Algoritmi e Principi dell'Informatica

Fabio Vokrri

25 giugno 2022

Indice

Parte I

Informatica Teorica

Capitolo 1

Linguaggi Formali

1.1 Alfabeti e Stringhe

Di seguito sono riportate alcune definizioni importanti riguardanti i linguaggi formali, necessarie per affrontare la quasi totalità degli argomenti presentati in questo documento:

Definizione 1.1.1 (Alfabeto)**.** *Un alfabeto (o vocabolario) V è un insieme finito di oggetti elementari detti simboli (o caratteri)*

Definizione 1.1.2 (Stringa)**.** *Una stringa x appartenente ad un alfabeto V è una sequenza finita e ordinata di caratteri appartenenti a tale alfabeto.*

Data una stringa x, la notazione $|\,x\,|$ indica la lunghezza di tale stringa, ovvero il numero di caratteri di cui è composta (la cardinalità del suo dominio V). Per convenzione la stringa vuota, che non contiene nessun carattere, è indicata con la lettera ε. La stringa vuota ε ha lunghezza zero.

Per poter confrontare due stringhe è necessario prima di tutto verificare che le due stringhe abbiano uguale lunghezza ($|x| = |y|$) e poi che $x_i = y_i \; \forall i = 0, 1, 2, ..., |x|$.

La concatenazione (o prodotto) $x.y$ di due stringhe x e y è una stringa composta da tutti i caratteri di x seguiti da tutti i caratteri di y. Inoltre, la concatenazione $x.\varepsilon$ produce come risultato la stringa x.

Data una stringa s, una stringa x è sottostringa (o fattore) di s se esistono due stringhe y e z, tali che $s = yxz$. Inoltre:

- se $y = \varepsilon$, x è detta prefisso.
- se $z = \varepsilon$, x è detta suffisso.
- se $z = y = \varepsilon$, $x = s$

Data una stringa x, l'espressione x^i indica la concatenazione della stringa x con sè stessa per $i - 1$ volte. In maniera induttiva:

- $x^0 = \varepsilon$;
- $x^{i+1} = x.x^i$.

1.2 Operatori di Kleene

Per poter proseguire con il trattato, è prima necessario introdurre alcuni concetti matematici fondamentali. Si danno quindi le seguenti definizioni:

Definizione 1.2.1 (Semigruppo)**.** *Un semigruppo è una coppia $< S, \circ >$, dove:*

- *S è un insieme chiuso rispetto a $\circ$; per cui, se si prendono due qualsiasi elementi A e B di tale insieme, l'operazione $A \circ B$ produce come risultato un elemento appartenente ad S;*
- *$\circ$ è un'operazione associativa su S.*

Nel contesto dei linguaggi, l'operatore $\circ$ rappresenta la concatenazione di stringhe.

Definizione 1.2.2 (Monoide)**.** *Un monoide è un semigruppo in cui è definito un elemento unitario $u \in S$, tale che $\forall x, \exists u(x \circ u = x)$.*

Nel contesto dei linguaggi, l'elemento unitario u rappresenta la stringa vuota ε: infatti, la concatenazione di una generica stringa x con la stringa vuota ε produce come risultato nuovamente la stringa originaria x.

Definizione 1.2.3 (Gruppo)**.** *Un gruppo è un monoide in cui è definito un elemento inverso x^{-1} unico per ogni elemento x dell'insieme S, tale che $\forall x(x \circ x^{-1} = u)$.*

Date le definizioni 1.2.1, 1.2.2 e 1.2.3, si possono definire gli operatori di Kleene:

Definizione 1.2.4 (Più di Kleene)**.** *Sia $< S, \circ >$ un semigruppo. Per ogni $X \subseteq S$, S^+ indica il sottoinsieme di S generato da X, ovvero l'insieme di tutti gli elementi $s \in S$ tale per cui $s = x_1 \circ x_2 \circ ... \circ x_n$ per qualche $n \geq 1$, con $x_i \in X$ (stringhe non vuote).*

Nel contesto dei linguaggi, l'operatore + di Kleene rappresenta l'insieme infinito di stringhe non vuote, che si possono generare a partire dall'insieme S di simboli, a cui si applica tale operatore.

Definizione 1.2.5 (Stella di Kleene)**.** *Se $< S, \circ, u >$ è un monoide, allora $X^* = X^+ \cup \{u\}$ è un monoide (più precisamente un sottomonoide) di S ed è detto monoide libero generato da X.*

Nel contesto dei linguaggi, la stella di Kleene rappresenta l'insieme infinito di stringhe, inclusa la stringa vuota ε, che si possono generare a partire dall'insieme S di simboli, a cui si applica tale operatore. Quindi, $X^* = X^+ \cup \{\varepsilon\}$.

Ad esempio, dato l'insieme di simboli $S = \{a, b, c\}$, $S^+ = \{a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa...\}$ e $S^* = \{\varepsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa...\}$.

1.3 Linguaggi

Data la definizione della stella di Kleene, si può ora dare la definizione di linguaggio:

Definizione 1.3.1 (Linguaggio)**.** *Un linguaggio L su un alfabeto V è un sottoinsieme di V^*.*

Dato un insieme di simboli, si possono generare infiniti linguaggi.

Poichè i linguaggi sono un inisieme di stringhe, valgono tutte le operazioni insiemistiche come:

- Unione ($L_1 \cup L_2$): l'inisieme di tutte le stringhe che appartengono o ad L_1 o ad L_2 o ad entrambi i linguaggi;
- Intersezione ($L_1 \cap L_2$): l'insieme di tutte le stringhe che appartengono sia ad L_1 che ad L_2;
- Differenza ($L_1 \backslash L_2$): l'insieme di tutte le stringhe che appartengono ad L_1 ma non ad L_2;
- Complementare ($L^C = A^* \backslash L$): l'insieme di tutte le stringhe che non appartengono al linguaggio L;
- Concatenazione ($L_1.L_2$): l'insieme di tutte le stringhe che si ottengono concatenando ad ogni stringa di L_1 ogni stringa di L_2; formalmente $L_1.L_2 = \{xy : x \in L_1,\ y \in L_2\}$;
- Potenza n-esima (L^n): l'insieme di tutte le stringhe che si ottengono concatenando L con sè stesso n volte, utilizzando le regole della concatenazione precedentemente definite;
- Chiusura di Kleene $\left(L^* = \bigcup_{n=0}^{\infty} L^n \text{ e } L^+ = \bigcup_{n=1}^{\infty} L^n \right)$

Le operazioni su di un determinato linguaggio crea nuove classi di linguaggi con caratteristiche proprie, talvolta interessanti. Un linguaggio diventa di interesse nel momento in cui le stringhe di cui è composto possono essere utilizzate per veicolare informazioni, problemi, soluzioni o per rappresentare programmi, documenti, elementi multimediali o, nel caso più rilevante, per rappresentare computazioni.

Capitolo 2

Automi Deterministici

In questo capitolo verrà introdotto il concetto di automa e se ne analizzeranno le varie classi esistenti, differenziandole in base alla loro espressività nella rappresentazione di linguaggi via via sempre più complessi. Si inizierà con lo studio degli automi determinstici per passare gradualmente a quelli non deterministici, analizzandone le differenze e le caratteristiche più importanti. Di seguito sono date le definizioni di automa, accettatore e trasduttore:

Definizione 2.0.1. *Gli automi sono sistemi dinamici discreti e tempo invarianti. Lo stato di un automa viene modificato a fronte di un simbolo di ingresso. Ogni automa presenta uno stato iniziale, da cui ha inizio la sua esecuzione, e uno o più stati finali in cui la sua esecuzione ha termine.*

Definizione 2.0.2. *Gli accettatori sono automi che indicano se una determinata sequenza di simboli appartiene ad un linguaggio.*

Definizione 2.0.3. *I trasduttori sono automi che mappano una determinata sequenza di simboli appartenenti ad un linguaggio in una sequenza di simboli appartenenti ad un linguaggio differente. Un trasduttore lavora quindi su due nastri infiniti: il primo nastro in ingresso di sola lettura, il secondo nastro in uscita di sola scrittura.*

Un automa è rappresentato graficamente come un inisieme di circonferenze denominate (che rappresentano gli stati dell'automa), ognuna delle quali è connessa ad altri stati tramite frecce. Su tali frecce è segnata la condizione da soddisfare per intraprendere quel passaggio di stato e come viene modificato lo stato durante il passaggio. Lo stato iniziale si riconosce per via di una freccia entrante nel nodo, mentre gli stati finali sono rappresentati da circonferenze con bordo doppio.

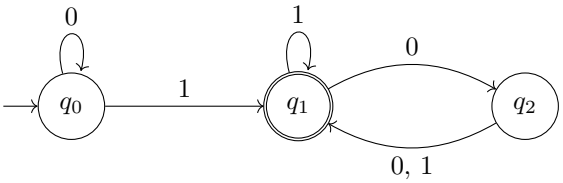


Figura 2.1: Esempio di rappresentazione di un automa a stati finiti

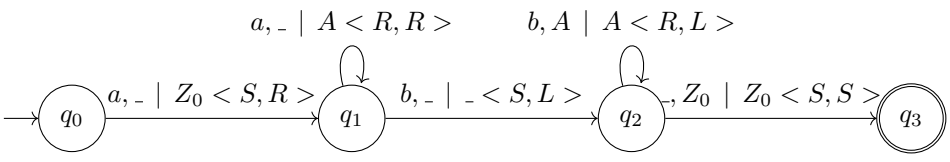


Figura 2.2: Esempio di rappresentazione di una macchina di Turing

2.1 Automi a Stati Finiti

Gli FSA (Finite State Automaton) sono automi che presentano un numero finito di stati . Formalmente:

Definizione 2.1.1. *Un FSA è una tupla di 5 elementi* $< Q, A, \delta, q_0, F >$*, dove:*

- Q *è un insieme finito di stati;*
- $A := \{a_0, a_1, a_2, ..., a_n\}$ *è l'alfabeto di ingresso;*
- $\delta : Q \times A \to Q$*: è la funzione di transizione (eventualmente parziale);*
- $q_0 \in Q$ *è lo stato iniziale;*
- $F \subseteq Q$ *è un insieme di stati finali*

Quando l'automa riceve in ingresso un simbolo del proprio alfabeto, cambia lo stato in cui si trova. Il passaggio da uno stato a quello successivo avviene attraverso la funzione di transizione δ; tale funzione indica, a partire da un determinato stato, in quale stato si troverà l'automa dopo la ricezione di un simbolo in ingresso.

Attenzione! La funzione di transizione δ non sempre è totale: ciò significa che in alcuni automi non tutte le transizioni da uno stato a quello successivo sono definite. Gli automi che presentano una funzione di transizione completa sono detti completi.

Data la funzione di transizione δ è possibile definire una sequenza di mosse δ^*, che opera su una stringa anzichè su un unico simbolo in ingresso. Formalmente:

Definizione 2.1.2. *La chiusura riflessiva e transitiva della funzione di trasferimento, indicata con* $\delta^* : Q \times A^* \to Q$*, è definita in maniera induttiva come segue:*

- $\delta^*(q, \varepsilon) = q$*;*
- $\delta^*(q, yi) = \delta(\delta^*(q, y), i)$

$\forall x : x \in L \Leftrightarrow \delta^*(q_0, x) \in F$.

In altri termini, se l'automa riceve in ingresso la stringa vuota ε, allora rimane nello stato q in cui si trova, mentre nel caso in cui riceve una sequenza di caratteri yi, lo stato che l'automa raggiunge è quello ottenuto con ingresso i, a partire dallo stato che raggiunge da q con la stringa y.

Tramite questa definizione, si può affermare che una generica stringa x appartiene al linguaggio L se e solo se l'automa si trova in uno stato finale a fronte della lettura della stringa in ingresso: in tal caso si dice che l'automa accetta o riconosce la stringa di ingresso. Per questo motivo, gli FSA sono detti essere accettatori, in quanto verificano se una determinata stringa appartiene o meno ad un dato linguaggio.

2.1.1 Trasduttore a Stati Finiti

Gli FST (Finate State Trasducer) sono trasduttori che presentano un numero finito di stati. Formalmente:

Definizione 2.1.3. *Un FST è una tupla di 7 elementi* $< Q, I, \delta, q_0, F, O, \eta >$*, dove:*

- Q *è un insieme finito di stati;*
- I *è l'alfabeto di ingresso;*
- $\delta : Q \times I \to Q$ *è la funzione di transizione (eventualmente parziale);*
- $q_0 \in Q$ *è lo stato iniziale;*
- $F \subseteq Q$ *è un insieme di stati finali*
- O *è l'alfabeto di uscita;*
- $\eta : Q \times I \to O^*$ *è la funzione di uscita (eventualmente parziale).*

Quando l'automa riceve in ingresso un simbolo, modifica il proprio stato e mostra in uscita una sequenza di uno o più simboli. Il passaggio da uno stato a quello successivo avviene attraverso la funzione di transizione δ, mentre l'operazione di uscita avviene tramite la funzione di uscita η.

Attenzione! La funzione di uscita η non sempre è totale: ciò significa che in alcuni automi è possibile che non ci sia nessuna associazione fra un simbolo del linguaggio in ingresso e un simbolo del linguaggio di uscita.

Data la funzione di uscita η, è possibile definire una sequenza di uscita η^*, che opera su una stringa anzichè su un unico simbolo in ingresso. Formalmente:

Definizione 2.1.4. *La chiusura riflessiva e transitiva della funzione di uscita, indicata con $\eta^* : Q \times I^* \rightarrow O^*$, è definita in maniera induttiva come segue:*

- $\eta^*(q, \varepsilon) = \varepsilon$;
- $\eta^*(q, yi) = \eta^*(q, y).\eta(\delta^*(q, y), i)$

$\forall x : \tau(x) = \eta^*(q_0, x) \Leftrightarrow \delta^*(q_0, x) \in F$ *in cui τ rappresenta la funzione di traduzione.*

In altri termini, se l'automa riceve in ingresso la stringa vuota ε, allora rimane nello stato q in cui si trova e non scrive nessun simbolo sul nastro di uscita, mentre, nel caso in cui riceve in ingresso una sequenza di caratteri yi, allora l'automa produce in uscita quello che produce da q leggendo y seguito da ciò che produce dallo stato che raggiunge da q leggendo y, avendo i come ingresso. Quindi la funzione η^* ha il compito di concatenare il simbolo (o la stringa) che si ottiene a fronte della prima transizione con i simboli (o le stringhe) che si ottengono nei successivi passaggi di stato.

Tramite questa definizione, si può affermare che la traduzione di una stringa in ingresso x è accettata se e solo se l'automa si trova in uno stato finale a fronte della lettura di x.

2.1.2 Operazioni sugli Automi a Stati Finiti

Gli Automi a Stati Finiti sono chiusi rispetto alle seguenti operazioni:

- Intersezione. Formalmente, dati:
  - $A^1 =< Q^1, I, \delta^1, q_0^1, F^1 >$
  - $A^2 =< Q^2, I, \delta^2, q_0^2, F^2 >$

  allora $< A^1, A^2 >=< Q^1 \times Q^2, I, \delta, < q_0^1, q_0^2 >, F^1 \times F^2 >$ e si può dimostrare che il linguaggio $L(< A^1, A^2 >) = L(A^1) \cap L(A^2)$
- Unione. Formalmente, dati:
  - $A^1 =< Q^1, I, \delta^1, q_0^1, F^1 >$
  - $A^2 =< Q^2, I, \delta^2, q_0^2, F^2 >$

  Allora $< A^1, A^2 >=< Q^1 \times Q^2, I, \delta, < q_0^1, q_0^2 >, F^1 \times Q^2 \cup Q^1 \times F^2$ con $\delta(< q^1, q^2 >, i) =< \delta^1(q^1, i), \delta^2(q^2, i) >$
- Complementazione. L'idea alla base del ragionamento è che $F^C = Q - F$, quindi per poter complementare un automa a stati finiti è necessario rendere finali gli stati che non lo sono e viceversa. Bisogna però prestare attenzione al caso in cui la funzione di transizione dell'automa sia parziale: in tal caso, è necessario innanzitutto completare l'automa, definendo tutte le transizioni precedentemente non definite, e solo in seguito procedere con lo stesso algoritmo.

2.1.3 Pumping Lemma

Gli automi appena analizzati sono molto semplici e hanno delle evidenti limitazioni, che li rendono inefficaci nella risoluzione di alcuni problemi. Dallo studio di questi automi si possono ricavare alcuni teoremi dimostrabili, utili per mettere in luce tali limitazioni:

Teorema 2.1.1. *Dato un automa a stati finiti $A =< Q, I, \delta, q_0, F >$, dove Q ha cardinalità n, il linguaggio riconosciuto da A non è vuoto se e solo se A accetta una stringa x con $|x| < n$.*

Teorema 2.1.2. *Dato un automa a stati finiti $A =< Q, I, \delta, q_0, F >$, dove Q ha cardinalità n, il linguaggio riconosciuto da A è infinito se e solo se A accetta una stringa x con $n \leq |x| < 2n$.*

I teoremi 2.1.1 e 2.1.2 sono basati sul fatto che un determinato automa a stati finiti può presentare cicli nella sua rappresentazione grafica. Nel caso in cui l'automa non presenta alcun ciclo, il linguaggio è sicuramente finito, in quanto la stringta in ingresso può far passare l'automa una sola volta in ciascuno dei suoi stati: di conseguenza, la stringa può essere lunga al più come il numero di stati dell'automa meno uno.

Si enuncia quindi il seguente teorema noto con il nome di Pumping Lemma:

Teorema 2.1.3 (Pumping Lemma)**.** *Sia A un automa a stati finiti. Se $x \in L$ e $|x| \leq |Q|$, allora esistono uno stato $q \in Q$ e una stringa $w \in I^+$ tali che:*

- $x = ywz$;
- $\delta^*(q, w) = q$

Perciò vale che $\forall n \geq 0 \;\; yw^n z \in L$

Il Pumping Lemma fornisce una condizione necessaria, ma non sufficiente sulla struttura dei linguaggi che vengono riconosciuti da un FSA. In altri termini, il teorema 2.1.3 afferma che se una stringa x è lunga almeno quanto il numero degli stati interni di un determinato automa che accetta tale stringa, allora x passa necessariamente per una sequenza di mosse contenti un ciclo. Inoltre, tutte le stringhe che si ottengono da x ripetendo la sua sottostringa che attraversa il ciclo, sono sequenze riconosciute e accettate dall'automa. Allo stesso modo, sono accettate anche tutte le stringhe che si ottengono da x cancellando qualsiasi sua sottostringa che attraversi tale ciclo.

Ora supponiamo di progettare un automa A che riconosce tutte e sole le stringhe del linguaggio $L = \{a^n b^n : n < 0\}$. Inizialmente si assume che A sia in grado di riconoscere un tale linguaggio. Si considera innanzitutto il caso in cui $x = a^m b^m : m > |Q|$. Sappiamo, grazie al Pumping Lemma, che ci sarà un ciclo interno all'automa e può esistere nei tre casi sottolineati:

- ...aa<u>aaa</u>abbbbbb... : se così fosse si potrebbe rimovere dalla stringa la sottostringa che passa all'interno del ciclo e l'automa riconoscerebbe comunque come accettabile tale stringa, ma ciò non è vero in quanto il numero di a presenti dopo la rimozione di tale sottostringa sarebbe minore del numero delle b e dunque la stringa sarebbe riconoscibile;
- ...aaaaaab<u>bbb</u>bb... : si applica lo stesso ragionamento del punto precedente;
- ..aaa<u>aaabbb</u>bbb... : se così fosse si potrebbe ripercorrere quel determinato ciclo anche due volte, ma così facendo si otterrebbe la stringa ...aaaaaaaabbaabbbbbbbb... che, secondo il lemma dovrebbe essere riconosciuta dall'automa, ma così non è.

Si nota quindi che non esiste nessun automa a stati finiti che riesca ad interpretare un tale linguaggio, in quanto ha una struttura che richiede di contare e ricordare il numero di simboli letti. Per poter contare un numero n arbitrariamente grande si renderebbe necessaria una memoria infinita, che abbia, quindi, un numero infinito di stati.

2.2 Automi a Pila

I PDA (Push Down Automaton) sono automi che presentano un numero finito di stati e una memoria organizzata su una struttura a pila. Tale memoria ha lo stesso comportamento dell'omonima struttura dati: i nuovi simboli vengono inseriti in cima allo stack ed estratti dalla cima, secondo la politica LIFO (Last In First Out). Per indicare che non ci sono più simboli in memoria, si utilizza il simbolo speciale Z_0, detto di fondo pila.

Questi particolari tipi di automi hanno la possibilità di utilizzare il simbolo in cima alla pila per decidere quale transizione effettuare e la possibilità di manipolare la pila, rimuovendone il simbolo alla cima, oppure aggiungendo un nuovo simbolo.

