto Program Counter cioè un numero intero che specifica l'istruzione da eseguire. Il suo valore viene incrementato di 1 dopo l'esecuzione dell'istruzione, tranne nel caso delle istruzioni di salto in cui il valore assunto dal Program Counter può essere cambiato.

Un programma RAM termina in due circostanze:

1. quando viene eseguita l'istruzione HALT;
2. quando viene incontrata una istruzione sintatticamente errata (per esempio si fa riferimento ad un registro negativo, per esempio STORE -4, oppure ad un'istruzione non esistente, come JUMP -6).

Inizialmente tutti il contenuto di tutti i registri è inizializzato a zero e il valore del Program Counter è 1. I dati di input di un programma RAM sono memorizzati in un array finito di *Registri di input*:

$$I = (i_1, i_2, \dots, i_k).$$

Il contenuto di un registro di input può essere trasferito nell'accumulatore attraverso l'istruzione READ. Quando si esegue l'istruzione HALT il valore di output deve essere memorizzato nell'accumulatore. Quando viene eseguita la prima istruzione del programma la RAM modifica il contenuto dei registri in base ad essa, modifica il valore del Program Counter in κ e passa all'esecuzione dell'istruzione π_κ e prosegue fino alla fine del programma. Le computazioni di una RAM possono essere meglio formalizzate definendo il concetto di *configurazione*.

La configurazione di una RAM è una coppia $C = (\kappa, R)$, dove κ è il valore del Program Counter relativo all'istruzione da eseguire, mentre R è un insieme di coppie ordinate:

$$R = \{(j_1, r_{j_1}), (j_2, r_{j_2}), \dots, (j_k, r_{j_k})\}$$

che rappresentano tutti i registri il cui valore è stato modificato dalla RAM e il loro contenuto. La configurazione iniziale della RAM è:

$$C = (1, \emptyset).$$

Se Π è un programma RAM che agisce sull'input $I = (i_1, i_2, \dots, i_m)$, $C = (\kappa, R)$ e $C' = (\kappa', R')$ sono due configurazioni, allora si dice che la configurazione $C = (\kappa, R)$ produce in un passo la configurazione $C' = (\kappa', R')$ e si scrive

$$(\kappa, R) \xrightarrow{\Pi, I} (\kappa', R')$$

se:

1. κ' è il nuovo valore del Program Counter dopo l'esecuzione dell'istruzione π_κ ;
2. il nuovo insieme R' è uguale a R se è stata eseguita un'istruzione di salto o l'istruzione HALT, oppure se è uguale a R da cui è stata cancellata la coppia (j, x) e aggiunta la coppia (j, x') se il contenuto del registro j è cambiato da x a x' .

In modo simile a quanto visto per le macchine di Turing possiamo dire che la configurazione (κ, R) produce in m passi la configurazione (κ', R') e si scrive

$$(\kappa, R) \xrightarrow{\Pi, I^m} (\kappa', R')$$

se tale passaggio avviene dopo l'esecuzione di m istruzioni, e, inoltre, la configurazione (κ, R) produce la configurazione (κ', R') e si scrive

$$(\kappa, R) \xrightarrow{\Pi, I^*} (\kappa', R')$$

se esiste un numero m intero positivo tale che:

$$(\kappa, R) \xrightarrow{\Pi, I^m} (\kappa', R').$$

Sia Π un programma RAM, D un insieme finito di numeri interi e ϕ una funzione definita da D nell'insieme dei numeri interi. Si dice che Π calcola ϕ se per ogni $I \in D$ si ha:

$$(1, \emptyset) \xrightarrow{\Pi, I^*} (0, R)$$

dove $(0, \phi(I)) \in R$.

Esempio 1.4.2 *Come primo esempio di RAM consideriamo quella che trova il massimo tra due numeri interi. L'input in questo caso è rappresentato dai due numeri interi i_1 e i_2 .*

1	READ	2
2	STORE	2
3	READ	1
4	STORE	1
5	SUB	2
6	JNEG	9
7	LOAD	1
8	HALT	
9	LOAD	2
10	HALT	

Riscriviamo ora il programma RAM evidenziando le operazioni effettuate sui registri.

1	READ	2	$r_0 := i_2$
2	STORE	2	$r_2 := r_0$
3	READ	1	$r_0 := i_1$
4	STORE	1	$r_1 := r_0$
5	SUB	2	$r_0 := r_0 - r_2 := i_1 - i_2$
6	JNEG	9	if $r_0 < 0$ then $\kappa := 9$
7	LOAD	1	$r_0 := r_1$
8	HALT		$\kappa := 0$
9	LOAD	2	$r_0 := r_2$
10	HALT		$\kappa := 0$

Le prime quattro istruzioni servono a leggere i dati dai registri di input e a memorizzarli nei registri della RAM. L'istruzione 5 effettua la differenza tra i numeri interi e quindi se negativa vuol dire che il maggiore è quello memorizzato nel registro 2 che viene quindi spostato nell'accumulatore (istruzione 9), altrimenti il risultato voluto è il dato memorizzato nel registro 1 che viene memorizzato nell'accumulatore (istruzione 7). Osserviamo inoltre che l'istruzione 8 potrebbe essere sostituita dall'istruzione JUMP 10, che avrebbe lo stesso effetto.

Consideriamo ora l'insieme delle configurazioni della RAM che abbiamo appena descritto quando viene applicata ai valori di input $I = (3, 7)$, cioè i dati di input sono $i_1 = 3$ e $i_2 = 7$.

κ	R
1	\emptyset
2	$\{(0, 7)\}$
3	$\{(0, 7), (2, 7)\}$
4	$\{(0, 3), (2, 7)\}$
5	$\{(0, 3), (1, 3), (2, 7)\}$
6	$\{(0, -4), (1, 3), (2, 7)\}$
9	$\{(0, -4), (1, 3), (2, 7)\}$
10	$\{(0, 7), (1, 3), (2, 7)\}$
0	$\{(0, 7), (1, 3), (2, 7)\}$

Esempio 1.4.3 Consideriamo ora la Macchina RAM che calcola la somma di una sequenza di numeri interi letti dall'input e che termina con un elemento uguale a zero.

1	LOAD	= 1	$r_0 := 1$
2	STORE	1	$r_1 := r_0$
3	READ	$\uparrow 1$	$r_0 := i_{r_1}$
4	JZERO	11	if $r_0 \leq 0$ then $\kappa := 11$
5	ADD	2	$r_0 := r_0 + r_2$
6	STORE	2	$r_2 := r_0$
7	LOAD	1	$r_0 := r_1$
8	ADD	= 1	$r_0 := r_0 + 1$
9	STORE	1	$r_1 := r_0$
10	JUMP	3	$\kappa := 3$
11	LOAD	2	$r_0 := r_2$
12	HALT		$\kappa := 0$

Consideriamo ora l'insieme delle configurazioni della macchina RAM appena descritta quando viene applicata all'input

$$I = (i_1 = 7, i_2 = 3, i_3 = 0).$$

κ	R
1	\emptyset
2	$\{(0, 1)\}$
3	$\{(0, 1), (1, 1)\}$
4	$\{(0, 7), (1, 1)\}$
5	$\{(0, 7), (1, 1)\}$

6	{(0, 7), (1, 1)}
7	{(0, 7), (1, 1), (2, 7)}
8	{(0, 1), (1, 1), (2, 7)}
9	{(0, 2), (1, 1), (2, 7)}
10	{(0, 2), (1, 2), (2, 7)}
3	{(0, 2), (1, 2), (2, 7)}
4	{(0, 3), (1, 2), (2, 7)}
5	{(0, 3), (1, 2), (2, 7)}
6	{(0, 10), (1, 2), (2, 7)}
7	{(0, 10), (1, 2), (2, 10)}
8	{(0, 2), (1, 2), (2, 10)}
9	{(0, 3), (1, 2), (2, 10)}
10	{(0, 3), (1, 3), (2, 10)}
3	{(0, 3), (1, 3), (2, 10)}
4	{(0, 0), (1, 3), (2, 10)}
11	{(0, 0), (1, 3), (2, 10)}
12	{(0, 10), (1, 3), (2, 10)}
0	{(0, 10), (1, 3), (2, 10)}

Esempio 1.4.4 Vediamo ora un esempio più complesso, cioè la macchina che effettua la moltiplicazione tra due numeri interi con rappresentazione binaria.