Formalmente:

Definizione 2.2.1. *Un PDA è una tupla di 7 elementi $< Q, I, \delta, \Gamma, q_0, Z_0, F >$, dove:*

- *Q è un inisieme finito di stati;*
- *I è l'alfabeto di ingresso;*
- *Γ è l'alfabeto della pila, tale per cui $\Gamma \cap I = \emptyset$;*
- *$\delta : Q \times (I \cup \{\varepsilon\}) \times \Gamma \to Q \times \Gamma^*$ è la funzione di transizione (eventualmente parziale);*
- *q_0 è lo stato iniziale*
- *Z_0 è il simbolo di fondo pila, ovvero l'unico simbolo che appare inizialmente nella pila;*
- *$F \subseteq Q$ è un insieme di stati finali.*

Dalla definizione, si può osservare che l'automa è in grado di decidere il passo successivo anche senza la lettura di alcun simbolo dal nastro di ingresso, ma solo a fronte della lettura di un simbolo sulla cima dello stack. Tali mosse prendono il nome di ε-mosse o mosse spontanee. Ovviamente, per uno stesso stato, non è possibile definire sia una transizione spontanea che una non spontanea, in quanto l'automa in questione sarebbe non deterministico e non sarebbe possibile definirne il comportamento a priori.

Una mossa spontanea non altera la posizione della testina di lettura sul nastro di ingresso, proprio per il fatto che, in tal caso, l'automa ignora deliberatamente il nastro in questione.

Per gli automi a pila, si può introdurre il concetto di configurazione, ossia una generalizzazione della nozione di stato, definita come segue:

Definizione 2.2.2. *Una configurazione c è una tupla di 3 elementi $< q, x, \gamma >$, dove:*

- *$q \in Q$ è lo stato corrente del dispositivo di controllo;*
- *$x \in I^*$ è la porzione non ancora letta della stringa d'ingresso;*
- *$\gamma \in \Gamma^*$ è la stringa composta da tutti i simboli contenuti all'interno della pila.*

Una configurazione può essere quindi vista come una fotografia dello stato dell'automa in un determinato istante di tempo.

Le transizioni tra configurazioni, indicate con il simbolo $\vdash$ (detto tornello), dipendono naturalmente dalla funzione di transizione δ e si distinguono in due categorie:

1. Se $\delta(q, i, A) =< q', \alpha >$ è una mossa definita, si ha che $c =< q, x, \gamma > \vdash\ c' =< q', x', \gamma' >$, dove:
   - $\gamma = A\beta$, la pila contiene il simbolo A, utilizzato per eseguire la mossa, seguito dalla stringa β composta da tutti i restanti simboli della pila;
   - $x = iy$, la stringa in ingresso è composta da un simbolo i, utilizzato per eseguire la mossa, seguito dalla stringa y composta da tutti i restanti simboli non ancora letti;
   - $\gamma' = \alpha\beta$, dopo la transizione, la pila contiene la stringa α appena inserita, seguita dalla stringa β composta da tutti i restanti simboli della pila;
   - $x' = y$, dopo la transizione, la stringa in ingresso è composta da tutti i simboli non ancora letti; in altre parole, la testina di lettura si sposta a destra di una posizione.
2. Se $\delta(q, \varepsilon, A) =< q', \alpha >$ è una mossa definita, si ha che $c =< q, x, \gamma > \vdash\ c' =< q', x', \gamma' >$. dove:
   - $\gamma = A\beta$, la pila contiene il simbolo A, utilizzato per eseguire la mossa, seguito dalla stringa β composta da tutti i restanti simboli della pila;
   - $\gamma' = \alpha\beta$, dopo la transizione, la pila contiene la stringa α appena inserita, seguita dalla stringa β composta da tutti i restanti simboli della pila;
   - $x' = x$, dopo la transizione, la stringa in ingresso è la medesima; in altre parole, la testina di lettura non si sposta.

Se $\delta(q, \varepsilon, A) \neq \perp$, ovvero se la funzione di transizione è definita, allora $\forall i \in I, \delta(q, i, A) = \perp$: se questa proprietà non fosse soddisfatta, entrambe le transizioni sarebbero consentinte e, come detto precedentemente, l'automa non avrebbe un comportamento deterministico.

Data la funzione di transizione tra configurazioni $\vdash$, si può definire la chiusura transitiva e riflessiva $\vdash^*$ che opera su un insieme di configurazioni anzichè su un'unica configurazione. Un automa a pila accetta una determinata stringa x se c'è un cammino coerente che va dallo stato iniziale ad uno stato finale al termine della lettura della stringa d'ingresso. Formalmente:

$\forall x \in I^*, x \in L \Leftrightarrow c_0 =< q_0, x, Z_0 > \vdash^* c_F =< q \in F, \varepsilon, \gamma >$

Sappiamo quindi che il linguaggio $a^n b^n$ non può essere riconosciuto da alcun FSA per il Pumping Lemma, ma è riconosciuto da un PDA. Inoltre, ogni linguaggio riconosciuto da un FSA è riconoscibile anche da un PDA: si può affermare, quindi, che gli automi a pila sono più espressivi e di maggiore potenza rispetto agli automi a stati finiti, i quali possono essere visti come un sottoinsieme dei PDA.

A differenza degli FSA, i PDA potrebbero anche non terminare la propria esecuzione dopo un numero finito di mosse, in quanto possono presentare cicli di ε-mosse. Tali cicli non aggiungono potere espressivo ai PDA: questo significa che gli automi a pila ciclici riconoscono lo stesso insieme di linguaggi degli automi aciclici. Per questo motivo, si preferisce eliminare tale categoria di automi, utilizzando tranisizioni tra configurazioni del tipo $< q, x, \alpha > \vdash_d^* < q', x, \beta >$, in cui il simbolo $\vdash_d^*$ indica che per $< q, x, \alpha > \vdash^* < q', x, \beta >$ con $\beta = Z\beta'$, $\delta(q', \varepsilon, Z) = \perp$, ovvero transizioni che hanno necessariamente bisogno di un simbolo in ingresso per proseguire la propria esecuzione. Formalmente:

Definizione 2.2.3. *Un PDA si definisce aciclico se e solo se:*
$\forall x \in I^* \; \exists q, \gamma :< q_0, x, Z_0 > \vdash_d^* < q, \varepsilon, \gamma >$.

Un tale automa legge sempre per intero la stringa in ingresso prima di terminare la propria esecuzione ed accettare tale stringa, non potendo finire in un ciclo di ε-mosse. Un automa a pila cicliclo può sempre essere trasformato in un automa a pila aciclico equivalente.

2.2.1 Trasduttori a Pila

I PDT (Push Down Trasducer) sono trasduttori che presentano un numero finiti di stati e una memoria organizzata su una struttura a pila. Formalmente:

Definizione 2.2.4. *Un PDT è una tupla di 9 elementi* $< Q, I, \delta, \Gamma, q_0, Z_0, F, O, \eta >$*, dove:*

- Q *è un inisieme finito di stati;*
- I *è l'alfabeto di ingresso;*
- Γ *è l'alfabeto della pila, tale per cui* $\Gamma \cap I = \emptyset$*;*
- $\delta : Q \times (I \cup \{\varepsilon\}) \times \Gamma \to Q \times \Gamma^*$ *è la funzione di transizione (eventualmente parziale);*
- q_0 *è lo stato iniziale*
- Z_0 *è il simbolo di fondo pila, ovvero l'unico simbolo che appare inizialmente nella pila;*
- $F \subseteq Q$ *è un insieme di stati finali;*
- O *è l'alfabeto di uscita;*
- $\eta : Q \times (I \cup \{\varepsilon\}) \times \Gamma \to O^*$ *è la funzione di uscita (eventualmente paziale).*

Definizione 2.2.5. *Una configurazione c di un PDT è una tupla di 4 elementi* $< q, x, \gamma, z >$*, dove:*

- $q \in Q$ *è lo stato corrente del dispositivo di controllo;*
- $x \in I^*$ *è la porzione non ancora letta della stringa dàingresso;*
- $\gamma \in \Gamma^*$ *è la stringa composta da tutti i simboli contenuti all'interno della pila;*
- $z \in O^*$ *è la stringa presente sul nastro di uscita.*

Le relazioni di transizione $c =< q, x, \gamma, z > \vdash c' =< q', x', \gamma', z' >$ possono essere di due categorie:

1. se $\delta(q, i, A) =< q', \alpha >$ è una mossa definita, si ha che $\eta(q, i, A) = w$, dove:
   - $\gamma = A\beta$, la pila contiene il simbolo A, utilizzato per eseguire la mossa, seguito dalla stringa β composta da tutti irestanti simboli della pila;
   - $x = iy$, la stringa in ingresso è composta da un simbolo i, utilizzato per eseguire la mossa, seguito dalla stringa y composta da tutti i restanti simboli non ancora letti;
   - $\gamma' = \alpha\beta$, dopo la transizione, la pila contiene la stringa α appena inserita, seguita dalla stringa β composta da tutti i restanti simboli delle pila;
   - $x' = y$, dopo la tranisizone, la stringa in ingresso è composta da tutti i simboli non ancora letti; in altre parole, la testina di lettura si sposta a destra di una posizione;
   - $z' = zw$, dopo la tranisizone, alla stringa sul nastro di uscita viene concatenato il simbolo w prodotto dalla funzione di traduzione.
2. Se $\delta(q, \varepsilon, A) =< q', \alpha >$ è una mossa definita, si ha che $\eta(q, \varepsilon, A) = w$, dove:
   - $\gamma = A\beta$, la pila contiene il simbolo A, utilizzato per eseguire la mossa, seguito dalla stringa β composta da tutti i restanti simboli della pila;
   - $\gamma' = \alpha\beta$, dopo la transizione, la pila contiene la stringa α appena inserita, seguita dalla stringa β composta da tutti i restanti simboli della pila;
   - $x' = x$, dopo la transizione, la stringa in ingresso è la medesima; in altre parole, la testina di lettura non si sposta;
   - $z' = zw$, dopo la tranisizone, alla stringa sul nastro di uscita viene concatenato il simbolo w prodotto dalla funzione di traduzione.

La condizione di accettazione della traduzione è quindi la seguente:
$\forall x \in I^* : x \in L \wedge z = \eta(x) \iff c_0 =< q_0, x, Z_0, \varepsilon >\vdash^* c_F =< q \in F, \varepsilon, \gamma, z >$.

2.2.2 Operazioni sugli Automi a Pila

Gli Automi a Pila sono chiusi solamente rispetto all'operazione di complementazione. Ogni PDA può essere complementato seguendo le operazioni qui elencate:

1. Eliminazione dei cicli;
2. Aggiunta di stati di errore;
3. Eliminazione delle mosse spontanee che collegano stati non finali a stati finali;
4. Scambio di stati finali e non finali.

I PDA non sono aperti rispetto ad alcun'altra operazione, quindi un automa a pila non riesce a riconoscere i linguaggi generati dall'intersezione o l'unione di più linguaggi. Inoltre, gli automi a pila presentano ulteriori limitazioni dovute al fatto che lo stack di memoria sia di tipo distruttivo: per poter leggere le informazioni contenute in memoria è prima necessario eliminare tutte le altre informazioni che si trova al di sopra di essa. Quindi, linguaggi del tipo $L = \{a^n b^n c^n \mid n \geq 1\}$ non possono essere riconosciuti da normali PDA perchè una volta terminata la lettura delle b e, quindi, una volta eliminati tutti i simboli sulla pila per controllare che il numero di b sia uguale al numero di a, non rimane più alcuna informazione per poter verificare che il numero di c sia uguale al numero delle altre lettere.

2.3 Macchine di Turing

Le TM (Turing Machine) sono automi che presentano un numero finito di stati e che operano su un insieme di nastri infiniti a destra: un nastro di ingresso, un nastro di uscita e uno o più nastri di memoria. Ogni nastro è composto da celle inizializzate dal simbolo di blank (tramite cui si indica che la cella è vuota) che può essere sovrascritto con un qualsiasi simbolo dell'alfabeto in ingresso. Il dispositivo di controllo può muovere le testine di lettura dei nastri in maniera indipendente l'una dall'altra, spostandole a destra (R) o a sinistra (L) di una cella, oppure lasciandole ferme (S) nella posizione in cui si trovano. L'unica

eccezione è costituita dal nastro di uscita che può essere spostato solamente a destra. In alternativa, l'automa può anche non effettuare alcuna operazione, fermandosi definitivamente.

Formalmente:

Definizione 2.3.1. *Una TM a k nastri è una tupla di 9 elementi* $< Q, I, \Gamma, O, \delta, \eta, q_0, Z_0, F >$*, dove:*

- Q *è un insieme finito di stati;*
- I *è l'alfabeto di ingresso;*
- Γ *è l'alfabeto di memoria;*
- O *è l'alfabeto di uscita;*
- $F \subseteq Q$ *è l'insieme di stati finali;*
- $q_0 \in Q$ *è lo stato iniziale;*
- $Z_0 \in \Gamma$ *è il simbolo iniziale dell'alfabeto di memoria;*
- $\delta : (Q - F) \times I \times \Gamma^k \to Q \times \Gamma^k \times \{R, L, S\}^{k+1}$ *è la funzione di transizione (eventualmente parziale);*
- $\eta : (Q - F) \times I \times \Gamma^k \to O \times \{R, S\}$ *è la funzione di uscita (eventualmente parziale), definita dove è definita la funzione di transizione* δ*;*

Si noti che la funzione di transizione δ è definita in modo tale che non ci siano transizioni uscnti da uno stato finale.

Definizione 2.3.2. *Una configurazione c di una TM a k nastri è una tupla di k+3 elementi* $< q,\ x\uparrow iy,\ \alpha_1 \uparrow A_1\beta_1, ..., \alpha_k \uparrow A_k\beta_k,\ u\uparrow o >$*, dove:*

- $q \in Q$ *è lo stato attuale in cui si trova l'automa;*
- $x\uparrow iy$ *è la posizione in cui si trova la testina di lettura del nastro di ingresso, con* $x, y \in I^*$ *e* $i \in I$*;*
- $\alpha_1 \uparrow A_1\beta_1$ *è la posizione in cui si trova la testina di lettura del primo nastro di memoria, con* $\alpha_1, \beta_1 \in \Gamma^*$ *e* $A_1 \in \Gamma$*.*
- $\alpha_k \uparrow A_k\beta_k$ *è la posizione in cui si trova la testina di lettura del k-esimo nastro di memoria, con* $\alpha_k, \beta_k \in \Gamma^*$ *e* $A_k \in \Gamma$*.*
- $u\uparrow o$ *è la posizione in cui si trova la testina di lettura del nastro di uscita, con* $u \in O^*$ *e* $o \in O$*;*
- $\uparrow \notin I \cup \Gamma \cup O$*, rappresenta la posizione della testina di lettura di un determinato nastro; la testina di lettura punta la cella in cui è contenuto il carattere immediatamente alla destra del simbolo* $\uparrow$*.*

La configurazione iniziale c_0 *di una TM a k nastri è una tupla di k+3 elementi* $< q_0, \uparrow iy, \uparrow Z_0, ..., \uparrow Z_0, \uparrow \_ >$*. Quindi,* $x = \varepsilon, A_1, ..., A_k = Z_0, \alpha_1, ..., \alpha_k = \varepsilon, \beta_1, ...\beta_k = \varepsilon, u = \varepsilon$ *e* $o = \_$*.*

Ora, data la definizione di configurazione, è necessario anche formalizzare il significato di transizione $\vdash$ tra due date configurazioni c e c' (detta anche mossa o passo computazionale):

Definizione 2.3.3. *Siano:*
$c =< q, x \uparrow iy, \alpha_1 \uparrow A_1\beta_1, ..., \alpha_k \uparrow A_k\beta_k, u \uparrow o >$*, con* $x = \underline{xi}, y = \underline{jy}, \alpha_1 = \underline{\alpha_1 A_1}, ..., \alpha_k = \underline{\alpha_k A_k}$ *e* $\beta_1 = \underline{\beta_1 B_1}, ..., \beta_k = \underline{\beta_k B_k}$*,* $c' =< q', x'\uparrow i'y', \alpha'_1\uparrow A'_1\beta'_1, ..., \alpha'_k\uparrow A'_k\beta'_k, u'\uparrow o' >$*,*
$\delta(q, i, \overline{A_1}, ..., A_k) =< p, C_1, ..., C_k, N, N_1, ...N_k >$*, con* $p \in Q, N, N_1, ..., N_k \in \{R, L, S\}$ *e* $C_1, ..., C_k \in \Gamma$*,*
e $\eta(q, i, A_1, ..., A_k) =< v, M >$*, con* $v \in O$ *e* $M \in \{R, S\}$*.*
Allora $c \vdash c'$ *se e solo se:*

1. *Vale che* $p = q'$*;*
2. *Vale anche una delle seguenti condizioni:*
   (a) *x=x', i=i', y=y' ed N=S;*
   (b) *x'=xi, i'=*$\underline{j}$*, y'=*$\underline{y}$ *ed N=R;*
   (c) *x'=*$\underline{x}$*, i'=i, y'=iy ed N=L;*

3. *Vale anche una delle seguenti condizioni, per* $1 \leq r \leq k$*:*

   (a) $\alpha'_r = \alpha_r, A'_r = C_r, \beta'_r = \beta_r$, *ed* $N_r = S$*;*

   (b) $\alpha'_r = \alpha_r C_r, A'_r = B_r, \beta'_r = \underline{\beta_r}$, *ed* $N_r = R$*; se* $\beta_r = \varepsilon$*, allora* $A'_r = \_$ *e* $\beta'_r = \varepsilon$*;*

   (c) $\alpha'_r = \underline{\alpha_r}, A'_r = \underline{A_r}, \beta'_r = C_r \beta_r$ *ed* $N_r = L$*;*

4. *Vale anche una delle seguenti condizioni:*

   (a) *u'=u, o'=v ed M=S;*

   (b) *u'=uv, o=_ ed M=R*

Se nei casi 2c e 3c, $x = \varepsilon$ *oppure* $\alpha = \varepsilon$*, allora non esiste una configurazione c' tale che* $c \vdash c'$ *e la macchina di Turing termina la propria esecuzione. Lo stesso comportamento si ottiene nel caso in cui le funzioni* δ *ed* η *non sono definite in* $< 1, i, A_1, ..., A_k >$*. In tali casi, c viene anche definita configurazione di arresto.*

In altre parole, se $\delta(q, i, A_1, ..., A_k) =< p, C_1, ..., C_k, N, N_1, ..., N_k >$, la condizione 1 vincola lo stato di x' ad essere quello di arrivo della transizione. Le condizioni di tipo 2 definiscono l'evoluzione del nastro di ingresso al passaggio da c a c': se la testina rimane ferma (condizione 2a), le tre parti in cui la testina divide il nastro saranno identiche in c e c'; se la testina si muove a destra (condizione 2b), la parte a sinistra della testina conterrà anche il simbolo corrente di c, il simbolo corrente di c' sarà il primo simbolo della parte destra in c e la rimanente parte destra di c sarà la parte destra di c'; infine, se la testina si muove a sinistra (condizione 2c), la parte a sinistra della testina in c' conterrà tutti i simboli della parte sinistra in c, tranne l'ultimo che diventerà il simbolo corrente di c' e la parte destra di c' sarà composto dal simbolo corrente in c seguito dalla sua parte destra. Analogamente le condizioni di tipo 3 specificano l'evoluzione dei nestri di memoria e le condizioni 4 quella del nastro di uscita.

Si introduce di seguito il teorema tramite cui si afferma la supremazia delle macchine di Turing nella computazione e nella traduzione.

Teorema 2.3.1. *La classe di linguaggi riconosciuti dalle Macchine di turing include strettamente la classe dei linguaggi riconosciuti dagli Automi a Pila. Inoltre, le macchine di Turing sono trasduttori più potenti rispetto ai trasduttori a pila, per cui tutte le traduzioni effettuate da un PDT possono essere effettuate anche da una TM trasduttrice, ma non viceversa.*

Le macchine di Turing sono il formalismo più potente di cui siamo a disposizione.

2.3.1 Macchine di Turing come Accettatori

Quando le macchine di Turing vengono impiegate per il riconoscimento di un determinato linguaggio o per definire nuovi linguaggi, il nastro di uscita viene completamente ignorato. Dunque, una TM utilizzata come accettatore è una tupla di 7 elementi, ottenuta eliminando dalla definizione 2.3.1 gli elementi O (l'alfabeto di uscita) ed η (la funzione di uscita). Di conseguenza, anche la definizione 2.3.2 di configurazione viene modificata omettendo la rappresentazione del nastro di uscita.

Definizione 2.3.4. *Sia M una TM a k nastri. Una stringa* $x \in I^*$ *è accettata da M se e solo se:*
$c_0 =< q_0, \uparrow x, \uparrow Z_0, ..., \uparrow Z_0 > \vdash^* c_F < q \in F, x' \uparrow iy, \ \alpha_1 \uparrow A_1 \beta_1, ..., \alpha_k \uparrow A_k \beta_k >$.