1	READ	1
2	STORE	1
3	STORE	5
4	READ	2
5	STORE	2
6	HALF	
7	STORE	3
8	ADD	3
9	SUB	2
10	JZERO	14
11	LOAD	4
12	ADD	5
13	STORE	4
14	LOAD	5
15	ADD	5

16	STORE	5
17	LOAD	3
18	JZERO	20
19	JUMP	5
20	LOAD	4
21	HALT	

Analizziamo ora le configurazioni assunte dalla RAM che esegue la moltiplicazione tra due numeri interi nel caso in cui $i_1 = 4$ e $i_2 = 3$:

κ	R
1	\emptyset
2	$\{(0, 4)\}$
3	$\{(0, 4), (1, 4)\}$
4	$\{(0, 4), (1, 4), (5, 4)\}$
5	$\{(0, 3), (1, 4), (5, 4)\}$
6	$\{(0, 3), (1, 4), (2, 3), (5, 4)\}$
7	$\{(0, 1), (1, 4), (2, 3), (5, 4)\}$
8	$\{(0, 1), (1, 4), (2, 3), (3, 1), (5, 4)\}$
9	$\{(0, 2), (1, 4), (2, 3), (3, 1), (5, 4)\}$
10	$\{(0, -1), (1, 4), (2, 3), (3, 1), (5, 4)\}$
11	$\{(0, -1), (1, 4), (2, 3), (3, 1), (5, 4)\}$
12	$\{(0, 0), (1, 4), (2, 3), (3, 1), (5, 4)\}$
13	$\{(0, 4), (1, 4), (2, 3), (3, 1), (5, 4)\}$
14	$\{(0, 4), (1, 4), (2, 3), (3, 1), (4, 4), (5, 4)\}$
15	$\{(0, 4), (1, 4), (2, 3), (3, 1), (4, 4), (5, 4)\}$
16	$\{(0, 8), (1, 4), (2, 3), (3, 1), (4, 4), (5, 4)\}$
17	$\{(0, 8), (1, 4), (2, 3), (3, 1), (4, 4), (5, 8)\}$
18	$\{(0, 1), (1, 4), (2, 3), (3, 1), (4, 4), (5, 8)\}$
19	$\{(0, 1), (1, 4), (2, 3), (3, 1), (4, 4), (5, 8)\}$
5	$\{(0, 1), (1, 4), (2, 3), (3, 1), (4, 4), (5, 8)\}$
6	$\{(0, 1), (1, 4), (2, 1), (3, 1), (4, 4), (5, 8)\}$
7	$\{(0, 0), (1, 4), (2, 3), (3, 1), (4, 4), (5, 8)\}$
8	$\{(0, 0), (1, 4), (2, 3), (3, 0), (4, 4), (5, 8)\}$
9	$\{(0, 0), (1, 4), (2, 3), (3, 0), (4, 4), (5, 8)\}$
10	$\{(0, -3), (1, 4), (2, 3), (3, 0), (4, 4), (5, 8)\}$
11	$\{(0, -3), (1, 4), (2, 3), (3, 0), (4, 4), (5, 8)\}$
12	$\{(0, 4), (1, 4), (2, 3), (3, 0), (4, 4), (5, 8)\}$

13	$\{(0, 12), (1, 4), (2, 3), (3, 0), (4, 4), (5, 8)\}$
14	$\{(0, 12), (1, 4), (2, 3), (3, 0), (4, 12), (5, 8)\}$
15	$\{(0, 8), (1, 4), (2, 3), (3, 0), (4, 12), (5, 8)\}$
16	$\{(0, 16), (1, 4), (2, 3), (3, 0), (4, 12), (5, 8)\}$
17	$\{(0, 16), (1, 4), (2, 3), (3, 0), (4, 12), (5, 16)\}$
18	$\{(0, 0), (1, 4), (2, 3), (3, 0), (4, 12), (5, 16)\}$
20	$\{(0, 0), (1, 4), (2, 3), (3, 0), (4, 12), (5, 16)\}$
21	$\{(0, 12), (1, 4), (2, 3), (3, 0), (4, 12), (5, 16)\}$
0	$\{(0, 12), (1, 4), (2, 3), (3, 0), (4, 12), (5, 16)\}$

Dalle configurazioni si evince subito che i due valori di input i_1 e i_2 sono memorizzati rispettivamente nei Registri 1 e 2. Più nel dettaglio il programma ripete $\lceil \log_2 i_2 \rceil$ volte le istruzioni comprese tra la 5 e la 19. All'inizio della k -esima iterazione (iniziando con l'iterata zero) il Registro 3 contiene $\lfloor i_3/2^k \rfloor$. Il Registro 5 contiene $i_1 2^k$ ed il Registro 4 contiene $i_1(i_2 \bmod 2^k)$. Alla fine delle iterazioni si testa il valore del Registro 3: se è uguale a zero il programma si arresta e fornisce in output il valore del Registro 4 altrimenti prosegue.

Per valutare il costo computazionale di una RAM introduciamo la seguente semplificazione: il costo dell'esecuzione di una singola istruzione è indipendente dal numero di cifre della rappresentazione binaria degli operandi. Tale ipotesi è ovviamente non realistica ma dobbiamo considerare che le RAM sono essenzialmente delle rappresentazioni ideali di macchine reali.

Se i è un numero intero sia $b(i)$ la sua rappresentazione binaria, senza cifre 0 ridondanti all'inizio e con il segno meno se negativo. Si definisce lunghezza di i il numero:

$$\ell(i) = |b(i)|$$

pari al numero di cifre binarie della sua rappresentazione. Se $I = (i_1, i_2, \dots, i_k)$ è una sequenza di interi allora la lunghezza di I è:

$$\ell(I) = \sum_{j=1}^k \ell(i_j).$$

Supponiamo che un programma RAM Π calcoli la funzione ϕ definita nel dominio D a valori nell'insieme dei numeri interi.

Se $f : \mathbb{N} \rightarrow \mathbb{N}$ e supponiamo che per ogni $I \in D$ si ha

$$(1, \emptyset) \xrightarrow{\Pi, I^k} (0, R)$$

dove $k \leq f(\ell(I))$, allora si dice che Π calcola la funzione ϕ in un tempo $f(n)$. Quindi il tempo richiesto dalla RAM coincide con il numero di configurazioni necessarie per calcolare la funzione ϕ partendo dall'input I espresso in funzione della lunghezza di I .

Volendo valutare la complessità delle due RAM presentate negli esempi precedenti si può osservare facilmente che la prima richiede un numero di configurazioni costante per fornire il risultato quindi il suo costo computazionale è

$$f(n) = \mathcal{O}(c)$$

con c costante. Al contrario il programma per il calcolo della moltiplicazione binaria tra due numeri interi richiede $\mathcal{O}(n)$ istruzioni perchè il numero di istruzioni richieste è proporzionale al logaritmo degli interi coinvolti nelle operazioni. Questo è una conseguenza del fatto che il numero di iterazioni richieste dal programma è al più $\log_2 i_2 \leq \ell(I)$ e ogni iterazione richiede un numero costante di istruzioni.

1.5 Efficienza degli algoritmi

Nei paragrafi precedenti abbiamo descritto in breve i due modelli di calcolo più noti per la codifica degli algoritmi e abbiamo accennato alle modalità per la valutazione del costo computazionale di un algoritmo. Le problematiche inerenti tale problema hanno evidenziato come tale valutazione richieda molto spesso un approccio di tipo intuitivo e che è difficile fornire delle regole certe per compiere tale operazione. È invece molto importante riepilogare le assunzioni da tener presente quando si deve valutare il costo di un algoritmo e le ripercussioni che esso ha nella classificazione dei problemi:

1. **Codifica Ragionevole delle Istanze:** si deve cioè presupporre che i dati del problema siano stati memorizzati nel miglior modo possibile;
2. **Riferimento alla Complessità in Tempo:** tra tempo e spazio la risorsa da privilegiare è sempre il tempo;
3. **Modello di Calcolo Deterministico e Sequenziale:** i modelli di calcolo non deterministici non hanno un interesse pratico, quindi vanno privilegiati modelli di interesse reale;

4. **Criterio di Costo Uniforme:** il tempo di calcolo va eguagliato al numero di istruzioni (operazioni aritmetiche, confronti) necessario per risolvere una determinata istanza, nel *Criterio di Costo Logaritmico* si tiene conto anche del numero di bit necessari a codificare i dati;
5. **Analisi della Complessità nel Caso Peggior** [*Worst Case Analysis*]: il costo computazionale va calcolato sempre nei casi meno favorevoli cosicchè si possa ragionevolmente ritenere che il comportamento dell'algoritmo sia migliore di quanto atteso.

Per quest'ultima osservazione potremmo aggiungere che in alternativa si potrebbe analizzare il costo dell'algoritmo nel caso migliore, tuttavia si può facilmente comprendere che tale valore non avrebbe un grande significato. Talvolta può essere utile una specie di analisi inversa del minor costo, cioè lo studio delle caratteristiche dei dati per i quali un determinato algoritmo ha un costo minore.