In altre parole, il linguaggio riconosciuto da una TM M è composto da tutte e sole le stringhe che permettono di andare dallo stato iniziale ad uno stato finale. Inoltre, non è richiesto che al termine della computazione la testina si trovi al termine della stringa in ingresso: la TM M può raggiungere uno stato finale anche senza aver completamente letto la stringa in ingresso.

Se $x \notin L$, M può anche non raggiungere mai una configurazione di arresto e continuare la sua esecuzione per un tempo indefinito.

2.3.2 Macchine di Turing come Trasduttori

Quando le macchine di Turing vengono impiegate per la traduzione di linguaggi, ovvero per mappare stringhe di I^* in stringhe di O^*, il nastro di uscita non viene più ignorato come nel caso degli accettatori. Dunque, una TM utilizzata come trasduttore è una tupla di 9 elementi, esattamente come presentata nella definizione 2.3.1.

Definizione 2.3.5. *Sia M una MT a k nastri. M definisce una traduzione $\tau(x) = y$, con $\tau_M : I^* \to O^*$, se e solo se:*

$c_0 =< q_0,\uparrow x,\uparrow Z_0, ...,\uparrow Z_0,\uparrow_- >\vdash^* c_F =< q \in F, x'\uparrow iy,\ \alpha_1\uparrow A_1\beta_1, ..., \alpha_k\uparrow A_k\beta_k,\ y\uparrow_- >$

In altre parole, una stringa x viene tradotta in una stringa y da una TM M se esiste un cammino che parte da una configurazione iniziale con x sul nastro di ingresso e termina in una configurazione finale con y sul nastro di uscita.

2.3.3 Altri Modelli di Macchine di Turing

Il modello originario di automa, pensato da Alan Turing (padre fondatore dell'odierna informatica), era una macchina, simile a quelle analizzate nelle sezioni precedenti, con l'unica differenza che presentava un solo nastro infinito (sia a destra che a sinistra), utilizzato come ingresso, uscita e memoria. Formalmente:

Definizione 2.3.6. *Una TM a nastro singolo è una tupla di 5 elementi $< Q, A, \delta, q_0, F >$, dove:*

- *Q è un insieme finito di stati;*
- *A è un alfabeto finito caratteristico del nastro;*
- *$q_0 \in Q$ è lo stato iniziale;*
- *$F \subseteq Q$ è l'insieme di stati finali;*
- *$\delta : (Q - F) \times A \to Q \times A \times \{R, L, S\}$ è la funzione di transizione (eventualmente parziale).*

Si noti che, le macchine di Turing ad 1 nastro sono un modello di automa differente rispetto ad una macchina di Turing a singolo nastro: nel primo caso, la TM presenta ben tre nastri (uno di ingresso, uno di uscita e uno di memoria), mentre nel secondo caso, la TM presenta un unico nastro.

Teorema 2.3.2. *Le TM multinastro e le TM a nastro singolo sono formalismi equivalenti, ossia accettano la stessa classe di linguaggi e realizzano la stessa classe di traduzioni.*

Ognuno dei modelli di macchine di Turing precedentemente analizzati possono essere dotati di nastri multidimensionali. Un nastro k-dimensionale è realizzato in modo che ogni cella sia identificata univocamente da una tupla di k interi positivi. Anche questo modello di macchina di Turing è equivalente agli altri analizzati, in quanto si può stabilire una corrispondenza biiettiva fra $\mathbb{N} - \{0\}$ e $(\mathbb{N} - \{0\})^k$, che indicano rispettivamente l'insieme delle posizioni della testina per i nastri lineari e l'insieme delle posizioni della testina per i nastri multidimensionali. Un esempio importante è l'enumerazione delle celle del nastro bidimensionale, che si ottiene tramite la formula:

$$d(x, y) = x + \frac{(x + y)(x + y - 1)}{2}$$

ottenuta con il metodo della diagonalizzazione.

Capitolo 3

Automi non Deterministici

Tutti i modelli analizzati fino ad ora sono deterministici, nel senso che una volta fissato lo stato iniziale e l'ingresso, il comportamento di tale automa è univocamente determinato. In altri termini, per ogni stato e per ogni ingresso è sempre possibile determinare a priori lo stato in cui l'automa termina la propria esecuzione.

Ci sono però casi in cui questo non è possibile, in quanto non si ha una conoscenza sufficientemente accurata del comportamento dell'automa per poter prevedere l'esatta evoluzione del modello. In tal caso si dice che l'automa è non deterministico.

3.1 Automi a Stati Finiti non Deterministici

Gli NFSA (Nondeterministic Finite State Automaton) sono automi che presentano un numero finito di stati. Tali automi sono definiti come i corrispettivi automi deterministici, con l'unica differenza che presentano una funzione di transizione definita nel seguente modo:

$\delta : Q \times I \to \wp(Q)$ [1]

Di conseguenza, la chiusura riflessiva e transitiva di tale funzione, si definisce induttivamente nel seguente modo:

$$\delta^*(q, \varepsilon) = \{q\},\ \forall q \qquad\qquad \delta^*(q, xi) = \bigcup_{q' \in \delta^*(q,x)} \delta(q', i)$$

Nel caso di accettatori, $x \in I^*$ è accettata da un NFSA $< Q, I, \delta, q_0, F >$ se e solo se $\delta^*(q_0, x) \cap F = \emptyset$.

In altre parole, un NFSA può presentare diverse sequenze di transizioni per ogni dato stato e per ogni data sequenza di ingresso, quindi la chiusura riflessiva e transitiva della funzione δ non rappresenta più un singolo cammino coerente con la stringa di ingresso, ma un insieme di cammini possibili. Questo porta anche alla necessità di dover modificare la condizione di accettazione di una determinata stringa in ingresso: intuitivamente, infatti, tale nastro è accettato dall'automa se e solo se almeno una delle possibili sequenze di transizioni conduce a uno stato finale.

Gli automi non deterministici a stati finiti hanno la stessa potenza di calcolo dei corrispettivi automi deterministici, ma sono spesso più convenienti da utilizzare. Da qui il seguente teorema:

Teorema 3.1.1. *Per ogni NFSA A, può essere costruito un FSA A_D deterministico che accetti lo stesso linguaggio.*

Infatti, dato un NFSA, si può costruire un FSA equivalente che ha come stati gli insiemi formati da stati dell'NFSA. La funzione di transizione è costruita in modo che se un insieme di stati è raggiungibile a partire da uno stato dell'NFSA, allora tale relazione deve essere presente anche nell'FSA sfruttando la costruzione degli stati come insiemi di stati dell'NFSA.

[1] $\wp(Q)$ rappresenta l'insieme delle parti di Q, i cui elementi sono insiemi di stati.

3.2 Automi a Pila non deterministici

Un NPDA (Nondeterministic Push Down Automaton) è un automa che presenta un numero finito di stati e una memoria organizzata su una struttura a pila. Tali automi sono definiti come i corrispettivi automi deterministici, con l'unica differenza che presentano una funzione di tranisizione definita nel seguente modo:

$\delta : Q \times (I \cup \{\varepsilon\}) \times \Gamma \to \wp_F(Q \times \Gamma^*)$

La relazione di transizione fra configurazioni su $Q \times I^* \times \Gamma^*$ è definita da $< q, x, \gamma > \vdash < q', x', \gamma' >$ se e solo se vale una delle seguenti condizioni:

1. $x = ay, x' = y, \gamma = A\beta, \gamma' = \alpha\beta, < q', \alpha > \in \delta(q, a, A)$;
2. $x = x', \gamma = A\beta, \gamma' = \alpha\beta, < q', \alpha > \in \delta(q, \varepsilon, A)$.

Inoltre, la stringa $x \in I^*$ è accettata dall'automa se e solo se:
$< q_0, x, Z_0 > \vdash^* < q \in F, \varepsilon, \gamma \in \Gamma^* >$.

Si noti che i PDA sono automi intrinsecamente non deterministici: infatti, nella prima definizione di automa a pila si era dovuto aggiungere il vincolo che, se per una determinata condizione era definita una ε-mossa, allora non era definita nessun'altra transizione che partisse da quello stato. Nella definizione appena data, invece, questo vincolo viene rimosso.

3.3 Macchine di Turing non deterministiche

Una NTM (Nondeterministic Turing Machine) è un automa che presenta un numero finito di stati e che opera su un insieme di nastri infiniti a destra. Tali automi sono definiti come i corrispettivi automi deterministici, con l'unica differenza che presentano una funzione di transizione e di accettazione definita nel seguente modo:

$< \delta, \eta >: (Q - F) \times I \times \Gamma^k \to \wp(Q \times \Gamma^k \times \{R, L, S\}^{k+1} \times \{R, S\})$

mentre per una NTM a nastro singolo, la funzione di transizione è definita nel seguente modo:

$\delta : (Q - F) \times A \to \wp(Q \times A \times \{R, L, S\})$

Teorema 3.3.1. *Le macchine di Turing non deterministiche non sono più potenti delle corrispettive macchine di Turing deterministiche se utilizzate come riconoscitori di linguaggi.*

Data una qualsiasi NTM M, è sempre possibile costruire una TM deterministica M' che riconosce lo stesso linguaggio di M. Se si considera una computazione di M su una stringa in ingresso, questa è ben definita da un albero di configurazioni, in cui è inserita ogni configurazione raggiungibile dallo stato iniziale. Una stringa viene accettata solo se esiste almeno un cammino all'interno della struttura ad albero che si conclude in una configurazione finale. M' potrà simulare il comportamento della TM M, ricostruendo in maniera sequenziale tutte le possibili computazioni di M. Si noti però che quando una stringa non viene accettata, la macchina di Turing M potrebbe entrare in un ciclo infinito, senza mai terminare la propria esecuzione: per questo motivo, l'albero delle computazioni potrebbe presentare rami infiniti. La TM M', che simula M, deve quindi evitare di visitare l'albero delle computazioni 'in profondità' (ovvero tramite un algoritmo di depth first search), ossia seguendo un percorso fino al suo termine, prima di passare ad un ramo successivo, in quanto alcuni rami sono, appunto, infiniti. Conviene quindi visitare l'albero delle computazioni 'in ampiezza' (ovvero tramite un algoritmo di breadth first search), ossia seguendo tutti i nodi dei rami sullo stesso livello. In questo modo, se esiste un cammino dell'albero di M che porti all'accettazione della stringa, M' riuscirà a trovarlo in un tempo finito, terminando la propria esecuzione. Altrimenti, se tutti i cammini di M terminano in stati non finali, M' terminerà la propria esecuzione senza aver trovato nessun nodo di accettazione e rifiuterà la stringa in ingresso.

Capitolo 4

Grammatiche

Come visto in precedenza, spesso gli automi vengono utilizzati come modelli per il riconoscimento di linguaggi. Gli automi sono quindi uno strumento formale per la descrizione e la definizione di un determinato linguaggio, costituito dall'insieme delle stringhe accettate dall'automa stesso.

Un altro formalismo utilizzato per la definizione di un linguaggio sono le cosiddette grammatiche formali, che froniscono il procedimento mediante cui si ottengono le stringhe appartenenti al linguaggio stesso.

4.1 Introduzione

Una grammatica formale è un insieme di regole per costruire stringhe appartenenti ad un determinato linguaggio, attraverso il meccanismo di riscrittura, che consiste in un insieme di tecniche che determinano come sostituire le parti di una formula con parti più semplificate. In generale, un meccanismo di riscrittura consiste in un insieme di regole linguistiche, di cui una descrive l'oggetto principale come una sequenza di componenti. Ogni componente si può raffinare da elementi via via sempre più dettagliati, fino ad ottenere una sequenza di componenti elementari.

Una grammatica non è altro che un meccanismo linguistico, composto dall'oggetto principale, detto anche simbolo iniziale, da un insieme di componenti, a loro volta da sostituire durante il processo di derivazione, detti anche simboli non terminali, un insieme di elementi di base, detti anche simboli elementari, e da un insieme di regole di raffinamento o sostituzioni, chiamate produzioni.

Definizione 4.1.1. *Una grammatica G è una tupla di 4 elementi $G =< V_T, V_N, P, S >$, dove:*

- *V_T è un insieme di simboli terminali (solitamente indicati con lettere minuscole), detto anche alfabeto terminale;*
- *V_N è un insieme di simboli non terminali (solitamente indicati con lettere maiuscole), tali che $V_T \cap V_N = \emptyset$, detto anche alfabeto non terminale; V indica $V_T \cup V_N$;*
- *P è un insieme finito di $V_N^+ \times V^*$, detto anche insieme delle produzioni di G. Un elemento $p =< \alpha, \beta >\in P$ si indica con $\alpha \to \beta$, in cui α è la parte sinistra di p, mentre β è la parte destra di p;*
- *S è un elemento particolare di V_N, detto assioma o simbolo iniziale.*

Quindi, un elemento che deve essere ancora raffinato è un simbolo non terminale, un elemento di base è un simbolo terminale, le componenti di un oggetto possono essere sia simboli terminali che non terminali, mentre una produzione corrisponde ad una regola di raffinamento.

Definizione 4.1.2. *Data una grammatica G, si definisce su V^* la relazione binaria di derivazione immediata, indicata con il simbolo $\Rightarrow$ da α a β. Tale relazione sussiste se e solo se $\alpha = \alpha_1\gamma\alpha_2, \beta = \alpha_1\delta\alpha_2$, con $\alpha_1, \alpha_2, \delta \in V^*, \gamma \in V_N^+, \gamma \to \delta \in P$.*

Data la definizione di derivazione immediata, si può anche definire la chiusura riflessiva e transitiva, indicata con il simbolo $\Rightarrow^*$, che opera su una serie di stringhe (di simboli elementari o non elementari), anzichè che su una sola stringa.

Date le precedenti definizioni, si può ora definire il linguaggio generato da una grammatica, tramite la seguente definizione:

Definizione 4.1.3. *Data una grammatica G, il linguaggio L(G) generato da G è definito come:*
$L(G) = \{x \mid S \Rightarrow^* x, x \in V_T^*\}$

Quindi il linguaggio generato da una grammatica è costituito da tutte e sole le stringhe di simboli terminali, derivati a partire dall'assioma S, applicando un numero qualsiasi di sostituzioni.

4.2 Classificazione

Una volta definite cosa siano le grammatiche, è possibile classificarle in base alle loro proprietà e in base alla forma ammessa per le produzioni. Tale classificazione viene anche detta Gerarchia di Chomsky, tramite cui si dividono le grammatiche in quattro categorie:

- Grammatiche di tipo 0 (non ristrette): sono grammatiche definite come nella 4.1.1, ovvero grammatiche che non possiedono nessuna restrizione nel tipo di produzione;

- Grammatiche di tipo 1 (sensibili al contesto): sono grammatiche a cui si introduce il vincolo per cui le produzioni possono essere solo nella forma $\alpha A\beta \to \alpha\gamma\beta$, dove $\alpha, \beta, \gamma \in V$ e $A \in V_N$, con $\gamma \neq \varepsilon$; inoltre, la derivazione $S \to \varepsilon$ è consentita solo se S non appare a destra in nessuna regola di derivazione;

- Grammatiche di tipo 2 (non contestuali): sono grammatiche a cui si introduce il vincolo per cui ad ogni produzione $\alpha \to \beta \in P$ si verifica che $|\alpha| = 1$ (quindi $\alpha \in V_N$) e $\beta \in V^*$;

- Grammatiche di tipo 3 (regolari): sono grammatiche a cui si introduce il vincolo per cui ad ogni produzione $\alpha \to \beta \in P$ si verifica che $|\alpha| = 1$ (quindi $\alpha \in V_N$) e che β sia in una sola delle seguenti forme: aB, Ba, a oppure ε, con $a \in V_T$ e $B \in V_N$; inoltre, la derivazione $S \to \varepsilon$ è consentita solo se S non appare a destra in nessuna regola di derivazione;

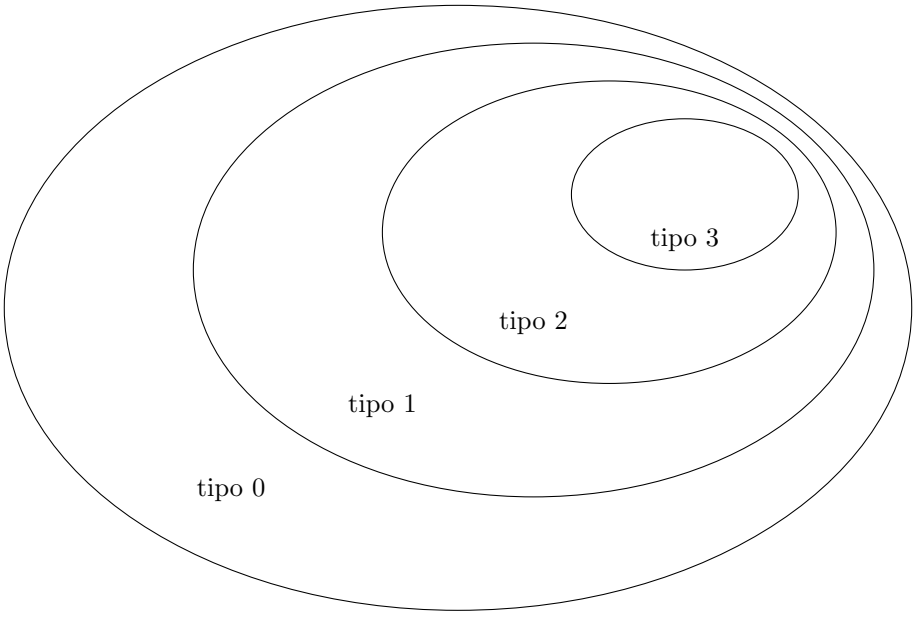


Figura 4.1: Gerarchia di Chomsky

4.3 Grammatiche e Automi

Studiando le grammatiche e i linguaggi da esse generate, si può osservare una certa corrispondenza con gli automi analizzati nei capitoli precedenti. Si introducono qui alcuni teoremi che mettono in luce la correlazione esistente fra automi e grammatiche.

Teorema 4.3.1. *Dato un FSA A, è possibile costruire una grammatica regolare (di tipo 3) G ad esso equivalente, ossia in grado di riconoscere lo stesso linguaggio riconosciuto da A, e viceversa. Dunque, le grammatiche regolari e gli automi a stati finiti sono modelli differenti per descrivere la stessa classe di linguaggi.*

Dato un FSA $A =< I, \delta, q_0, F >$, si può costruire una grammatica $G =< V_N, V_T, P, S >$ regolare, tale che:

- $V_N = Q$;
- $V_T = I$;
- $S = q_0$;
- $\forall B \to bC \Leftrightarrow C \in \delta(B, b)$;
- $\forall B \to \varepsilon, B \in F$

Data una grammatica $G =< V_N, V_T, P, S >$ regolare, si può costruire un FSA $A =< I, \delta, q_0, F >$, tale che:

- $Q = V_N \cup \{q_F\}$;
- $I = V_T$;
- $q_0 = S$;
- $F = \{q_F\}$
- $\forall A \to bC, C \in \delta(A, B)$;
- $\forall A \to b, q_F \in \delta(A, b)$

In generale, l'automa a stati finiti A, ottenuto a partire dalla grammatica regolare G, è non deterministico.

Teorema 4.3.2. *Dato un NPDA A è possibile costruire una grammatica G non contestuale (di tipo 2) ad esso equivalente, ossia in grado di riconoscere lo stesso linguaggio riconosciuto da A, e viceversa. Dunque, le grammatiche non contestuali e gli automi a pila non deterministici sono modelli differenti per descrivere la stessa classe di linguaggi.*

Teorema 4.3.3. *Data una TM M utilizzata come accettatore di linguaggi è possibile costruire una grammatica generale G (di tipo 0) ad essa equivalente, ossia in grado di riconoscere lo stesso linguaggio riconosciuto da M, e viceversa. Dunque, le grammatiche non ristrette e le macchine di Turing sono modelli differenti per descrivere la stessa classe di linguaggi.*

4.4 Espressioni Regolari

Un'espressione regolare è una sequenza di simboli utilizzabile per denotare un linguaggio attraverso la struttura delle stringhe che lo compongono.

Definizione 4.4.1. *Dato un alfabeto di simboli terminali denotato con V_T, si definiscono su di esso le espressioni regolari e i corrispondenti linguaggi denotati:*

- *$\emptyset$ è un'espressione regolare che denota il linguaggio vuoto;*
- *$\forall a \in V_T, a$ è un'espressione regolare che denota il linguaggio formato solo dal simbolo a;*
- *Se R_1 ed R_2 sono espressioni regolari, anche la loro unione, indicata con $R_1 + R_2$ o $R_1 \mid R_2$, è un'espressione regolare;*

- *Se R_1 ed R_2 sono espressioni regolari, anche la loro concatenazione, indicata con $R_1 \cdot R_2$, è un'espressione regolare;*
- *Se R è un'espressione regolare, anche la stella di Kleene di R, indicata con R^*, è un'espressione regolare.*

Nessun'altra stringa è un'espressione regolare.