1.5.1 Il criterio di efficienza polinomiale

Analizzando l'algoritmo per determinare la raggiungibilità dei grafi abbiamo notato che esso richiede un tempo proporzionale a n^2 quindi è $\mathcal{O}(n^2)$. Va però osservato che tale stima è in realtà piuttosto pessimistica, in accordo con il criterio di analisi del costo nel caso peggiore che abbiamo descritto nel precedente paragrafo. Ciò che lascia soddisfatti è il fatto che la velocità di crescita del tempo dipende da n^2 . Si parla di *velocità di crescita polinomiale* come uno dei requisiti che consentono di affermare che un problema è risolto in modo soddisfacente da un algoritmo. Al contrario velocità di crescita esponenziali come 2^n o, peggio, come $n!$, possono essere sintomo di una notevole inefficienza. Appare ovvio che tale fattore dipende dall'algoritmo usato e, se usando diversi algoritmi la velocità di crescita non diminuisce allora questo può essere un sintomo del fatto che tale problema non ammette algoritmi che diano la soluzione in un tempo polinomiale cioè non è **trattabile**. Questa dicotomia tra limiti di tempo polinomiale e non polinomiale e soprattutto l'identificazione tra algoritmi polinomiali e computazione pratica non è tuttavia priva di controversie. Ci sono algoritmi efficienti che non sono polinomiali e viceversa algoritmi polinomiali che non sono efficienti. Per esempio un algoritmo $\mathcal{O}(n^{80})$ sarà di limitato valore pratico mentre un algoritmo esponenziale $\mathcal{O}(2^{n/100})$ oppure *sub-esponenziale* $\mathcal{O}(n^{\log n})$ può essere più utile. Ci sono tuttavia delle forti argomentazioni che spingono comunque verso il tipo

polinomiale. Innanzitutto il fatto che ogni funzione con velocità di crescita polinomiale sarà superata da una qualsiasi funzione esponenziale cosicchè un algoritmo del secondo tipo può essere preferito al primo solo fino ad un certo valore. La seconda considerazione è che raramente si raggiungono valori estremi come $\mathcal{O}(n^{80})$ oppure $\mathcal{O}(2^{n/100})$. Nella pratica algoritmi polinomiali hanno piccoli esponenti e ragionevoli costanti moltiplicative mentre quelli esponenziali sono impraticabili sempre. Un'ulteriore motivazione a sostegno del criterio di efficienza polinomiale è dovuta al fatto che essi sono comunque sempre efficienti nella pratica mentre gli algoritmi esponenziali non lo sono quasi mai. Infatti abbiamo osservato più volte che l'analisi del costo computazionale viene effettuata sempre nel peggiore dei casi il che vuol dire che, nella pratica, ci aspettiamo che un algoritmo si comporti mediamente meglio di quanto sappiamo. Per gli algoritmi polinomiali vuol dire che il comportamento sarà ancora polinomiale (magari si abbassa l'esponente di n , oppure la costante moltiplicativa si riduce). Per gli algoritmi esponenziali un migliore comportamento non vuol dire che arriveremo ad un comportamento polinomiale dell'algoritmo (ciò può accadere ma non è detto che sia sempre così per esempio si può ridurre il costo da 3^n a 2^n , ma l'algoritmo resta esponenziale). La prestazione esponenziale di un algoritmo potrebbe accadere solo in determinate situazioni e per un insieme di dati statisticamente irrilevante e quindi potrebbe accadere che mediamente l'algoritmo si comporti in modo soddisfacente. Si potrebbe quindi analizzare il comportamento di un algoritmo nel caso di un insieme atteso di dati e non nel peggiore dei casi: il problema è che molto raramente si conosce la distribuzione dell'input di un problema, cioè la probabilità che un certo dato occorra come input, cosicchè tale analisi è impossibile. Inoltre se il nostro interesse è risolvere il problema per un particolare insieme di dati e ci accorgiamo che questo è uno di quei casi in cui l'algoritmo si comporta peggio non ci aiuta o ci consola certo sapere che questo è un'eccezione statisticamente insignificante. Considerare come criterio di efficienza la performance di un algoritmo nel peggiore dei casi risulta in una teoria elegante ed utile che dice qualcosa di significativo circa la computazione pratica e sarebbe impossibile senza tale semplificazione. Tuttavia c'è anche una motivazione pratica dal punto di vista matematico: infatti i polinomi formano una classe di funzioni chiusa rispetto all'addizione, alla moltiplicazione e inoltre i logaritmi di polinomi sono tutti legati da una costante (essi sono $\Theta(\log n)$).

1.5.2 Progresso tecnologico e complessità computazionale

Consideriamo un algoritmo A di complessità $\mathcal{O}(2^n)$ ed un algoritmo B di complessità $\mathcal{O}(n^k)$ ed un terzo algoritmo con complessità in tempo $\log_2 n$. Supponiamo di essere in grado, con la tecnologia esistente di usare A per risolvere problemi di dimensione inferiore ad m_A , cioè 2^{m_A} sia un limite di tempo oltre il quale non ha senso andare. Indichiamo con m_B ed m_C gli stessi limiti per gli algoritmi B e C rispettivamente. Supponiamo ora che in un ipotetico futuro sia stata sviluppata una tecnologia in grado di offrire dispositivi di calcolo con prestazioni α volte superiori rispetto a quelle esistenti e vediamo quale è la dimensione massima m'_A che può essere affrontata nella nuova situazione con lo stesso algoritmo A . Deve essere:

$$2^{m'_A} = \alpha 2^{m_A} = 2^{\log_2 \alpha} 2^{m_A} = 2^{\log_2 \alpha + m_A}$$

quindi

$$m'_A = m_A + \log_2 \alpha.$$

Ripetiamo lo stesso discorso per l'algoritmo B :

$$(m'_B)^k = \alpha m_B^k = (\sqrt[k]{\alpha})^k m_B^k = (\sqrt[k]{\alpha} m_B)^k$$

quindi

$$m'_B = \sqrt[k]{\alpha} m_B.$$

Per l'algoritmo C abbiamo invece:

$$\log_2 m'_C = \alpha \log_2 m_C = \log_2 (m_C^\alpha)$$

quindi

$$m'_C = m_C^\alpha.$$

Se fosse stato $\alpha = 256 = 2^8$ e $k = 4$ avremmo avuto

$$m'_A = m_A + 8, \quad m'_B = 4m_B, \quad m'_C = m_C^{256}.$$

A fronte di un progresso tecnologico che ha aumentato le potenzialità di calcolo di 256 volte si ha un incremento della dimensione massima di un fattore additivo pari a 8 (quindi se m era 1000 adesso vale 1008, praticamente è rimasto invariato). Ben poco può la tecnologia o, meglio, l'innovazione tecnologica di fronte ad algoritmi esponenziali.

Diverso è il discorso per l'algoritmo B perchè in questo caso il progresso

tecnologico si riflette in un fattore moltiplicativo il cui valore dipende dal grado del polinomio: minore il grado maggiore è l'incremento, cioè il fattore moltiplicativo. Infatti se il costo dell'algoritmo B fosse stato $\mathcal{O}(n^2)$ allora

$$k^2 = 256 m^2 = 16^2 m^2 \quad \Rightarrow \quad k = 16m.$$

Ovviamente il maggior vantaggio si ha per algoritmi a costo lineare $\mathcal{O}(n)$. Ancor maggiore è il vantaggio per algoritmi a costo di tipo logaritmico per i quali il progresso tecnologico comporta un aumento esponenziale delle dimensioni del problema.

1.6 Le classi di complessità

Per *classe di complessità* si intende un insieme di problemi risolvibili in un determinato modello di calcolo con una ben precisa limitazione sull'uso di una o più risorse. Quindi per determinare una classe di complessità si devono specificare le seguenti caratteristiche:

1. il modello di calcolo utilizzato;
2. la modalità con cui si eseguono le computazioni;
3. la risorsa disponibile per un certo limite;
4. la funzione $f : \mathbb{N} \rightarrow \mathbb{N}$ che esprime tale limite.

In realtà il punto 1. appare piuttosto superfluo poichè la cosiddetta *Tesi del Calcolo Sequenziale* stabilisce che tutti i modelli di calcolo sono polinomialmente correlati tra loro ed in particolare lo sono rispetto alle macchine di Turing. Il che significa che un algoritmo descritto dal programma di una RAM può essere trasformato in una macchina di Turing in un tempo polinomiale qualunque sia la sua complessità computazionale. Il passaggio da un modello all'altro non migliora (e non peggiora) il costo computazionale di algoritmi polinomiali.

Il punto 2. invece richiede che sia specificato se la modalità sia *deterministica*, *non deterministica* oppure *probabilistica*.

Il punto 3. invece fa riferimento al fatto se si misuri la complessità rispetto alla risorsa *Tempo* o *Spazio*.

Il punto 4. invece chiede di specificare quale tipo di funzione limiti la risorsa

e quindi quale sia la proprietà che condividono tutti i problemi considerati circa l'uso di tale risorsa.

Se $f(n)$ è tale funzione allora la classe $TIME(f(n))$ è l'insieme dei problemi che possono essere risolti in un tempo $f(n)$ in un qualsiasi modello di calcolo. Se $f(n) = n^k$ allora $TIME(n^k)$ è insieme dei problemi che ammettono un algoritmo che li risolve in tempo pari a n^k .