Gli operatori $|, \cdot, ^*$ definiti per le espressioni regolari, hanno un implicito ordine di applicazione, se non indicato diversamente dall'uso delle parentesi. In particolare, * ha la precedenza rispetto a $\cdot$, che ha a sua volta la precedenza su $|$.

Inoltre, vale anche il seguente teorema:

Teorema 4.4.1. *La classe dei linguaggi denotati dalle espressioni regolari coincide con la classe dei linguaggi regolari.*

4.5 Pattern

Definizione 4.5.1. *Un sistema di pattern è una tripla $< A, V, p >$, dove:*

- *A è un alfabeto;*
- *V è un insieme di variabili tale che $A \cap V = \emptyset$;*
- *p è una stringa su $A \cup V$ detta pattern.*

Il linguaggio generato dal sistema di pattern consiste in tutte le stringhe su A ottenute da p sostituendo ogni variabile in p con una stringa su A.

ESEMPIO. Il pattern $< 0, 1, v_1, v_2, v_1 v_1 0 v_2 >$ rappresenta il linguaggio composto dalle stringhe che iniziano con 0 (nel caso in cui $v_1 = \varepsilon$) oppure che iniziano con una qualunque stringa sull'alfabeto A (in questo caso binario) ripetuta due volte, seguita da uno 0 e terminano con una qualunque altra stringa dell'alfabeto A (inclusa ε).

Le espressioni regolari seguono la stessa idea dei sistemi di pattern, ma hanno un potere espressivo differente: nello specifico è possibile esprimere certi linguaggi con i pattern, ma non con le espressioni regolari, e viceversa, è possibile esprimere alcuni linguaggi con le espressioni regolari, ma non con i pattern. Per tale motivo, i due modelli si dicono essere non confrontabili

4.6 Riepilogo

Tabella 4.1: Relazione fra grammatiche, linguaggi e automi

Gerarchia	Grammatiche	Linguaggi	Automa minimo
tipo 0	Generali	Ricorsivamente enumerabili	TM
tipo 1	Dipendenti dal contesto	Dipendenti dal contesto	LBA*
tipo 2	Non contestuali	Non contestuali	NPDA
tipo 3	Regolari	Regolari	FSA

* Gli LBA (Linear Bounded Automata) sono un particolare tipo di macchina di Turing non deterministica in cui la lunghezza del nastro è funzione lineare della dimensione della stringa in ingresso. Tali automi non sono stati trattati in questo documento.

Capitolo 5

Logica

Come già visto più volte, i linguaggi possono essere rappresentati in modi differenti, tramite modelli e astrazioni via via più potenti, tra cui:

- Insiemi;
- Pattern;
- Espressioni Regolari;
- Modelli Operazionali (come gli Automi);
- Modelli Generativi (come le Grammatiche);
- Modelli Dichiarativi (come la Logica).

I concetti di logica introdotti di seguito sono utilizzati per definire in maniera differente i linguaggi.

5.1 Logica Proposizionale

Il calcolo proposizionale, detto anche logica proposizionale, è un modello dichiarativo formale della logica matematica che si basa sul concetto di proposizione, ovvero frasi che possono assumere solamente il valore vero o il valore falso. In generale, ogni linguaggio consiste di una sintassi e di una semantica: la sintassi è l'insieme delle regole attraverso cui è possibile costruire le frasi di cui il linguaggio è composto, mentre la semantica spiega il significato delle varie frasi del linguaggio.

Sintassi In modo formale:

Definizione 5.1.1. *La logica proprosizionale è composta da un linguaggio $\mathcal{L}$, il cui alfabeto è costituito dai seguenti elementi:*

1. *Un insieme numerabile (finito o non) di proposizioni (simboli di relazione nullaria), che possono essere simboli, stringhe o frasi;*
2. *Un insieme di connettivi logici: $\neg$ (NOT), $\wedge$ (AND), $\vee$ (OR), $\Rightarrow$ (Implicazione) e $\Leftrightarrow$(Coimplicazione);*
3. *Un insieme di simboli di punteggiatura: (e)*

I simboli che compongono l'alfabeto sono privi di significato: assegnarne uno è compito della semantica.

Una proposizione si dice essere atomica quando non può essere scomposta in parti più piccole. Nel caso contrario, la proposizione è composta da due o più proposizioni più piccole legate fra loro tramite i connettivi logici appena introdotti. Le parentesi sono utilizzare per modificare la precedenza dei connettivi, che di base avrebbero il seguente ordine logico: $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$.

La sintassi del linguaggio definisce le sequenze ammissibili di simboli sull'alfabeto, le cosiddette formule ben formate (fbf). L'insieme di queste formule ben definite su $\mathcal{L}$ è il più piccolo insieme tale che ogni proposizione è una formula e, se F e G sono formule, allora anche $\neg F$, $F \wedge G$, $F \vee G$, $F \Rightarrow G$ $F \Leftrightarrow G$ sono formule. In logica proposizionale si ha che se A è una proposizione, allora A e $\neg A$ sono letterali, in cui A è letterale positivo, mentre $\neg A$ è letterale negativo. Infine, si dice letterale complementare la proposizione $\bar{L}$ definito come $\neg A$ se $L = A$, oppure A se $L = \neg A$.

Tabella 5.1: Tabella della verità dei connettivi logici

F	G	$\neg F$	$F \wedge G$	$F \vee G$	$F \Rightarrow G$	$F \Leftrightarrow G$
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

Una volta introdotte le formule ben formate, è possibile definire le sottoformule, ovvero una parte di una fbf che è a sua volta una fbf. L'insieme $Stfm(F)$ delle sottoformule di F è definito come il più piccolo insieme di formule tale che:

- $F \in Stfm(F)$;
- Se $\neg G \in Stfm(F)$, allora $G \in Stfm(F)$;
- se $G \wedge H, G \vee H, G \Rightarrow H, G \Leftrightarrow H \in Stfm(F)$, allora $H, G \in Stfm(F)$.

Semantica La semantica, invece, ha lo scopo di assegnare un significato alle formule appena definite, tramite una funzione I, detta interpretazione, che mappa ogni proposizione ad un valore di verità (vero o falso): formalmente, $I : \{fbf\} \to \{0, 1\}$. Tale funzione, quindi, non fa altro che assegnare il valore di vero (1) o falso (0) alle lettere proposizionali costanti [1] e valuta il valore di verità di $\neg F, F \wedge G, F \vee G, F \Rightarrow G, F \Leftrightarrow G$ sulla base dei valori di verità delle proposizioni F e G.

Da qui, si introduce il simbolo $\vDash$ (doppio tornello), che si utilizza per associare formule ed interpretazioni. Dunque, la scrittura $I \vDash F$, che si legge I rende vera F, vale nei seguenti casi:

- $I \vDash A$ se e solo se $I(A) = T$, con A proposizione;
- $I \vDash \neg F$ se e solo se $I \nvDash F$;
- $I \vDash F \wedge G$ se e solo se $I \vDash F$ e $I \vDash G$;
- $I \vDash F \vee G$ se e solo se $I \vDash F$ o $I \vDash G$;
- $I \vDash F \Rightarrow G$ se e solo se $I \nvDash G$ o $I \vDash G$;
- $I \vDash F \Leftrightarrow G$ se e solo se $I \vDash F \Rightarrow G$ e $I \vDash G \to F$.

Dal concetto di interpretazione, si possono definire le seguenti proprietà della semantica delle formule proposizionali:

- Se $I \vDash F$, allora si dice che I è un modello di F;
- F si dice valida (o si dice essere una tautologia) se e solo se per ogni interpretazione I vale che $I \vDash F$;
- F è soddisfacibile se e solo se esiste un'interpretazione I tale che $I \vDash F$;
- F è falsificabile se e solo se esiste un'interpretazione I tale che $I \nvDash F$;
- F è insoddisfacibile se e solo se per ogni interpretazione I vale che $I \nvDash F$;
- F è contingente se e solo se è sia soddisfacibile che falsificabile;
- Ogni formula del tipo $F \wedge \neg F$ è una contraddizione, indicata con $\perp$;
- Ogni formula del tipo $F \vee \neg F$ è detta principio del terzo escluso, indicata con $\top$

[1] Le lettere proposizionali costanti sono T (o V) per rappresentare una proposizione vera, oppure $\perp$ (o F) per rappresentare una proposizione falsa

Un insieme di formule $\mathcal{F}$ comporta logicamente una formula G o, equivalentemente, una formula G è una conseguenza logica di un insieme di formule $\mathcal{F}$, se ogni modello di $\mathcal{F}$ è anche un modello di G e si scrive con $\mathcal{F} \vDash G$. Dopo aver stabilito una corrispondenza semantica fra due formule logiche, è possibile anche stabilire una relazione di equivalenza fra due formule logiche, relazione che vale solamente se la corrispondenza fra le due formule è biunivoca, ovvero se vale sia $F \vDash G$ che $G \vDash F$: una tale relazione si rappresenta con la scrittura $F \equiv G$.

Esistono innumerevoli equivalenze notevoli:

$$
\begin{aligned}
F \wedge F &\equiv F \\
F \vee F &\equiv F \\
F \wedge G &\equiv G \wedge F \\
F \vee G &\equiv G \vee F \\
F \wedge (G \wedge H) &\equiv (F \wedge G) \wedge H \\
F \vee (G \vee H) &\equiv (F \vee G) \vee H \\
(F \wedge G) \vee F &\equiv F \\
(F \vee G) \wedge F &\equiv F \\
F \wedge (G \vee H) &\equiv (F \wedge G) \vee (F \wedge H) \\
F \vee (G \wedge H) &\equiv (F \vee G) \wedge (F \vee H) \\
\neg\neg F &\equiv F \\
\neg(F \wedge G) &\equiv \neg F \wedge \neg G \\
\neg(F \vee G) &\equiv \neg F \wedge \neg G \\
F \Leftrightarrow G &\equiv (F \Rightarrow G) \wedge (G \Rightarrow F) \\
F \Rightarrow G &\equiv \neg F \vee G \\
F \Rightarrow G &\equiv \neg G \Rightarrow \neg F
\end{aligned}
$$

In logica prposizionale è anche possibile sostituire una sottoformula G, di una formula ben formata F, con una formula H: la formula risultate viene indicata con la scrittura $F[G \backslash H]$. Tale sostituzione, però, può avvenire solamente se $G \equiv H$.

In base a questi concetti si può notare che non tutti i connettivi logici sono strettamente necessari, in quanto possono essere sostituiti con altri. A questo proposito, un insieme di connettivi è detto funzionalmente completo se e solo se qualunque formula proposizionale può essere trasformata in una formula semanticamente equivalente che contiene solamente i connettivi dell'insieme dato. Sfruttando tali insiemi, detti minimali, e le equivalenze semantiche, è possibile definire delle forme normali, che introducono degli schemi sintattici di completa generalità semantica per scrivere formule, permettendo così di formalizzare il significato di qualsiasi formula che si possa scrivere con la completa generalità della logica proposizionale. In altre parole, per ogni formula ben formata esistono una o più formule logicamente equivalenti ad essa scritte in una forma normale. Esistono tre principali forme normali per la logica proposizionale, chiamate forma negativa, forma congiuntiva e forma disgiuntiva. Una formula è in forma normale negativa se e solo se è composta solamente da letterali, congiunzioni e disgiunzioni; una formula è in forma normale congiuntiva (detta anche CNF) se e solo se ha la forma $C_1 \wedge C_2 \wedge ... \wedge C_n$, dove C_i è una disgiunzione di letterali; una formula è in forma normale disgiuntiva (detta anche DFN) se e solo se ha la forma $D_1 \vee D_2 \vee ... \vee D_n$, dove D_i è una congiunzione di letterali.

A questo punto è possibile completare la definizione della logica proposizionale attraverso i concetti di assioma e regole di inferenza, che costituiscono un sistema formale assiomatico-deduttivo (in inglese calculus). Questi elementi definiscono una relazione di derivabilità (relazione già analizzata nel contesto delle grammatiche), detta anche dimostrabilità, tra un insieme di formule $\mathcal{F}$ e una formula G. Dunque, i sistemi formali della logica hanno un compito molto simile a quello assolto dalle grammatiche, ovvero producono meccanicamente una formula risultante a partire da un insieme iniziale di formule, applicando assiomi e regole di inferenza. Si scrive, quindi, $\mathcal{F} \vdash G$ se G può essere ottenuto da $\mathcal{F}$. Idealmente, la relazione di derivabilità dovrebbe essere corretta (cioè se $\mathcal{F} \vdash G$ allora $\mathcal{F} \vDash G$) e completa (cioè se $\mathcal{F} \vDash G$ allora $\mathcal{F} \vdash G$). Se una formula F può essere derivata in una teoria $\mathcal{F}$ usando solamente assiomi e regole di inferenza di un sistema, allora si dice che F è un teorema.

5.2 Logica del Primo Ordine

La logica proposizionale appena analizzata ha molte applicazioni, ma il suo potere espressivo è ristretto. Per questo motivo, nel 1979 è stata sviluppata la logica del primo ordine, che permette dal punto di vista ontologico di considerare non solo fatti (come avveniva nella logica proposizionale), ma anche proprietà, relazioni e funzioni.

Sintassi In modo formale:

Definizione 5.2.1. *La logica del primo ordine è composta da un linguaggio $\mathcal{L}$, il cui alfabeto è costituito dai seguenti elementi:*

1. *Un insieme numerabile infinito di variabili;*
2. *Un insieme di simboli di funzione;*
3. *Un insieme di simboli di predicati (o relazioni);*
4. *Un insieme di connettivi logici: $\neg$ (NOT), $\wedge$ (AND), $\vee$ (OR), $\Rightarrow$ (Implicazione) e $\Leftrightarrow$(Coimplicazione);*
5. *Un insieme di quantificatori: $\exists$ (Esiste) e $\forall$ (Per ogni);*
6. *Un insieme di simboli di punteggiatura: (,) e le virgole.*

Ogni simbolo di funzione e relazione ha una arietà fissata, che indica il numero di argomenti associati a quella determinata funzione. Le funzioni nullarie sono dette costanti, mentre i predicati costanti sono detti proposizioni. I simboli dell'alfabeto sono privi di significato: assegnarne uno è compito della semantica.

Per poter scrivere formule nella logica del primo ordine c'è la necessità di denotare tutti gli oggetti di cui il linguaggio $\mathcal{L}$ può parlare, detti termini. Tale denotazione avviene induttivamente come segue:

- ogni variabile è un termine della formula;
- se f è un simbolo di funzione n-aria e $t_1, t_2, ..., t_n$ sono termini, allora $f(t_1, t_2, ..., t_n)$ è un termine.

Gli oggetti appena denotati attraverso i termini, si possono utilizzare all'interno delle formule della logica del primo ordine, definite anch'esse in maniera induttiva. L'insieme delle formule della logica del primo ordine è definito come il più piccolo insieme tale che:

- Se p è un simbolo di relazione n-aria e $t_1, t_2, ..., t_n$ sono termini, allora $p(t_1, t_2, ..., t_n)$ è una formula detta atomica o semplicemente atomo;
- Se F e G sono formule e X è una variabile, allora $\neg F, F \vee G, F \wedge G, F \Rightarrow G, F \Leftrightarrow G, \exists X F, \forall X F$ sono formule.

Nella scrittura di formule appartenenti alla logica del primo ordine, c'è un'osservazione da fare: quando si utilizza il quantificatore $\forall$ il connettivo principale utilizzato è $\Rightarrow$, mentre nel caso si utilizzi il quantificatore $\exists$ allora il connettivo principale è $\wedge$. Inoltre, se $QX(F)$ rappresenta una formula in cui Q è un quantificatre, allora F si dice ambito di Q e che Q è applicato ad F. Un'occorrenza di una variabile in una formula è legata se e solo se la sua occorrenza è entro l'ambito di un quantificatore che impiega quella variabile, altrimenti è libera. Una formula è chiusa se e solo se non contiene occorrenze libere di variabili. Le formule chiuse sono quelle per le quali, data un'interpretazione I, si può calcolare la veridicità.

Semantica Come per il caso della logica proposizionale, anche la logica del primo ordine ha una semantica basata sul concetto di interpretazione: un'interpretazione I di un alfabeto A è un dominio non vuoto D (indicato anche con $|I|$) e una funzione che che associa ogni costante $c \in A$ a un oggetto $c_I \in D$, ogni simbolo n-ario di funzione $f \in A$ a una funzione $f_I : D^n \to D$ e ogni simbolo n-ario di predicato $p \in A$ a una relazione $p_I \subseteq D \times D \times ... \times D$, n volte. Prima di poter assegnare un significato alle fromule, va definito il significato di ogni termine, indicato con $\phi_I(t)$ con t il termine a cui dare singificato nell'interpretazione I.

ϕ è induttivamente definito nel seguente modo:

1. c_I se t è una costante c;

2. $\phi(X)$ se t è una variabile X;

3. $f_I(\phi_I(t_1), ..., \phi_I(t_n))$ se t è nella forma $f(t_1, ..., t_n)$.

Ora, sia ϕ una valutazione, X una variabile, I un'interpretazione e $c_I \in |I|$, allora $\phi[X \to c_I]$ è una valutazione identica a ϕ, eccetto per il fatto che mappa X in c_I. Il significato di una formula, quindi, è un valore di verità che è definito induttivamente. Dunque la scrittura $I \vDash_\phi F$, che si legge F è vero rispetto all'interpretazione I e al significato ϕ, vale nei seguenti casi:

- $I \vDash_\phi p(t_1, ..., t_n)$ se e solo se $< \phi_I(t_1), ..., \phi_I(t_n) > \in p_I$;

- $I \vDash_\phi \neg F$ se e solo se $I \nvDash_\phi F$;

- $I \vDash_\phi (F \wedge G)$ se e solo se $I \vDash_\phi F$ e $I \vDash_\phi G$;

- $I \vDash_\phi (F \vee G)$ se e solo se $I \vDash_\phi F$ o $I \vDash_\phi G$;

- $I \vDash_\phi (F \Rightarrow G)$ se e solo se $I \nvDash_\phi F$ o $I \vDash_\phi G$

- $I \vDash_\phi (F \Leftrightarrow G)$ se e solo se $I \vDash_\phi (F \Rightarrow G)$ e $I \vDash_\phi (G \Rightarrow F)$;

- $I \vDash_\phi \forall X(F)$ se e solo se $I \vDash_{\phi[X \to c_I]} F$ per ogni $c_I \in |I|$;

- $I \vDash_\phi \exists X(F)$ se e solo se $I \vDash_{\phi[X \to c_I]} F$ per qualche $c_I \in |I|$

Se F è una formula chiusa, il suo significato dipende solamente dall'interpretazione I, che viene detta modello per F ($I \vDash F$) se e solo se per ogni valutazione ϕ si ha che $I \vDash_\phi F$. Inoltre, se $\mathcal{F}$ è un insieme di formule, un'interpretazione è modello di $\mathcal{F}$ se e solo se tale interpretazione è modello per ogni $F \in \mathcal{F}$. La relazione di conseguenza logica $\vDash$ tra insiemi di formule e formule può essere estesa anche per la logica del primo ordine, così come anche i concetti di validità, soddisfacibilità, falsificabilità, contingenza e insoddisfacibilità, analizzati nello studio della logica proposizionale.

Come nel caso della logica proposizionale, esistono innumerevoli equivalenze notevoli:

$$\begin{aligned}
\forall X(F) &\equiv \neg(\exists X(\neg F)) \\
\exists X(F) &\equiv \neg(\forall X(\neg F)) \\
\forall X(F) \wedge (\forall X)G &\equiv \forall X(F \wedge G) \\
\exists X(F) \vee (\exists X)G &\equiv \exists X(F \vee G) \\
(\forall X)(\forall Y)F &\equiv (\forall Y)(\forall X)F \\
(\exists X)(\exists Y)F &\equiv (\exists Y)(\exists X)F \\
(\forall X(F)) \wedge G &\equiv \forall X(F \wedge G)^* \\
(\forall X(F)) \vee G &\equiv \forall X(F \vee G)^* \\
(\exists X(F)) \wedge G &\equiv \exists X(F \wedge G)^* \\
(\exists X(F)) \vee G &\equiv \exists X(F \vee G)^*
\end{aligned}$$

Le equivalenze segnate con * sono valide solo se X non è libera in G.

5.3 Logica Monadica del Primo Ordine

In questa sezione si analizza un frammento della logica monadica del primo ordine, che permette di descrivere certe stringhe su un determinato alfabeto I. La logica monadica, come suggerisce il nome, si occupa solamente dei predicati monadici, ovvero di tutti quei predicati che hanno arietà pari ad uno.

Sintassi Una generica formula F appartenente alla logica monadica del primo ordine ha una sintassi molto semplice, articolata nei seguenti casi:

- $a(x)$ con $a \in I$, ovvero un predicato unario per ogni simbolo dell'alfabeto I;
- $x < y$, che costituisce l'unica eccezione della logica monadica in quanto non è un operatore unario, ma binario;
- $\neg F$, operatore NOT;
- $F \wedge F$, operatore AND;
- $\forall x(F)$, quantifcatore di una formula rispetto ad una variabile;

Il dominio delle variabili è $\mathbb{N}$.

Si può osservare come, rispetto alla logica proposizionale e del primo ordine, la logica monadica non fa uso della maggior parte degli operatori logici, del quantificatore esistenziale, della maggior parte degli operatori matematici, di oggetti o funzioni. Tutti i costrutti non utilizzati da tale logica possono essere drivati dagli operatori che sono stati presentati.

Le formule scritte in questa logica possono confrontare dei numeri rappresentati dalle variabili, che a loro volta rappresentano posizioni all'interno delle stringhe scritte sull'alfabeto di riferimento. Quindi con l'alfabeto I di riferimento posso scrivere delle stringhe mediante i predicati unari e, con la logica monadica è possibile calcolare la posizione, di determinati caratteri all'interno di tali stringhe per poterli confrontare con altre posizioni.

Con gli operatori logici, matematici e i quantificatori utilizzati nella logica monadica del primo ordine, come già detto in precedenza, è possibile derivare tutti gli altri operatori. Nello specifico:

$$
\begin{aligned}
F_1 \vee F_2 &\equiv \neg(\neg F_1 \neg F_2) \\
F_1 \Rightarrow F_2 &\equiv \neg F_1 \vee F_2 \\
\exists x(F) &\equiv \neg \forall x(\neg F) \\
x = y &\equiv \neg(x < y) \wedge \neg(y < x) \\
x \leq y &\equiv \neg(y < x)
\end{aligned}
$$

In realtà si possono anche introdurre alcune abbraviazioni di comodo. Le più utilizzate sono:

$$
\begin{aligned}
x = 0 &\equiv \forall y(\neg(y < x)) \\
successor(x, y) &\equiv x < y \wedge \exists z(x < z \wedge z < y) \\
y = x + 1 &\equiv successor(x, y) \\
y = x + k\,(k > 1) &\equiv \exists z_1, ..., z_{k-1}(y = z_{k-1} + 1 \wedge ... \wedge z_1 = x + 1) \\
y = x - 1 &\equiv successor(y, x) \\
last(x) &\equiv \neg \exists y(y > x)
\end{aligned}
$$

Le altre costanti (1, 2, 3, ...) possono essere ricavate tramite l'utilizzo del predicato successor (di arietà due) applicata alla costante 0 tante volte quanto il numero che si vuole ottenere.

Una volta introdotti tutti questi concetti della logica monadica del primo ordine, si può ora analizzare l'interpretazione di queste formule logiche rispetto a stringhe di un determinato alfabeto di riferimento.

Dato un alfabeto I, una stringa $w \in I^+$ ed un simbolo $a \in I$, $a(x)$ è vero se e solo se l'x-esimo simbolo di w è a, considerando un'indicizzazione che parte da 0.

Semantica La semantica, nella logica monadica del primo ordine, assegna un valore di verità o falsità alle formule, tramite una funzione di assegnamento $v_1 : V_1 \to [0, ..., |w| - 1]$, in cui $w \in I^+$ è una stringa composta a partire dall'alfabeto I, mentre V_1 è l'insieme delle variabili. Tale funzione si comporta nel seguente modo:

- $w, v_1 \vDash a(x)$ se e solo se $w = uav$ e $|u| = v_1(x)$;
- $w, v_1 \vDash x < y$ se e solo se $v_1(x) < v_1(y)$;
- $w, v_1 \vDash \neg F$ se e solo se $w, v_1 \nvDash F$;
- $w, v_1 \vDash F_1 \wedge F_2$ se e solo se $w, v_1 \vDash F_1$ e $w, v_1 \vDash F_2$;
- $w, v_1 \vDash \forall x(F)$ se e solo se $w, v_1' \vDash F$ $\forall v_1'$ con $v_1'(y) = v_1(y)$, $y \neq x$.

Proprietà La logica monadica del primo ordine ha delle importanti proprietà che consentono di identificare quali linguaggi è possibile rappresentare con tale logica. Nello specifico, tutti i linguaggi esprimibili mediante questa logica sono chiusti rispetto all'unione, all'intersezione e al complemento, Inoltre, si può dimostrare che non è possibile esprimere il linguaggio L_p composto da tutte e sole le stringhe di lunghezza pari con $I = \{a\}$: quindi, si può osservare come la logica monadica del primo ordine è strettamente meno espressiva rispetto agli automi a stati finiti, il che significa che data una formula di tale logica, si può sempre costruire un FSA equivalente, ma non il contrario. Infine, i linguaggi definiti da questa logica non sono chiusi rispetto alla stella di Kleene, ed è per questo motivo che tali linguaggi sono anche detti star-free, definibili tramite l'unione, l'intersezione, il complemento e la concatenazione di linguaggi finiti.

5.4 Logica Monadica del Secondo Ordine

Per riuscire ad ottenere lo stesso potere espressivo degli FSA è necessario permettere alla logica del primo ordine di quantificare sui predicati monadici, ovvero di poter quantificare anche su insiemi di posizioni. Si ammettono quindi formule del tipo $\exists X(F)$ oppure $\exists X(x)$, in cui X è detta essere una variabile del secondo ordine, il cui dominio non è più l'insieme dei numeri naturali $\mathbb{N}$, ma l'insieme dei predicati monadici. Per convenzione si utilizzano le lettere maiuscole per indicare le variabili del secondo ordine e lettere minuscole per le variabili del primo ordine.

Semantica L'assegnamento delle variabili del secondo ordine, che fanno parte dell'insiemeV_2, avviene attraverso la funzione $v_2 : V_2 \to \wp([0, ..., |w| - 1])$ tale che:

- $w, v_1, v_2 \vDash X(x)$ se e solo se $v_1(x) \in v_2(X)$;
- $w, v_1, v_2 \vDash \exists X(F)$ se e solo se $w, v_1, v_2' \vDash F$ per qualche $v_2'(Y) = v_2(Y)$, $Y \neq X$

Tramite questa logica, si possono scrivere formule che risolvono il problema della rappresentazione del linguaggio L_p, composto da tutte e sole le stringhe di lunghezza pari con $I = \{a\}$, che si può indicare nel seguente modo: $\exists D(\forall x(\neg D(0) \wedge (\neg D(x) \Leftrightarrow D(x+1)) \wedge a(x) \wedge (last(x) \Rightarrow D(x))))$, in cui D è una variabile del secondo ordine che indica le posizioni dispari.

Si può dimostrare che, data una qualsiasi formula F appartenente alla logica monadica del secondo ordine, è possibile costruire un automa a stati finiti che accetta lo stesso linguaggio L definito da F e, viceversa, dato un qualsiasi automa a stati finiti, è possibile enunciare una formula che riconosce lo stesso linguaggio. Di conseguenza, si può affermare che la classe dei linguaggi definibili dalle formule della logica monadica del secondo ordine coincide con i linguaggi regolari.

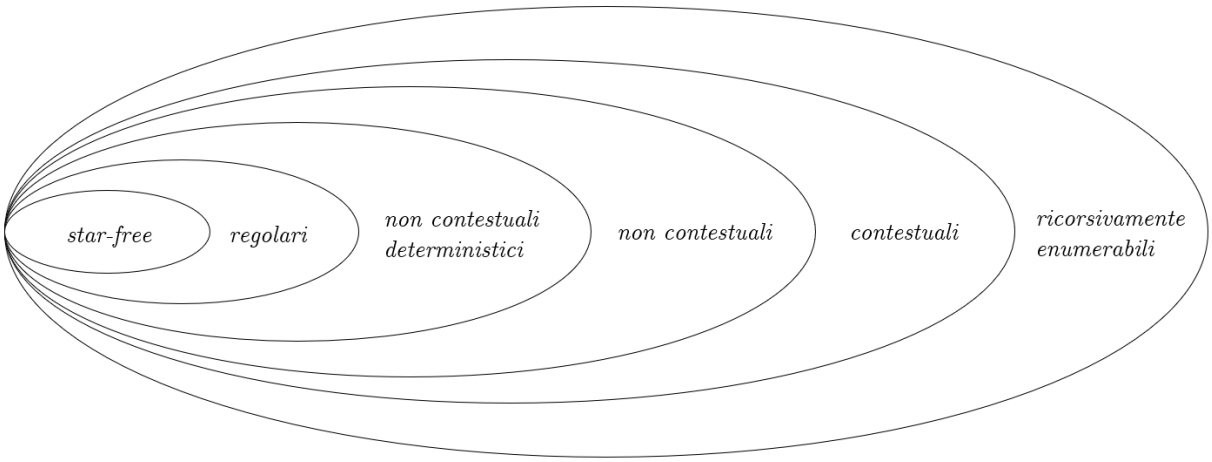


Figura 5.1: Gerarchia dei linguaggi

5.5 Logica per la descrizione di Proprietà

Quando si programma una funzione è importante definire con precisione quale sia il suo funzionamento, senza necessariamente descrivere come funzioni. A tal proposito si è soliti scrivere le cosiddette prcondizioni e postcondizioni:

- Precondizione: indica le condizioni che devono valere prima che la funzione venga invocata;
- Postcondizione: indica le condizioni che devono valere al termine dell'esecuzione del programma.

Dunque, il programma P deve essere tale per cui se la precondizione Pre vale prima dell'esecuzione di tale programma, allora la post condizione $Post$ deve valere dopo l'esecuzione. Tali condizioni possono essere espresse in linguaggi diversi, ma il più comune è la logica del primo ordine.

Capitolo 6

Computabilità

In questa sezione si cercherà di rispondere alla domanda: quali problemi possono essere affrontati e risolti mediante gli automi analizzati? Si ricordi che automi e grammatiche, pur essendo modelli matematici, si possono considerare dispositivi meccanici per la risoluzione di problemi e che esistono formalismi che hanno un potere espressivo maggiore di altri, ossia sono in grado di riconoscere una classe di linguaggi che altri formalismi non riescono a riconoscere. Inoltre, si ricordi che, nessun formalismo è più potente delle macchine di Turing, sia dal punto di vista del riconoscimento che della traduzione di linguaggi e, per tanto, sono detti formalismi massimi.

6.1 Formalizzazione dei Problemi

Molti problemi possono essere opportunamente descritti come il riconoscimento di un determinato linguaggio o come la sua traduzione in un altro linguaggio. Ogni problema matematico è descrivibile mediante una di queste forme, alla sola condizione che il dominio di tale problema sia un insieme numerabile, in maniera tale che i suoi elementi si possono porre in corrispondenza biunivoca con gli elementi di $\mathbb{N}$ o, se si preferisce, di V^*, in cui V rappresenta un alfabeto. Dunque, il problema di origine si può riformulare come il problema di calcolo di una funzione $f : \mathbb{N} \to \mathbb{N}$. Quanto detto è in perfetto accordo con tutti i formalismi matematici esaminati fino ad ora: questi, infatti, sono discreti e hanno un dominio matematico numerabile.

Il riconoscimento di linguaggi e la loro traduzione sono due formulazioni differenti di un problema, che sono facilmente riducibili l'uno all'altro. Infatti, il problema di stabilire se una determinata stringa x appartenga o meno al linguaggio L può anche essere impostato come la traduzione $\tau_L(x)$, per cui $\tau_L(x) = 1$ se $x \in L$, $\tau_L(x) = 0$ altrimenti. Viceversa, data la traduzione $\tau : V_1^* \to V_2^*$, si può definire il linguaggio seguente:

$L_\tau = \{z \mid z = x\$y,\ x \in V_1^*,\ y = \tau(x) \in V_2^*,\ \$ \notin (V_1 \cup V_2)\}$

ovvero il linguaggio formato da una stringa e la sua traduzione, separati dal simbolo \$. Un dispositivo che riconosce il linguaggio L_τ può essere utilizzato come trasduttore che calcola τ: per ogni x, infatti, è possibile enumerare tutte le $y \in V_2^*$ e verificare se $x\$y \in L_\tau$ oppure no. Prima o poi, se la funzione $\tau(x) \neq \perp$, verrà trovata una stringa per cui la macchina risponderà positivamente.

6.2 Tesi di Church

Le macchine di Turing, come visto in precedenza, sono il formalismo più potente che si ha a disposizione per il calcolo computazionale: ogni programma eseguibile da un calcolatore moderno può essere eseguito anche da una macchina di Turing. Dunque, le macchine di Turing hanno la stessa espressività dei linguaggi di programmazione ad alto livello, detti anche Turing completi.

Più formalmente, data una TM M, è possibile costruire un programma, scritto in un determinato linguaggio di programmazione (come C, Java ecc...), che simuli il ocmportamento di M, purchè il calcolatore disponga di una quantità di memoria sufficiente durante l'esecuzione. Inoltre, dato un programma scritto in un determinato linguaggio di programmazione, è possibile coistruire una TM M che calcoli la stessa funzione calcolata dal programma.

Tesi 6.2.1 (Tesi di Church - Prima Parte)**.** *Non esiste alcun formalismo, per modellare una determinata computazione meccanica, che sia più potente delle TM e dei formalismi ad essi equivalenti.*

La tesi di Church non è un teorema perchè per sua natura non è dimostrabile, in quanto andrebbe verificato ogni qual volta si introduce un nuovo formalismo computazionale.

In base a questo risultato si può affermare che, se si riesce a dimostrare che un determinato problema è risolvibile da una TM, allora è sicuramente possibile risolverlo mediante un modello matematico di calcolo, che abbia la stessa potenza delle macchine di Turing. Viceversa, se si dimostra che un problema non può essere risolto da una TM, allora è verificato che tale problema è irrisolvibile da qualunque modello matematico.

Alogoritmi Si introduce ora il concetto di algoritmo, centrale nell'informatica. Per algoritmo si intende la procedura di risoluzione di un problema mediante un dispositivo automatico di calcolo. Gli algoritmi si possono anche intendere come un metodo astratto di descrizione dei programmi eseguibili, ovvero una sequenza di comandi che, una volta eseguiti, portano alla risoluzione del problema.

Ogni algoritmo ha le seguneti proprietà:

1. Un algoritmo deve contenere una sequenza finita di istruzioni;
2. Ogni istruzione deve essere immediatamente eseguibile da qualche procedimento meccanico di calcolo, ossia deve esistere un processore che sia in grado di comprendere univocamente le istruzioni e di eseguirle producendo risultati precisi ed inequivocabili;
3. Il processore è dotato di celle di memoria in cui possono essere immagazzinati i riultati intermedi;
4. La computazione è discreta, ossia l'informazione è codificata in forma digitale e la computazione procede attraverso passi discreti;
5. Gli algorimi vengono eseguiti deterministicamente;
6. Non esiste un limite finito sui dati di ingresso e di uscita: ogni calcolatore può ricevere in ingresso o emettere in uscita stringhe di lunghezza arbitraria;
7. Non esiste un limite alla quantità di memoria richiesta per effettuare i calcoli;
8. Non esiste un limite al numero di passi discreti richiesti per effettuare un calcolo ed è dunque possibile avere computazioni infinite.

La tesi di Church non si ferma solo nell'affermazione che nessun formalismo sia più espresivo delle TM, ma afferma anche che nessun algoritmo è in grado di risolvere un problema che non è risolvibile da una TM. Formalmente:

Tesi 6.2.2 (Tesi di Church - Seconda Parte)**.** *Ogni algoritmo per la soluzione automatica di un problema può essere codificato in termini di una TM (o di un formalismo a potenza equivalente).*

Teorema 6.2.1. *Ogni funzione (o problema), per cui esiste una TM che la calcoli (o risolva), si dice computabile o calcolabile (o risolvibile). Un problema risolvibile la cui risposta sia booleana ed esistente per ogni valore del dominio di definizione (ossia è formalizzato da una funzione calcolabile e totale) si dice decidibile.*

Grazie alla seconda parte della tesi di Church si può affermare che è possibile studiare i limiti del calcolo automatico indipendentemente dalla formalizzazione del problema e del particolare modello computazionale.

6.3 Enumerazione delle TM

Le macchine di Turing possono essere viste come dei calcolatori astratti, specializzati nella risoluzione di un solo problema e non programmabili. Ci si pone quindi la domanda: 'le TM sono in grado di simulare i calcolatori programmabili e di risolvere i problemi da $\mathbb{N}$ a $\mathbb{N}$?'

Per poter risondere a tale domanda, si noti innanzitutto che dato un qualsiasi insieme S, questo può essere enumerato algoritmicamente se è possibile stabilire una biiezione fra l'insieme S e l'insieme dei numeri naturali $\mathbb{N}$, calcolabile attraverso un algoritmo o da una TM. Alla stessa maniera è possibile enumerare algoritmicamente l'insieme delle TM tramite una biiezione $E : \{TM\} \leftrightarrow \mathbb{N}$. Tale biiezione è implementabile da un algoritmo che riceve in ingresso un numero n e ritorna la n-esima macchina

di Turing. Un'enumerazione calcolabile da una TM viene chiamata Gödelizzazione, mentre il numero naturale biiettivamente associato da tale enumerazione ad una TM è detto numero di Gödel della TM.

Inoltre, è noto che una TM M può risolvere una funzione $f_M : D \to R$, con D ed R opportunamente codificati nell'alfabeto di M, dunque si indicherà con f_y la funzione calcolata dalla y-esima macchina di Turing, indicata con $M_y = E(y)$.

6.4 Macchine di Turing Universali

Le UTM (Universal Turing Machines) sono TM in grado di modellare dispositivi generali di risoluzione dei problemi, in cui il problema da risolvere non viene codificato nella struttura del dispositivo (come avviene per le TM), ma gli viene fornito come input, assieme ai dati con cui operare (esattamente come gli oderni calcolatori). Le UTM sono quindi MT che calcolano la funzione $g(y, x) = f_y(x)$, in cui y rappresenta la funzione f_y, calcolata dalla TM M_y, ed x rappresenta l'ingresso su cui M_y opera; calcolano, dunque, il valore della funzione f_y applicata ad x.

Come si può osservare, la UTM così definita non sembra appartenere all'insieme delle macchine di Turing, in quanto la funzione $g(y, x)$ è opera da $\mathbb{N} \times \mathbb{N}$ ad $\mathbb{N}$, anzichè da $\mathbb{N}$ ad $\mathbb{N}$ come tutte le altre TM. È però possibile, come già dimostrato in precedenza, definire una biiezione calcolabile algoritmicamente, tramite la funzione:

$$d(x, y) = x + \frac{(x + y)(x + y - 1)}{2}$$

che mette in corrispondenza l'insieme $\mathbb{N} \times \mathbb{N}$, composto dalle coppie di numeri naturali, all'insieme $\mathbb{N}$, composto da numeri naturali.

Graficamente, è come visitare le coppie di punti nel piano in un ordine prefissato, dove la posizione di un punto nella visita rappresenta il numero naturale associato alla coppia che identifica le coordinate del punto.

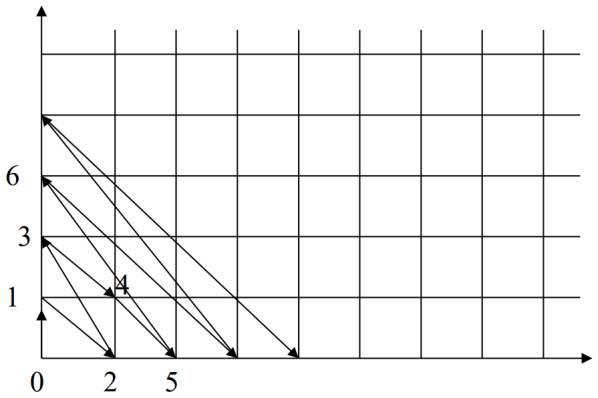


Figura 6.1: Grafico di biiezione

Si osservi che la funzione $g(y, x)$ è computabile da una macchina di Turing, ossia $\exists i \in \mathbb{N} : f_i = g$, ed è possibile calcolarla tramite i seguenti passaggi:

1. Si sceglie un alfabeto finito A per codificare i numeri naturali ed ogni altra informazione richiesta per la computazione;
2. Si traduce la rappresentazione di n in un'opportuna rappresentazione della coppia $< x, y >$ corrispondente ad n. La rappresentazione decimale di n può essere tradotta nelle due rappresentazioni decimali di x ed y, separate dal simbolo \$;
3. Si traduce y in un'opportuna codifica della TM y-esima M_y nella enumerazione di Gödel;
4. Si simula la computazione di M_y su x.