Analoghe classi si possono avere per la risorsa spazio: la classe $SPACE(f(n))$ è l'insieme dei problemi che possono essere risolti usando $f(n)$ spazio in un qualsiasi modello di calcolo.

Se $f(n) = n^k$ allora $SPACE(n^k)$ è insieme dei problemi che ammettono un algoritmo che li risolve usando spazio uguale a n^k .

Scegliendo come risorsa il Tempo allora si definisce la cosiddetta **Classe P** come l'insieme di tutti i problemi risolvibili in tempo polinomiale. In modo più rigoroso la classe P non è altro che:

$$P = \bigcup_{k=0}^{\infty} TIME(n^k).$$

E analogamente per la risorsa spazio

$$PSPACE = \bigcup_{k=0}^{\infty} SPACE(n^k).$$

In modo analogo si possono definire classi come la EXPTIME, cioè quella composta da tutti i problemi la cui complessità di calcolo calcolata rispetto alla risorsa tempo è di tipo esponenziale:

$$EXPTIME = \bigcup_{k=0}^{\infty} TIME(2^{n^k}).$$

Un'altra classe interessante è la seguente:

$$L = SPACE(\log_2 n).$$

L'appartenenza di un problema ad una classe di complessità dipende essenzialmente dal costo computazionale dell'algoritmo più efficiente. Per attribuire la funzione costo ad un problema si considera l'insieme degli algoritmi che lo risolvono $\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_k$, e le relative funzioni di complessità relative ad una determinata risorsa, $f_1(n), f_2(n), \dots, f_k(n)$. Si sceglie l'algoritmo con la funzione avente una velocità di crescita inferiore, al variare di n , e si associa tale funzione al problema. Quindi tale proprietà dipende essenzialmente dal progresso nello studio di algoritmi efficienti per risolvere problemi.

Non determinismo e classe NP

Prima di introdurre il concetto di non determinismo riprendiamo un esempio che abbiamo già incontrato: il problema del commesso viaggiatore. Sappiamo già che probabilmente non esistono algoritmi polinomiali per risolverlo e consideriamo la versione del problema sotto forma decisionale, cioè fissiamo una soglia B e chiediamoci se esiste un percorso avente lunghezza inferiore a B , cioè se esiste una permutazione $(\pi_1, \pi_2, \dots, \pi_{n-1}, \pi_n)$ dell'insieme $\{1, 2, \dots, n-1, n\}$ tale che

$$\sum_{i=1}^{n-1} d_{\pi_i, \pi_{i+1}} + d_{\pi_n, \pi_1} \leq B.$$

Supponiamo che un qualcuno affermi che la risposta a tale quesito è SI e fornisca la permutazione corrispondente, cioè l'ordine di visita delle n città. In questo caso è possibile verificare in tempo polinomiale se tale affermazione risulti vera o meno poichè basta calcolare la somma delle n distanze. Il problema del commesso viaggiatore appartiene ad una classe di problemi per i quali la veridicità di una soluzione può essere verificata efficientemente (cioè in tempo polinomiale).

Tale esempio mette in evidenza un altro aspetto della complessità degli algoritmi, cioè:

È più difficile calcolare la soluzione di un problema oppure verificare la correttezza di un'ipotetica soluzione?

È opinione generale (e l'esempio appena visto ne è la conferma) che la verifica è più facile del calcolo, anche se in realtà ciò è vero solo per alcuni tipi di problemi (per esempio quelli decisionali). Abbiamo quindi osservato che in definitiva esiste una seconda classe di problemi che presentano la seguente caratteristica: pur essendo presumibilmente difficilmente trattabili, essi emettono una procedura di verifica estremamente efficiente. Questa comune proprietà è espressa dalla classe di complessità **NP** che può essere definita in modo informale come l'insieme dei problemi decisionali per i quali le ipotetiche soluzioni possono essere verificate in tempo polinomiale. Per fornire una definizione formale della classe NP è necessario introdurre il concetto di non determinismo. Per *computazione non deterministica* si intende un modello di calcolo in cui un algoritmo viene visto come un insieme di passi in cui è possibile ottenere dei suggerimenti esterni circa le scelte da effettuare. Tale tipo di computazione gode delle seguenti proprietà:

- il modello di calcolo è in grado di generare la configurazione giusta e di verificare che essa effettivamente è una soluzione del problema;
- dovendo scegliere un percorso da seguire l'algoritmo compie sempre la scelta più opportuna (si utilizza il cosiddetto algoritmo di Gastone);
- l'algoritmo è in grado di procedere considerando simultaneamente tutte le possibili alternative, tra le quali anche quella che fornisce la soluzione;
- l'algoritmo è guidato da suggerimenti corretti rispetto alle scelte da compiere.

Appare superfluo mettere in evidenza che un tale modello di calcolo ha determinate valenze di tipo puramente teorico poichè non è supportato da nessuno strumento di calcolo attualmente disponibile. In modo ancora più formale: Una computazione si dice non deterministica quando consiste in una successione di passi discreti, ciascuno dei quali è una transizione da uno stato ad un insieme di stati (piuttosto che ad un singolo stato).

Il gran numero di problemi presenti nella classe NP ha spinto i ricercatori ad uno studio più raffinato dei suoi elementi, al fine di poter individuare delle caratteristiche comuni ad una parte di esse così da poter caratterizzare meglio l'intera classe. La prima osservazione è che dal punto di vista non deterministico nei problemi di decisione esiste una differenza tra i cammini non deterministici che corrispondono alla risposta SI da quelli che corrispondono alla risposta NO. Nel primo caso l'appartenenza alla classe NP garantisce che esiste un cammino di lunghezza limitata mentre nel secondo caso non si può dire nulla.

Esempio 1.6.1 (Asimmetria in NP: caso del TSP) *Risposta dell'istanza=SI*

Nella fase di ipotesi viene prospettato un percorso che certifica che l'istanza è positiva e nella fase di verifica viene eseguito il controllo che la lunghezza del percorso non supera la soglia B.

Risposta dell'istanza=NO

Se la verifica dà risultato negativo questo non vuol dire che il valore dell'istanza è NO. In altre parole non esiste alcun algoritmo non deterministico in grado di risolvere in tempo polinomiale il problema complementare del TSP.

Il problema complementare è, in generale, quello che si ottiene negando il problema stesso. Nel caso del TSP il complementare formula il seguente quesito:

È vero che la lunghezza di tutti i percorsi è superiore a B ?

Tale asimmetria nasce dal fatto che è semplice verificare la positività di un'istanza poichè essa è caratterizzata dall'esistenza di una permutazione, mentre per un'istanza negativa non sembra possibile individuare una prova diversa dall'elenco di tutte le permutazioni.

Tale situazione è però comune a molti problemi appartenenti a NP. Dalle osservazioni fatte ha senso considerare una terza classe di problemi cioè quella dei problemi complementari della classe NP, detta classe co-NP. In modo analogo si può definire la classe co-P.

Uno dei filoni di ricerca più recenti nello studio della complessità di problemi è la relazione che lega le classi P ed NP. Poichè gli algoritmi deterministici possono essere visti come casi particolari di quelli non deterministici è palese che vale la seguente inclusione:

$$P \subseteq NP.$$

Lo studio della validità o meno dell'inclusione inversa, ovvero se i problemi NP siano risolvibili in modo efficiente (ovvero utilizzando algoritmi con costo polinomiale, cioè siano trattabili o meno), è uno dei problemi aperti della Teoria della Complessità Computazionale. Ci sono tuttavia molte ragioni che inducono a credere che $P \neq NP$, ma non esiste una dimostrazione decisiva in tal senso.

È possibile indirizzare lo studio delle relazioni tra le classi P ed NP cercando i problemi più difficili all'interno della classe NP che sono i naturali candidati alla non appartenenza a P. In particolare è possibile individuare una classe di problemi, i problemi completi per NP, la cui complessità è rappresentativa della complessità dell'intera classe e la cui appartenenza a P implicherebbe l'uguaglianza $P=NP$.

Un'altra classe molto studiata di problemi è costituita da quei problemi che possono essere ridotti¹ a problemi in NP ma di cui non sono note dimostrazioni di appartenenza a NP. Questi problemi sono detti NP-hard e sono difficili quanto (se non di più) i problemi NP-completi. A questa classe appartengono soprattutto problemi di ottimizzazione di ricerca per i quali risulta essere di classe NP anche il solo cercare una soluzione approssimata per questo un'ulteriore ricerca consiste nel cercare (in tempo polinomiale o polinomiale non deterministico) soluzioni anche solo approssimate.

¹Un problema si dice riducibile ad un altro se, partendo da un metodo di risoluzione del secondo, è possibile ricostruire la soluzione del primo problema.