Teorema 6.4.1. *Per ogni x ed ogni y, esiste e si può costruire una macchina di Turing universale in grado di calcolare* $g(y, x) = f_y(x)$

Tramite questo teorema si può affermare che è possibile creare una macchina di Turing che simuli il comportamento degli odierni calcolatori "general purpose".

6.5 Problemi Algoritmicamente Irrisolvibili

Come si è visto in precedenza, tutte le funzioni computabili $f_y : \mathbb{N} \to \mathbb{N}$ si possono enumerare: questo significa che la cardinalità dell'insieme delle funzioni computabili è pari ad $\aleph_0$, ovvero alla cardinalità dei numeri naturali $\mathbb{N}$. L'insieme delle funzioni $\{f : \mathbb{N} \to \mathbb{N}\}$ contiene la classe delle funzioni $\{f : \mathbb{N} \to \{0,1\}$, in quanto $\{0,1\} \subseteq \mathbb{N}$. Quindi, poichè $\mid \{f : \mathbb{N} \to \mathbb{N}\} \mid \geq \mid \{f : \mathbb{N} \to \{0,1\}\} \mid = \wp(\mathbb{N}) = 2^{\aleph_0}$, si può dedurre che la cardinalità della classe delle funzioni da $\mathbb{N}$ ad $\mathbb{N}$ è strettamente maggiore della cardinalità della classe delle funzioni computabili: dunque, gran parte delle funzioni di $\mathbb{N}$ non può essere calcolata.

Ora, quando si vuole definire una funzione si usa un linguaggio che la esprima, ovvero un sottoinsieme del monoide libero su di un determinato alfabeto finito: dunque, il linguaggio è un insieme numerabile. Si ricava quindi che la classe delle funzioni denotabili è a sua volta numerabile.

Quando si scrive un programma, ci sono diverse proprietà che si vorebbero garantire. Una di queste è la terminazione del programma, ovvero la garanzia che, dato un qualsiasi ingresso conforme al programma stesso, esso termini la propria computazione e non vada, dunque, in un ciclo infinito. Nella realtà, però, non è possibile garantire a priori la terminazione del programma per un generico valore in ingresso, nè decidere attraverso un algoritmo se ciò possa avvenire in corrispondenza di uno specifico valore in ingresso. Più in generale, il problema della terminazione del calcolo automatico è in generale non decidibile, nonostante tale problema sia definibile. Si è quindi constatato che esistono problemi definibili, ma che non possono essere risolti algoritmicamente: dunque, l'insieme dei problemi definibili contiene strettamente l'insieme dei problemi risolvibili, nonostante entrambi siano numerabili e con la stessa cardinalità.

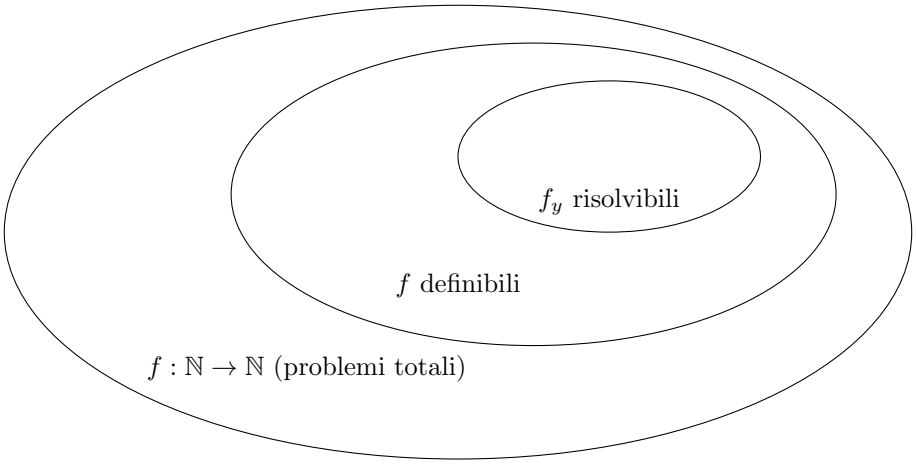


Figura 6.2: Gerarchia dei problemi

Teorema 6.5.1 (Halting Problem)**.** *Nessuna TM può calcolare la funzione* $g : \mathbb{N} \times \mathbb{N} \to \{0,1\}$ *definita nel seguente modo:*
$g(x,y) = if\ f_y(x) = \perp\ then\ 1\ else\ 0$

La dimostrazione di tale teorema si ottiene tramite la tecnica della diagonale, detta anche metodo di Cantor: l'obiettivo è quello di mostrare che un'enumerazione di oggetti di cardinalità almeno 2, non è completa, ossia che un oggetto che si vorrebbe trovare all'interno di tale enumerazione in realtà non è presente. L'enumerazione di una successione può essere rappresentata come una tabella con un numero infinito di righe. L'elemento che non compare in tale tabella viene individuato per assurdo considerando inizialmente la diagonale d (dunque d_i è l'elemento che si trova all'i-esima riga e all'i-esima colonna) e poi componendo una diagonale d' tale che, per ogni i, d'_i sia diverso da d_i.

Teorema 6.5.2. *Nessuna TM è in grado di calcolare la funzione totale* k *definita nel seguente modo:*
$k(x) = if\ f_x(x) \neq \perp\ then\ 1\ else\ 0$

Questo problema rappresenta un caso speciale della funzione $g(y,x)$ in quanto $k(x) = g(x,x)$, dunque la calcolabilità della funzione k è direttamente correlata alla calcolabilità della funzione g. Si noti che, in generale, se un problema è irrisolvibile, può accadere che un suo caso particolare sia risolvibile, mentre una sua generalizzazione è necessariamente irrisolvibile. Al contrario, se un problema è risolvibile,

può accadere che una sua generalizzazione diventi irrisolvibile, mentre un suo caso particolare rimane sicuramente risolvibile.

Un altro teorema certamente importante è il seguente:

Teorema 6.5.3. *Nessuna TM è in grado di calcolare la funzione k definita nel seguente modo:*
$k(y) = if\ f_y(x) \neq \perp\ then\ 1\ else\ 0$

Da un punto di vista pratico questo problema è interessante perchè qualifica tutti i possibili dati in ingresso. Afferma, infatti, l'irrisolvibilità del problema di decidere se un certo programma termini la propria esecuzione per qualsiasi dato in ingresso o se, al contrario, per qualche dato il programma andrebbe in loop. Nel caso precedente, invece, si era interessati al problema di sapere se un certo programma con certi dati avrebbe terminato o meno la propria esecuzione.

In definitiva, si è constatato che esistono problemi non risolvibili algoritmicamente. Ciò non esclude comunque la possibilità di trovare una soluzione per tali problemi, in quanto non tutti i problemi sono risolvibili tramite un procedimento algoritmico.

6.6 Problemi di Decisione

Un problema di decisione è una domanda che ha come uniche risposte si o no (0, 1). Questo può anche essere espresso come un problema di appartenenza di un determinato elemento ad un certo insieme. Più in generale si noti che una qualsiasi proprietà di un determinato elemento di un insieme può essere formalizzata come un suo sottoinsieme (ad esempio, la proprietà di terminazione del calcolo per ogni valore dei dati in ingresso individua un sottoinsieme dell'insieme di tutti i programmi).

In questa sezione si prendono in particolare considerazione i sottoinsiemi di $\mathbb{N}$, indicati convenzionalmente con $S \subseteq \mathbb{N}$. Quindi, formalmente, dato un determinato elemento $x \in \mathbb{N}$ e un insieme S, si cerca di capire se x appartenga ad S.

Definizione 6.6.1. *La funzione caratteristica $c_S : \mathbb{N} \to \{0, 1\}$ di un insieme S è definita come segue:*
$c_S(x) = if\ x \in S\ then\ 1\ else\ 0$

La risolvibilità del problema di appartenenza ad un insieme (detta anche ricorsività dell'insieme) dipende dalla computabilità della funzione caratteristica c_S, come definito di seguito:

Definizione 6.6.2. *Un insieme S è ricorsivo (o decidibile) se e solo se la sua funzione caratteristica è computabile.*

Si noti, inoltre, che per ogni insieme S, la sua funzione caratteristica c_S è totale: infatti, dato un qualsiasi elemento $x \in \mathbb{N}$, questo necessariamente apparteniene o non appartiene all'insieme.

Definizione 6.6.3. *Un insieme S è ricorsivamente enumerabile (o semidecidibile) se e solo se è l'insieme vuoto oppure è l'immagine di una funzione totale e computabile g_s, ovvero:*
$S = I_{g_s} = \{x \mid \exists y, y \in \mathbb{N} \wedge x = g_s(y)\}$

Gli insiemi decidibili devono il loro nome al fatto che il problema di appartenenza può essere risolto tramite un algoritmo meccanico e che, quindi, una TM che implementi la loro funzione caratteristica fornisce necessariamente una risposta al quesito se $x \in S, \forall x \in \mathbb{N}$. Inoltre, per ogni insieme ricorsivamente numerabile S è possibile costruire una sequenza $x_0 = g_S(0), x_1 = g_S(1), x_2 = g_S(2), ...$ tale per cui, se $x \in S$, allora esiste i tale che $x = g_S(i)$. In questo caso, esaminando la sequenza di elementi $\{x_i\}$ si riuscirà a trovare l'elemento x, concludendo che questo appartiene all'insieme S. Perciò, se per un qualsiasi $\bar{i}$ risultasse che $x \notin \{g_S(i) \mid 0 \leq i \leq \bar{i}\}$ non si potrebbe concludere nè che $x \in S$, nè che $x \notin S$: per questo motivo, l'insieme S viene anche detto semidecidibile.

Teorema 6.6.1. *Se S è ricorsivo, è anche ricorsivamente enumerabile.*
S è ricorsivo se e solo se sia S che $\bar{S} = \mathbb{N} - S$ sono ricorsivamente enumerabili.

Quindi, riepilogando, si dice che un insieme S è:

- Ricorsivo (o Decidibile), se e solo se la sua funzione caratteristica c_S è computabile;
- Ricorsivamente enumerabile (o Semidecidibile), se e solo se:
  - S è l'insieme vuoto;
  - S è l'immagine di una funzione g_S totale e computabile (detta generatrice); quindi, $S = I_{g_S} = \{g_S(0), g_s(1), g_S(2), ...\}$

Si consideri ora il seguente teorema:

Teorema 6.6.2. *Per ogni insieme S, se $i \in S$ implica che f_i sia totale e se per ogni funzione f totale e computabile, esiste $i \in S \mid f = f_i$, allora S non è ricorsivamente enumerabile.*

Informalmente, questo teorema stabilisce che tutte le funzioni totali computabili non sono ricorsivamente enumerabili (mentre le funzioni parziali computabili lo sono). Dunque, tale teorema afferma implicitamente che non esiste nessun formalismo ricorsivamente enumerabile in grado di definire tutte e sole le funzioni totali e computabili: infatti, gli FSA sono in grado di definire le funzioni totali, ma non tutte, le TM definiscono tutte le funzioni computabili, ma anche quelle non totali, e un linguaggio di programmazione (come il C) è in grado di definire tutti gli algoritmi, ma anche anche quelli che non terminano mai.

Si cerca quindi di comprendere se sia possibile eliminare le funzioni non totali: per far ciò, si prenda in considerazione una generica funzione parziale, ad esempio, arricchendo $\mathbb{N}$ con il valore $\{\perp\}$ o con qualsiasi altro simbolo che indichi che la funzione non è definita per certi valori. Tale trasformazione da funzione parziale a totale, però, non può essere applicata perchè nel passaggio è possibile perdere la computabilità della funzione. Questo risultato è enunciato nel seguente teorema:

Teorema 6.6.3. *Non esiste una funzione totale e computabile h che sia un'estensione della seguente funzione: $g(x) = if\ f_x(x) \neq \perp\ then\ f_x(x) + 1\ else\ \perp$*

Tale teorema, afferma quindi che non è possibile estendere una funzione parziale ad una totale, in quanto si potrebbe perdere la sua computabilità.

Vale anche il seguente risultato:

Teorema 6.6.4. *Un insieme S è ricorsivamente enumerabile se e solo se $S = D_h$, in cui h è una funzione parziale e computabile ($S = \{x \mid h(x) \neq \perp\}$), oppure se e solo se $S = I_g$, in cui g è una funzione parziale e computabile ($S = \{x \mid x = g(y), y \in \mathbb{N}\}$).*

Quindi, dato l'insieme $K = \{x \mid f(x) \neq \perp\}$ questo è semidecidibile perchè $K = D_h$, con $h(x) = f_x(x)$, ma è anche indecidibile in quanto la funzione caratteristica dell'insieme K, definita come $c_K(x) = if\ f_x(x) \neq \perp\ then\ 1\ else\ 0$, non è computabile. Si è appena dimostrato che esistono insiemi che sono semidecidibili, ma allo stesso tempo indecidibili.

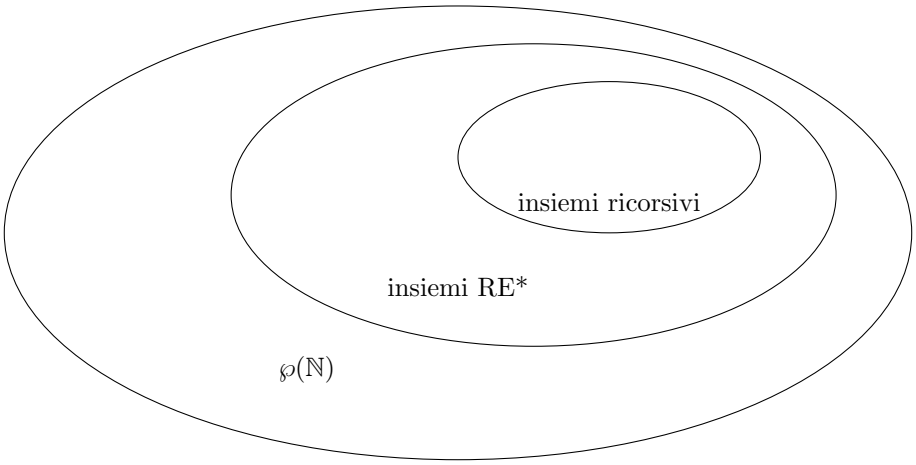


Figura 6.3: Gerarchia degli insiemi

*Con RE si indicano gli insiemi Ricorsivamente Enumerabili.
Si noti che tutte le inclusioni sono strette.

6.7 Teoremi di Kleene e Rice

Teorema 6.7.1 (Teorema di Kleene del punto fisso)**.** *Sia t una qualunque funzione totale e computabile. Allora è sempre possibile trovare un intero p, tale per cui:*

$f_p = f_{t(p)}$

La funzione f_p è detta punto fisso di t, perchè t trasforma f_p in f_p stessa.

Teorema 6.7.2 (Teorema di Rice)**.** *Sia F un insieme generico di funzioni computabili. L'insieme $S = \{x \mid f_x \in F\}$ degli indici delle TM che calcolano le funzioni di F, è ricorsivo se e solo se $F = \emptyset$ oppure F è l'insieme di tutte le funzioni computabili.*

Il teorema di Rice ha un forte impatto pratico negativo, in quanto afferma che, in tutti i casi non banali, S non è decidibile. Non è quindi possibile stabilire algoritmicamente se un dato algoritmo sia in grado di risolvere un determinato problema, nè se due programmi siano equivalenti (ossia se calcolino la stessa funzione). Il grande impatto pratico del teorema di Rice deriva dal fatto che il concetto di sottoinsieme F di funzioni computabili è un'espressione formale del concetto generale di proprietà di problemi risolvibili: una proprietà degli elementi di un insieme è un sottoinsieme dell'insieme dato e una funzione computabile è una formalizzazione del concetto di problema risolvibile; quindi, il teorema di Rice efferma che non è possibile stabilire se un determinato algoritmo risolve un problema pur risolvibile che godi di una qualsiasi proprietà non banale.

Parte II

Algoritmi e Strutture Dati

Capitolo 7

Complessità del Calcolo

Nei capitoli precedenti si è dimostrato come esistano alcuni problemi che, pur essendo ben definiti, non sono risolvibili algoritmicamente, per cui non esiste una TM in grado di risolvere quel determinato problema. Un'altra difficoltà da tenere in considerazione quando si manipolano i problemi, risiede nel tempo di esecuzione, ovvero il tempo impiegato dal programma per fornire una soluzione: se non è possibile ottenere una soluzione in un tempo ragionevole, ovviamente, il problema diventa intrattabile, nonostante sia teoricamente risolvibile.

Il concetto di intrattabilità è strettamente correlato al concetto di complessità del calcolo: più il problema è complesso e meno questo diventa trattabile. Informalmente, la complessità indica una misura del prezzo da pagare per risolvere il problema. Le risorse che si utilizzano per la risoluzione di un problema sono principalmente due: lo spazio, ovvero la memoria necessaria all'algoritmo, e il tempo richiesto per produrre la soluzione. Si parlerà quindi di complessità spaziale e di complessità temporale.

Si osservi inoltre che le due risorse, quella temporale e quella spaziale, seppur sembrino indipendenti l'una dall'altra, sono in realtà correlate: alla riduzione di utilizzo di una risorsa aumenta l'altra e viceversa.

7.1 Analisi di complessità

La complessità del calcolo dipende dalla dimensione e, spesso, anche dai particolari valori assunti dai dati in ingresso. Per questo motivo, si rende necessario effettuare un'analisi del caso pessimo, del caso medio e del caso ottimo, in funzione della dimensione dei dati in ingresso. Più formalmente, tali casi rappresentano rispettivamente la scelta di ingressi per cui viene eseguito il massimo numero di istruzioni nel programma, il comportamento del programma in relazione alla possibile distribuzione dei dati e la scelta di input per cui viene eseguito il minor numero di istruzioni.

In generale, il caso ottimo non è di particolare interesse e il caso medio richiede la conoscenza della distribuzione dei dati in ingresso (non sempre nota). Il caso pessimo, invece, è particolarmente interessante in quanto fornisce i peggiori risultati ottenibili dall'algoritmo in termini di consumo di spazio o tempo; si garantisce quindi che tutte le soluzioni proposte dall'algoritmo abbiano un dispendio di risorse minore rispetto al caso pessimo, indipendentemente dalla tipologia degli ingressi. Dunque, se il caso pessimo è ritenuto accettabile, allora anche tutti gli altri casi lo sono.

Inoltre, a differenza di quanto analizzato per la risolvibilità dei problemi, la complessità non è collegata solo al problema che si vuole affrontare, ma dipende anche dell'algoritmo che si utilizza per risolverlo.

Nel capitolo sugli automi, è stato più volte affermato che le Macchine di Turing sono il formalismo più potente che si ha a disposizione per la risoluzione di prblemi, dunque, risulta ragionevole definire la complessità temporale e spaziale impiegando un tale modello.

Definizione 7.1.1. *Sia M una TM deterministica a k nastri e sia $x \in I^*$. Sia $c_0 \vdash c_1 \vdash .... \vdash c_r$ una computazione, ovvero una sequenza di transizioni di M tale che $c_0 =< q_0, \uparrow x, \uparrow Z_0, ..., \uparrow Z_0 >$ e $c_i =< q_i, x_i \uparrow y_i, \alpha_{i1} \uparrow \beta_{i1}, ..., \alpha_{ik} \uparrow \beta_{ik} >$, in cui c_r è una configurazione di arresto, se esiste. Allora, la funzione che rappresenta la complessità temporale T_M di M è definita nel seguente modo:*

$$T_M = if \text{ la computazione termina then } r \text{ else } \infty.$$

Informalmente, quindi, la complessità temporale viene definita come una funzione che fornisce il numero esatto di passi richiesti da una TM per raggiungere la propria configurazione di arresto, se esiste, a partire dalla configurazione iniziale, per una qualsiasi stringa in ingresso. analogamente, si può definire la complessità spaziale come una funzione che fornisce il numero massimo di celle del nastro utilizzate.

Definizione 7.1.2. *Siano M, x, $c_0, ..., c_r$ definiti come nella definizione 7.1.1. La funzione che rappresenta la complessità spaziale S_M di M è definita nel seguente modo:*

$$S_M = \sum_{j=1}^{k} max_{i \in \{0,1,...,r\}}(|\alpha_{ij}|)$$

Inoltre, vale che:

$$\forall x, \frac{S_M}{k} \leq T_M(x)$$

Si noti, inoltre, che la definizione di $S_M(x)$ prende in considerazione solamente i nastri di memoria e ignora sia il nastro di ingresso che il nastro di uscita per il calcolo della complessità.

Secondo le definizioni 7.1.1 e 7.1.2, sia T_M che S_M sono funzioni definite su I^*, ma nella pratica la complessità di una soluzione dipende sia dal contenuto dell'ingresso che dalla sua dimensione: a partire da questa osservazione, si erano introdotti i concetti di complessità nel caso pessimo, medio e ottimo, che si possono ridefinire nel seguente modo, alla luce delle definizioni appena date:

Definizione 7.1.3. *Il caso pessimo, ottimo e medio sono così definiti:*

- *Caso pessimo:* $T_M(n) = max_{|x|=n} T_M(x)$;
- *Caso ottimo:* $T_M(n) = min_{|x|=n} T_M(x)$;
- *Caso medio:* $T_M(n) = \frac{\sum_{|x|=n} T_M(x)}{|I|^n}$

Una volta analizzata la complessità temporale tramite il formalismo delle macchine di Turing, si sposta l'attenzione sugli altri formalismi analizzati nel capitolo riguardante gli automi, in particolare sugli automi a stati finiti, sugli automi a pila e sulle macchine di turine a singolo nastro.

Dato un automa a stati finiti A, si definisce la complessità temporale T_A come l'intero i tale che $\delta^i(q_0, x) = q$ per qualche q, se esiste, ovvero il numero di transizioni effettuate per processare la stringa in ingresso x a partire dallo stato iniziale. Se $\delta^*(q_0, x)$ è indefinita, si pone $T_A = |x|$, ovvero pari alla lunghezza della stringa in ingresso. T_A, evidentemente, indica il numero di mosse compiute da A durante il riconoscimento della stringa x. Dunque, per ogni automa a stati finiti la complessità temporale T_A cresce in maniera lineare con il crescere della lunghezza della stringa. Al contrario, la sua complessità spaziale S_A non varia mai, in quanto gli FSA sono composti da un numero finito di stati definiti a priori, indipendentemente dalla lunghezza della stringa letta.

Dato un automa a pila A, si può analizzare sia la complessità temporale in funzione della stringa in ingresso ($T_A(x)$), sia in funzione della lunghezza della stringa ($T_A(n)$). Come per gli automi a stati finiti, quando si calcola la complessità temporale in funzione della stringa in ingresso, si contano il numero di passi che portano da una configurazione iniziale ad una configurazione finale. Quando invece si vuole utilizzare come parametro la lunghezza della stringa, la complessità temporale è calcolata come il massimo di tutte le complessità temporali in funzione delle stringhe in ingresso della lunghezza considerata. La complessità spaziale S_A, invece, viene associata al numero di celle di memoria della pila che vengono occupate dall'automa per portare a termine la computazione.

La complessità temporale e spaziale per le macchine di Turing a nastro singolo sono definite esattamente come per le TM a k-nastri. Data una macchina di Turing M a singolo nastro, la complessità spaziale $S_M(x)$, che corrisponde al massimo numero di celle del nastro di memoria occupate da M durante la computazione a fronte della stringa in ingresso x, non può mai essere minore di $|x|$, in quanto l'unico nastro presente in M è sia di ingresso, che di memoria, che di uscita. Ciò significa che l'unico nastro presente, di cui si deve analizzare l'occupazione, è precaricato con la stringa di ingresso x. Per quanto riguarda la complessità spaziale, le TM a singolo nastro sono meno efficienti delle TM multinastro e, in alcuni casi, di altri formalismi analizzati.

In generale, le macchine di Turing multinastro sono il formalismo più potente ed efficiente per la computazione di problemi.

7.2 Comportamento asintotico

Nella maggior parte dei casi si analizza la complessità spaziale o temporale di un determinato algoritmo per valori molto grandi dell'ingresso x, ovvero per x che tende ad infinito. In questi casi si analizza il comportamento asintotico dell'algoritmo, che fornisce un'approssimazione abbastanza precisa sul dispendio di risorse. La notazione dell'ordine di grandezza di una funzione, nota sotto il nome di notazione theta-grande (Θ), sottolinea i fattori dominanti che influeanzano la creascita della sua complessità in funzione della dimensione dell'ingresso. Oltre alla notazione theta-grande, esistono anche le notazioni o-grande (O) e omega-grande (Ω), le cui definizioni sono riportate di seguito:

Definizione 7.2.1 (Notazione O)**.** *Siano $g : \mathbb{N} \to \mathbb{R}^+$ ed $f : \mathbb{N} \to \mathbb{R}^+$ due funzioni. La funzione g è in $O(f)$ se e solo se esistono due numeri positivi c ed n_0 tali che per ogni $n \geq n_0$, $g(n) \leq cf(n)$. Ciò significa che $O(f) = \{g(n) : \mathbb{N} \to \mathbb{R}^+ \mid \exists c, n_0 > 0 \wedge \forall n \geq n_0, g(n) \leq cf(n)\}$*

Inoltre, vale che:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in O(g(n))$$

Definizione 7.2.2 (notazione Ω)**.** *Siano $g : \mathbb{N} \to \mathbb{R}^+$ ed $f : \mathbb{N} \to \mathbb{R}^+$ due funzioni. La funzione g è in $\Omega(f)$ se e solo se esistono due numeri positivi c ed n_0 tali che per ogni $n \geq n_0$, $cf(n) \leq g(n)$. Ciò significa che $\Omega(f) = \{g(n) : \mathbb{N} \to \mathbb{R}^+ \mid \exists c, n_0 > 0 \wedge \forall n \geq n_0, cf(n) \leq g(n)\}$.*

Inoltre, vale che:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) \in \Omega(g(n))$$

La notazione o-grande rappresenta quindi un limite superiore per la funzione data, mentre la notazione theta-grande rappresenta un limite inferiore. Nello specifico è di interesse pratico prendere in considerazione il minimo limite superiore e il massimo limite inferiore.

Definizione 7.2.3 (notazione Θ)**.** *Siano $g : \mathbb{N} \to \mathbb{R}^+$ ed $f : \mathbb{N} \to \mathbb{R}^+$ due funzioni. La funzione g è in $\Theta(f)$ se e solo se esistono tre numeri positivi c_1, c_2 ed n_0 tali che per ogni $n \geq n_0$, $c_1 f(n) \leq g(n) \leq c_2 f(n)$. Ciò significa che $\Theta(f) = \{g : \mathbb{N} \to \mathbb{R}^+ \mid \exists c_1, c_2, n_0 > 0 \wedge n \geq n_0, c_1 f(n) \leq g(n) \leq c_2 f(n)\}$*

Inoltre, vale che:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0 \Rightarrow f(n) \in \Theta(g(n))$$

Dalla definizione 7.2.3, si può dedurre che la funzione $f \in \Theta(g)$ se e solo se $f \in O(g)$ e $f \in \Omega(g)$.

Inoltre, Le notazioni indicate precedentemente godono delle seguenti proprietà:

- Transitività:
  - se $f(n) \in \Theta(g(n))$ e $g(n) \in \Theta(h(n))$, allora $f(n) \in \Theta(h(n))$;
  - se $f(n) \in O(g(n))$ e $g(n) \in O(h(n))$, allora $f(n) \in O(h(n))$;
  - se $f(n) \in \Omega(g(n))$ e $g(n) \in \Omega(h(n))$, allora $f(n) \in \Omega(h(n))$;
- Riflessività:
  - $f(n) \in \Theta(f(n))$;
  - $f(n) \in O(f(n))$;
  - $f(n) \in \Omega(f(n))$;
- Simmetria: $f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$;
- Simmetria trasposta: $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$.

Inoltre, la relazione Θ è una relazione di equivalenza.

7.3 Accelerazione Lineare

Si è precedentemente affermato che la complessità della soluzione di un determinato problema può essere migliorata mediante opportune modifiche all'algoritmo risolutivo. A tal proposito si enunciano i seguenti teoremi che pongono alcuni limiti al miglioramento degli algoritmi:

Teorema 7.3.1. *Dato L un linguaggio accettato da una TM M multinastro (deterministica o meno) di complessità spaziale $S_M(n)$, allora, per ogni costante $c \in \mathbb{R}^+$, L è accettato anche da un'opportuna TM M' tale che $S_{M'}(n) < c \cdot S_M(n)$.*

Teorema 7.3.2. *Dato L un linguaggio accettato da una TM M multinastro (deterministica o meno) di complessità spaziale $S_M(n)$, allora L è accettato anche da un'opportuna TM M' multinastro con $k = 1$ con la medesima complessità spaziale, concatenando i contenuti dei k nastri di M.*

Teorema 7.3.3. *Dato L un linguaggio accettato da una TM M multinastro (deterministica o meno) di complessità spaziale $S_M(n)$, allora, per ogni costante $c \in \mathbb{R}^+$, L è accettato anche da un'opportuna TM M' multinastro con $k = 1$ tale che $S_{M'}(n) < c \cdot S_M(n)$.*

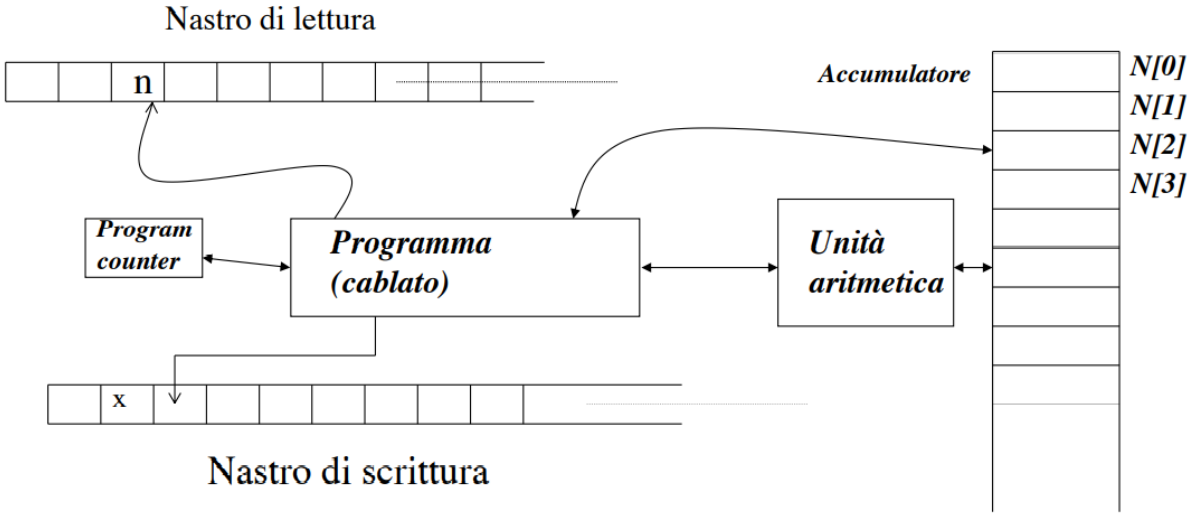
In generale, per la complessità temporale non si hanno risultati simili, ma è possibile formulare alcuni teoremi:

Teorema 7.3.4. *Dato L un linguaggio accettato da una TM M multinastro (deterministica o meno) di complessità temporale $T_M(n)$, allora, per ogni costante $c \in \mathbb{R}^+$, L è accettato anche da un'opportuna TM M' con k+1 nastri tale che $T_{M'}(n) = max\{n + 1, c \cdot T_M(n)\}$.*

I teoremi qui introdotti valgono anche per le moderne macchine di Von Neumann, in quanto possiamo avere speedup lineari arbitrariamente grandi (ovviamente, entro i limiti fisici della termodinamica), aumentando il parallelismo, ma miglioramenti più che lineari si possono ottenere solamente modificando l'algoritmo impiegato.

7.4 Macchina RAM

La macchina RAM (o Random Access Machine) è un modello classico ispirata all'architettura di Von Neumann. Tale macchina è costituita da un nastro in ingresso, un nastro in uscita, un programma rappresentato da un numero finito di istruzioni, un contatore che indica l'istruzione corrente da eseguire e una memoria ad accesso diretto.



Sia i nastri che la memoria sono composti da un numero illimitato di celle, ma al contrario dei nastri di ingresso e uscita che si possono accedere in maniera sequenziale, la memoria è indirizzata e si può accedere a una sua cella attraverso un numero intero $i > 0$ che indica l'indirizzo di tale cella di memoria. La cella 0 della memoria è un registro speciale, detto accumulatore, che si utilizza per contenere il valore

Tabella 7.1: istruzioni macchina RAM

ISTRUZIONE		SEMANTICA
LOAD=	x	M[0] $\leftarrow x$
LOAD	x	M[0] $\leftarrow$ M[x]
LOAD*	x	M[0] $\leftarrow$ M[M[x]]
STORE	x	M[x] $\leftarrow$ M[0]
STORE*	x	M[M[x]] $\leftarrow$ M[0]
ADD=	x	M[0] $\leftarrow$ M[0] + x
ADD	x	M[0] $\leftarrow$ M[0]+ M[x]
ADD*	x	M[0] $\leftarrow$ M[0] + M[M[x]]
SUB=	x	M[0] $\leftarrow$ M[0] - x
SUB	x	M[0] $\leftarrow$ M[0] - M[x]
SUB*	x	M[0] $\leftarrow$ M[0] - M[M[x]]
MULT=	x	M[0] $\leftarrow$ M[0] * x
MULT	x	M[0] $\leftarrow$ M[0] * M[x]
MULT*	x	M[0] $\leftarrow$ M[0] * M[M[0]]
DIV=	x	M[0] $\leftarrow$ M[0] $\setminus$ x
DIV	x	M[0] $\leftarrow$ M[0] $\setminus$ M[x]
DIV*	x	M[0] $\leftarrow$ M[0] $\setminus$ M[M[x]]
READ	x	M[x] $\leftarrow$ input
READ*	x	M[M[x]] $\leftarrow$ input
WRITE=	x	stampa x
WRITE	x	stampa M[x]
WRITE*	x	stampa M[M[x]]
JUMP	*label*	PC $\leftarrow$ *label*
JGZ	*label*	PC $\leftarrow$ *label* if M[0] > 0
JLZ	*label*	PC $\leftarrow$ *label* if M[0] < 0
JZ	*label*	PC $\leftarrow$ *label* if M[0] $= 0$
HALT		terminazione

di uno dei due operandi delle operazioni aritmetiche binarie che la macchina può effettuare. Un generico porgramma eseguibile dalla macchina RAM è composto da istruzioni riportate nella tabella 7.1.

Una volta introdotte tutte le istruzioni eseguibili dalla macchina RAM, è possibile ora studiarne la complessità temporale, come fatto per le TM a k-nastri. A differenza delle TM, nelle macchine RAM l'esecuzione delle diverse operazioni dipende dagli operandi necessari per eseguire tale operazione. Diventa quindi necessario analizzare tutte le istruzioni e definire il tempo richiesto per ciascuna di esse e la quantità di memoria allocata. Queste quantità possono essere calcolate secondo due criteri, ovvero tramite il criterio del costo costante e tramite il criterio del costo logaritmico. Il primo si basa sull'assunzione che l'esecuzione di ciascuna istruzione richieda un'unità di tempo e che ciascuna allocazione in memoria richieda un'unità di spazio (stessa assunzione fatta per le TM).

Si è appena affermato però che nella macchina RAM le istruzioni hanno diversa natura e manipolano dati di diversa dimensione: risulta, dunque, evidente che tale criterio è poco affine alla realtà. Per tener conto della differente velocità di esecuzione e della differente quantità di memoria allocata da ciascuna istruzione, si introduce il secondo criterio (del costo logaritmico), basato sulla supposizione che il tempo richiesto per eseguire un'istruzione sia proporzionale alla lunghezza degli operandi dell'istruzione considerata. Poichè gli operandi sono rappresentati in memoria in codice binario, un generico operando di valore v è rappresentato da $\lfloor log_2(|v|+1) \rfloor$.

Dunque, è possibile definire la funzione $l(i) = if\ i \neq 0\ then\ \lfloor log_2(|v|+1) \rfloor\ else\ 1$ tramite cui calcolare la complessità temporale logaritmica di ciascuna istruzione precedentemente analizzata nella tabella 7.1. Alla stessa maniera, è possibile calcolare i costi relativi allo spazio, introducendo la variabile m definita come l'indirizzo più alto della cella di memoria a cui si fa accesso durante l'esecuzione del programma, e la variabile M_i che rappresenta il valore assoluto più grande immagazzianto in M[i] durante l'esecuzione. La complessità spaziale logaritmica si definisce quindi con la seguente formula:

$$\sum_{i=0}^{m} l(M_i)$$

Di seguito sono riportati i costi logaritmici delle istruzioni RAM:

Tabella 7.2: costi logaritmici delle istruzioni macchina RAM

ISTRUZIONE		COSTO LOGARITMICO
LOAD=	x	$l(x)$
LOAD	x	$l(x) + l(M[x])$
LOAD*	x	$l(x) + l(M[x]) + l(M[M[x]])$
STORE	x	$l(M[0]) + l(x)$
STORE*	x	$l(M[0]) + l(x) + l(M[x])$
ADD=	x	$l(M[0]) + l(x)$
ADD	x	$l(M[0]) + l(x) + l(M[x])$
ADD*	x	$l(M[0]) + l(x) + l(M[x]) + l(M[M[x]])$
		SUB, MULT, DIV definite come ADD.
READ	x	$l(input) + l(x)$
READ*	x	$l(input) + l(x) + l(M[x])$
WRITE=	x	$l(x)$
WRITE	x	$l(x) + l(M[x])$
WRITE*	x	$l(x) + l(M[x]) + l(M[M[x]])$
JUMP	*label*	1
JGZ	*label*	$l(M[0])$
JLZ	*label*	$l(M[0])$
JZ	*label*	$l(M[0])$
HALT		1

Dunque, il criterio del costo costante si può applicare solo in situazioni in cui si prevede che ogni valore che comparirà durante l'esecuzione del programma occupi esattamente una cella di memoria, altrimenti si deve necessariamente applicare il criterio del costo logaritmico, che porta ad un calcolo più preciso della complessità.

7.5 Correlazione temporale fra TM e RAM

Una volta analizzato il comportamento della macchina RAM, è possibile studiarne la correlazione con le TM. Nello specifico, è possibile simulare una TM deterministica a k nastri attraverso una macchina RAM, nel seguente modo: innanzitutto, si considera la memoria della RAM come suddivisa in blocchi, tutti di dimensione k, ad eccezione del blocco 0, che ha dimensione $k+1$, in quanto memorizza lo stato della TM e le posizioni delle k testine. I successivi blocchi vengono impiegati per contenere i valori contenuti nelle successive posizioni di ciascuno dei k nastri di memoria della TM. Dunque, il valore rappresentato nella i-esima cella del j-esimo nastro della TM è contenuto nella locazione $c + k \cdot j + i$ in cui c è una costante opportuna della memoria della macchina RAM. Inoltre, per accedere al valore presente sotto la testina di lettura di un determinato nastro è prima necessario eseguire un accesso diretto al blocco 0, per poter trovare la posizione del nastro stesso. Poi, per eseguire la funzione di transizione $\delta(q, i, s_1, ..., s_k)$ e la funzione di uscita $\eta(q, i, s_1, ..., s_k)$, si richiedono un numero prefissato di accessi in memoria per ottenere $q, i, s_1, ..., s_n$, necessari per l'esecuzione di tali funzioni.

Tutto ciò conduce al seguente teorema:

Teorema 7.5.1. *Una TM multinastro con complessità temporale T_M può essere simulata da una macchina RAM con complessità temporale $T_R = \Theta(T_M)$, secondo il criterio di costo uniforme, oppure $T_R = \Theta(T_M \cdot log(T_M))$, secondo il criterio di costo logaritmico.*

Ovviamente è possibile anche simulare una macchina RAM tramite una macchina di Turing, ma tale costruzione è molto più complessa e richiede un'analisi approfondita. Si enuncia quindi solo il seguente teorema:

Teorema 7.5.2. *Sia L il linguaggio riconosciuto da una macchina RAM di complessità temporale T_R secondo il criterio del costo logaritmico. Se il programma RAM non utilizza le istruzioni $MULT$ e DIV, allora L può essere riconosciuto da un'opportuna TM multinastro, in un tempo $T_M = \Theta(T_R^2)$.*

Si può quindi osservare come il legame tra T_M e T_R sia di tipo polinomiale, implicazione molto importante perchè suggerisce quale sia la classe di problemi trattabili nella pratica.

Capitolo 8

Algoritmi

In questo capitolo si analizzano a fondo i principali algoritmi di ordinamento e i relativi tempi di esecuzione. Nello specifico, si utilizzerà come modello di riferimento la macchina RAM con un criterio di costo costante, come analizzato nei capitoli precedenti. Prima di proseguire nella trattazione è necessario dare una definizione generale di algoritmo:

Definizione 8.0.1. *Un algoritmo è una procedura di calcolo ben definita che prende un certo valore, o un insieme di valori, in input e genera un valore, o un insieme di valori, in output. Dunque, un algoritmo è una serie di passi computazionali che trasformano l'input in output.*

Un algoritmo può anche essere visto come uno strumento per la risoluzione di un problema computazionale ben definito: sotto questo sguardo, un algoritmo si definisce corretto se, per ogni istanza di input, termina con l'output corretto. Se un algoritmo è corretto, allora risolve quel determinato problema computazionale. Esistono molti modi per poter specificare un determinato algoritmo: si può utilizzare la lingua italiana o inglese, ma anche un linguaggio di programmazione come C, C++, JAVA e Pascal, o ancora tramite uno pseudocodice.

8.1 Pseudocodifica

La pseudocodifica può avvenire in molti modi, ma nel seguito si utilizzerano le convenzioni qui riportate:

- L'indentazione serve ad indicare la struttura a blocchi dello pesudocodice, in modo da comprendere quali istruzioni appartengono, per esempio, ad un ciclo ***for***, a un ciclo ***while*** o ad un ***if-else*** statement. Non sono utilizzate le parentesi graffe o parole chiave come begin ed end in quanto appesantiscono la sintassi;
- I costrutti iterativi ***while***, ***for***, ***repeat–until*** e il costrutto condizionale ***if–else*** hanno interpretazioni simili a quelle dei comuni linguaggi di programmazione. Il contatore del ciclo mantiene il suo valore dopo la fine del ciclo, quindi il valore che ha provocato la terminazione del ciclo stesso. Inoltre, si utilizza la parola chiave ***to*** quando il ciclo ***for*** incrementa il valore del suo contatore ad ogni iterazione, mentre si utilizza la parola chiave ***down to*** nel caso la variabile venga decrementata;
- Le assegnazioni di un valore ad una certa variabile avviene con il simbolo `:=`, differente dall'operatore `=`, che invece indica l'eguaglianza di due valori all'interno di un costrutto ***if***;
- Per identificare un elemento appartenente ad un array, si utilizza la notazione con le parentesi quadre, al cui interno si indica l'indice dell'elemento a cui si vuole accedere: `array[i]`; Per indicare un intervallo di valori all'interno dell'array si utilizza la seguente sintassi: `array[i..j]`, con cui si indica la sottomatrice composta dagli elementi compresi fra i e j;
- I dati utilizzati sono tipicamente organizzati in oggetti, formati da attributi, a cui si accede tramite la notazione punto: `oggetto.prop`. Le variabili che rappresentano un determinato oggetto sono trattate come puntatori a tale oggetto. Un puntatore che non fa riferimento ad alcun oggetto è inizializzato con il valore `NIL`;
- I parametri vengono passati ad una procedura per valore: la procedura chiamata riceve una sua copia dei parametri e, quindi, se a una di queste variabili è assegnato un nuovo valore, la modifica

non è visibile dalla procedura chiamante. Nel caso venga passato come argomento un oggetto, viene copiato il puntatore a tale oggetto e quindi le modifiche sono visibili anche dalla procedura chiamante;

- L'istruzione ***return*** restituisce immediatamente il controllo al punto in cui la procedura chiamante ha effettuato la chiamata. Le istruzioni ***return*** possono anche ritornare un valore al chiamante;
- Gli operatori booleani ***and*** e ***or*** sono cortocircuitati. Ciò significa che nella valutazione dell'espressione `x` ***and*** `y`, si valuta prima se il valore di `x` sia falso, in quanto, se lo fosse, l'intera espressione sarebbe falsa e non avrebbe quindi alcun senso valutare il valore della variabile `y`. Al contrario, nella valutazione dell'espressione `x` ***or*** `y`, si verifica innanzitutto se il valore di `x` sia vero, in quanto, se lo fosse, l'intera espressione sarebbe vera e non avrebbe quindi alcun senso valutare il valore della variabile `y`.

Tramite queste regole è possibile definire un generico algoritmo.

8.2 Insertion Sort

Una classe di algoritmi molto studiati è quella riguardante l'ordinamento di un vettore, che consiste nella disposizione dei suoi elementi in ordine crescente.

Il primo algoritmo analizzato è l'**insertion sort**, che prende in input una sequenza di n numeri $[a_1, a_2, ..., a_n]$ e restituisce in output una permutazione $[a'_1, a'_2, ..., a'_n]$ tale che $a'_1 \leq a'_2 \leq ... \leq a'_n$. Questo algoritmo ordina sul posto [1] gli elementi assumendo che la sequenza da ordinare sia inizialmente partizionata in una sottosequenza già ordinata, all'inizio composta da un unico elemento (il primo dell'array), e una sottosequenza ancora da ordinare. Ad ogni iterazione viene rimosso un elemento dalla sottosequenza non ordinata e inserita nella posizione corretta all'interno della sottosequenza già ordinata.
In pseudocodice:

```
1 insertionSort(A):
2   for j := 2 to A.length:
3     key := A[j]
4     i := j - 1
5     while i > 0 and A[i] > key:
6       A[i + 1] := A[i]
7       i := i - 1
8     A[i + 1] = key
```

All'inizio di ogni iterazione del ciclo ***for***, il cui indice è j, la sottosequenza di elementi `A[1..j-1]` è la parte ordinata dell'array, mentre la sottosequenza `A[j+1..n]` è costituita da elementi ancora da ordinare.

Si analizza ora il tempo di esecuzione della procedura `insertion sort`: per ogni $j = 2, 3, ..., n$ in cui $n =$ `A.length`, si indica con t_j il numero di volte che il test del ciclo ***while*** nella riga 5 viene eseguito per quel determinato valore di j.

Codice	Costo	Numero di volte
for `j:= 2` ***to*** `A.length`	c_1	n
`key := A[j]`	c_2	$n-1$
`i := j - 1`	c_3	$n-1$
while `i > 0` ***and*** `A[i] > key`	c_4	$\sum_{j=2}^{n} t_j$
`A[i + 1] := A[i]`	c_5	$\sum_{j=2}^{n} (t_j - 1)$
`i := i - 1`	c_6	$\sum_{j=2}^{n} (t_j - 1)$
`A[i + 1] := key`	c_7	$n-1$

Ad ogni riga di codice viene associato un costo c_i che va moltiplicato per il numero di volte che tale riga viene eseguita. Il tempo totale di esecuzione si calcola, dunque, sommando i vari contributi di tempo di ogni riga, ottenendo così l'espressione di $T(n)$:

[1] L'algoritmo risistema gli elementi della sequenza all'interno dell'array avendo, in ogni istante, al più un numero finito di elementi memorizzati all'esterno dell'array: ciò permette di risparmiare memoria nel calcolatore.

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Ovviamente, il caso migliore si verifica quando l'array in input è già ordinato. In questo caso, $t_j = 1 \ \ \forall j = 2, 3..., n$ e l'espressione di $T(n)$ assume la forma:

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

che è funzione lineare di n. Dunque, $T(n) = \Theta(n)$.

Al contrario, il caso pessimo si verifica quando l'array in input è ordinato, ma in ordine decrescente. In questo caso $t_j = j \ \ \forall j = 2, 3..., n$ e l'espressione di $T(n)$ assume la forma:

$$T(n) = \frac{1}{2}(c_4 + c_5 + c_6)n^2 + (c_1 + c_2 + c_3)n + \frac{1}{2}(c_4 - c_5 - c_6 + c_8)n - (c_2 + c_3 + c_4 + c_7)$$

che è funzione quadratica di n. Dunque, $T(n) = \Theta(n^2)$.

8.3 Merge Sort

L'algoritmo appena analizzato utilizza un approccio di tipo incrementale: dopo aver ordinato il sottoarray `A[1..j-1]` inserisce l'elemento `A[j]` nella posizione corretta, ottenendo il sottoarray ordinato `A[1..j]`. Nel seguito, invece, si analizza un secondo approccio, più efficiente del primo, soprattutto per array di molti elementi: Divide et Impera. Questo criterio si basa sulla suddivisione ricorsiva del problema in sottoproblemi più piccoli, simili a quello originario, ma di dimensione ridotta, per poi risolvere i sottoproblemi di dimensione minima e fondere i risultati ottenuti, per costruire una soluzione generale del problema originario.

Il paradigma Divide et Impera, si basa in realtà su tre passaggi:

1. Divide: il problema viene suddiviso in un certo numero di sottoproblemi, che sono istanze più piccole del problema originario, fino ad ottenere sottoproblemi minimi, non più divisibili;

2. Impera: i sottoproblemi di dimensione minima vengono risolti in maniera ricorsiva; se i problemi hanno dimensione sufficientemente piccola vengono risolti direttamente;

3. Combina: le soluzioni dei sottoproblemi vengono combinate per generare la soluzione del problema generale.

Un tipico algoritmo che segue questo metodo di risoluzione è il **merge sort**, che suddivide l'array originario a metà e ordina ricorsivamente i due sottoarray ottenuti, chiamando sè stesso fino ad ottenere sequenze di dimensione uno, di per sè già ordinate. A questo punto, le sottosequenze vengono fuse in modo da ottenere un array ordinato.

Quest'ultimo passaggio viene effettuato tramite una procedura ausiliaria `merge(A,p,q,r)`, dove A è un array, e p, q, r sono tre indici dell'array tali che $p \leq q < r$. La porcedura assume che le sottosequenze `A[p..q]` e `A[q+1..r]` siano ordinate e, quindi, le fonde per formare un unico sottoarray ordinato che sostituisce il sottoarray corrente `A[p..r]`. La procedura `merge(A,p,q,r)` impiega un tempo $\Theta(n)$ con $n = r - p + 1$ il numero di elementi da fondere. Ad ogni iterazione, la procedura `merge` confronta gli elementi più piccoli dei due sottoarray, inserendoli nel sottoarray "successivo" fino a quando uno dei due sottoarray è vuoto: a quel punto, i restanti elementi del sottoarray rimanente vengono copiati per completare l'array "successivo". Da un punto di vista computazionale, ogni iterazione della procedura impiega un tempo costante, in quanto deve semplicemente confrontare i due elementi dei due sottoarray. Poichè tale procedura viene effettuata per un massimo di n volte, la funsione impiega un tempo $\Theta(n)$.

In pseudocodice:

```
1 merge(A, p, q, r):
2   n1 := q - p + 1
3   n2 := r - q
4   L := [1 .. n1 + 1]
5   R := [1 .. n2 + 1]
```

```
6    for i := 1 to n_1:
7      L[i] := A[p + i - 1]
8    for j := 1 to n_2:
9      R[j] := A[q + j]
10   L[n_1 + 1] := ∞
11   L[n_2 + 1] := ∞
12   i := 1
13   j := 1
14   for k := p to r:
15     if L[i] <= R[j]:
16       A[k] := L[i]
17       i := i + 1
18     else:
19       A[k] := R[j]
20       j := j + 1
```

In altri termini, le righe 2 e 3 inizializzano i valori di n_1 ed n_2, che rappresentano la lunghezza dei due sottoarray `A[p..q]` e `A[q+1..r]`. Nelle due righe successive vengono creati i due sottoarray ausiliari `L` (per Left) ed `R` (per Right), che contano $n + 1$ elementi (per motivi che verranno chiariti a breve). Le righe dalla 6 alla 9, inizializzano gli array appena creati con i valori contenuti rispettivamente nella prima e nella seconda metà dell'array `A`. Le righe 10 e 11 inizializzano l'ultimo ($n+1$ -esimo) elemento dei due sottoarray `L` ed `R`, con un valore sentinella. Impostando tale valore ad ∞, si è certi che non possa essere il valore più piccolo fra i due confrontati: in questo modo, una volta arrivati alla fine di uno dei due sottoarray, gli elementi dell'altro vengono ricopiati nell'array "successivo" in quanto necessariamente più piccoli di ∞. Le ultime righe (dalla 12 alla 20) implementano la logica del confronto e dell'insermento dell'elemento correntemente più piccolo nell'array `A`.

Una volta analizzata la procedura `merge`, si può introdurre l'algoritmo di ordinamento `mergeSort`. In pseudocodice:

```
1  mergeSort(A, p, r):
2    if p < r:
3      q = ⌊ (p + r) / 2 ⌋
4      mergeSort(A, p, q)
5      mergeSort(A, q+1, r)
6      merge(A, p, q, r)
```

L'algoritmo calcola, in riga 2, un indice `q`, che serve a suddividere l'array `A` in due sottoarray che contengono rispettivamente $\lceil n/2 \rceil$ elementi ed $\lfloor n/2 \rfloor$ elementi, su cui richiama ricorsivamente sè stessa. Una volta suddiviso l'array `A` in sottoarray di dimensione minima, viene chiamata la procedura `merge`, precedentemente analizzata.

Come si può facilmente osservare, la procedura `mergeSort` è definita in maniera ricorsiva, quindi l'analisi delle prestazioni temporali diventa leggermente più complessa: infatti, si deve necessariamente far uso di un'equazione di ricorrenza, che esprime il tempo di esecuzione totale di un problema di dimensione n, in funzione del tempo di esecuzione per input più piccoli. Se la dimensione del problema diventa sufficientemente piccola, per esempio $n \leq c$ per qualche costante c, la soluzione del problema è diretta e richiede un tempo di esecuzione costante, indicata con $\Theta(1)$. Si suppone, inoltre, che il problema originario venga suddiviso in a sottoproblemi, tutti di dimensione $1/b$ volte la dimensione del problema originario. Dunque, è necessario un tempo $T(n/b)$ per risolvere un sottoproblema di dimensione n/b e un tempo $aT(n/b)$ per risolverli tutti. Infine, se si impiega un tempo $D(n)$ per suddividere il problema in a sottoproblemi e un tempo $C(n)$ per fonderne le soluzioni, si ottiene la ricorrenza:

$$T(n) = \begin{cases} \Theta(1) & if\ n \leq c \\ aT(n/b) + D(n) + C(n) & else \end{cases}$$

Per trovare ora il tempo di esecuzione $T(n)$ nel caso peggiore si può ragionare come segue. Nel caso in cui i sottoarray abbiano cardinalità uno, la soluzione è diretta, quindi viene impiegato un tempo costante per risolvere il problema, mentre se i sottoarray hanno $n > 1$ elementi, si suddivide il tempo di esecuzione impostando $D(n) = \Theta(1)$, in quanto si impiega un tempo costante per calcolare il centro di un array,

$C(n) = \Theta(n)$, in quanto si è già precedentemente dimostrato che la procedura `merge` impieghi un tempo lineare per la fusione delle soluzioni, e infine si pone $a = b = 2$, in quanto si suddivide ricorsivamente il problema in due sottoproblemi di uguale dimensione [2]. Con questo ragionamento, la ricorrenza assume l'espressione:

$$T(n) = \begin{cases} \Theta(1) & if\ n = 1 \\ 2T(n/2) + \Theta(n) + \Theta(1) & if\ n > 1 \end{cases}$$

Si può facilmente dimostrare (analiticamente oppure tramite il teorema dell'espreto, di cui si discuterà successivamente) che tale equazione ha soluzione $T(n) = \Theta(n\, log_2\, n)$, che rappresenta il tempo di esecuzione dell'algoritmo `mergeSort` nel caso pessimo. Si può osservare come tale algoritmo sia decisamente migliore rispetto all'`insertionSort`, il cui tempo di esecuzione nel caso passimo è $\Theta(n^2)$.

Un modo per comprendere meglio come mai la complessità temporale del `mergeSort` sia proprio $\Theta(n\, log_2\, n)$, si riscrive la ricorrenza nel seguente modo:

$$T(n) = \begin{cases} c & if\ n = 1 \\ 2T(n/2) + cn + c & if\ n > 1 \end{cases}$$

in cui la costante c rappresenta sia il tempo richiesto per risolvere i problemi di dimensione 1, sia il tempo per elemento dell'array dei passi divide e combina. Si può costruire un albero di ricorsione, in cui ogni ramo rappresenta una metà dell'array precedente e ogni foglia sia un array di dimensione unitaria. Il primo livello (in alto) ha un costo totale di cn, il secondo livello ha un costo totale di $cn/2 + cn/2 = cn$ e così via fino all'ultimo livello, con costo totale di $n + n + ... + n$ (c volte), quindi di cn. In generale, il livello i ha 2^i nodi, ciascuno dei quali ha un costo di $c(n/2^i)$, quindi, il numero totale di livelli dell'albero di ricorsione è $log_2\, n + 1$, con n la dimensione dell'input. Dunque, per calcolare il costo totale, basta sommare i costi di tutti i livelli, ottenendo $cn(log_2\, n + 1) = cn(log_2\, n) + cn$, ovvero $\Theta(n\, log_2\, n)$.

8.4 Risoluzione Ricorrenze

Come già detto in precedenza, quando i problemi sono abbastanza grandi da essere risolti ricorsivamente, si ha il cosiddetto caso ricorsivo, tramite cui si divide il problema in problemi più piccoli di uguale natura. Una volta che i sottoproblemi diventano sufficientemente piccoli da non richiedere più il passo ricorsivo, si è raggiunto il cosiddetto caso base, da cui inizia la soluzione del problema.

Questo modello di risoluzione del problema viene anche detto Divide et Impera e richiede l'utilizzo di equazioni di ricorrenza, tramite cui si caratterizzano i tempi di esecuzione degli algoritmi in termini dei loro valori con input più piccoli. Per risolvere tali equazioni, ovvero per trovare i limiti asintotici Θ oppure O, esistono tre metodi:

1. Metodo di Sostituzione: si fa un'ipotesi di soluzione e si utilizza l'induzione matematica per dimostrare che l'ipotesi sia corretta;
2. Metodo dell'Albero di ricorsione: si converte la ricorrenza in una struttura ad albero, i cui nodi rappresentano i costi ai vari livelli della ricorsione;
3. Metodo dell'Esperto (Master theorem): fornisce i limiti per ricorrenze nella forma $T(n) = aT(n/b) + f(n)$ con $a \geq 1, b > 1$ e $f(n)$ data. Una ricorrenza in questa forma caratterizza un algoritmo divide et impera che crea a sottoproblemi di dimensione $1/b$, i cui passi divide e combina richiedono un tempo $f(n)$.

A volte, le ricorrenze non saranno delle uguaglianze, ma delle disuguaglianze nella forma $T(n) \leq ...$, che stabilisce un limite superiore su $T(n)$ (quindi si utilizza la notazione O anzichè Θ), oppure nella forma $T(n) \geq ...$, che stabilisce invece un limite inferiore su $T(n)$ (quindi si utilizza la notazione Ω anzichè Θ). Inoltre, ci sono casi in cui si trascurano dei dettagli tecnici di poca importanza, come le condizioni al contorno: infatti, poichè il tempo di esecuzione di un algoritmo con un input di dimensione costante è costante, le ricorrenze che ne derivano hanno $T(n) = \Theta(1)$, per valori sufficientemente piccoli di n. Questa decisione risiede nel fatto che, sebbene le condizioni al contorno cambiano la soluzione esatta della ricorrenza, tuttavia la soluzione non cambia per più di un fattore costante e quindi asintoticamente rimane immutata.

[2] In realtà, sarebbe più accurato scrivere $T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil)$ in quanto non sempre la dimensione dell'array `A` è potenza di 2 e, dunque, divisibile ricorsivamente in due metà. Tale approssimazione, comunque, non influisce sulla complessità finale del calcolo.

8.4.1 Metodo di Sostituzione

Uno dei metodi per la risoluzione delle occorrenze e, quindi, per il calcolo del tempo di esecuzione degli algoritmi, è il metodo della sostituzione, che richiede due passaggi:

1. Ipotizzare la forma della soluzione;

2. Utilizzare l'induzione matematica per dimostrare che la soluzione ipotizzata sia corretta.

Questo metodo può essere applicato solamente se si ha un'idea della forma generale della soluzione e si vuole calcolare il limite superiore o inferiore della ricorrenza che si analizza.

ESEMPIO: Si determini il limite superiore della ricorrenza $T(n) = 2T(\lfloor n/2 \rfloor) + n$.

Si suppone che la soluzione sia $O(n\,log_2\,n)$. Il metodo di sostituzione consiste nel dimostrare che $T(n) \leq c\,n\,log_2\,n$ per un generico $c > 0$. Si verifica, innanzitutto, che questo limite sia valido anche per $\lfloor n/2 \rfloor$, ovvero che $T(\lfloor n/2 \rfloor) \leq c\,\lfloor n/2 \rfloor\,log_2(\lfloor n/2 \rfloor)$. Facendo le opportune sostituzioni si ha:

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor log_2(\lfloor n/2 \rfloor)) + n \\ &\leq c\,n\,log_2(n/2) + n \\ &= c\,n\,log_2\,n - c\,n\,log_2\,2 + n \\ &= c\,n\,log_2\,n - c\,n + n \\ &\leq c\,n\,log_2\,n \end{aligned}$$

L'ultimo passaggio è vero solo per $c \geq 1$. A questo punto, l'induzione matematica richiede di dimostrare che la soluzione vale per le condizioni al contorno. Si suppone, per esempio, che l'unica condizione al contorno sia $T(1) = 1$: si deve dimostrare che è possibile scegliere una costante c sufficientemente grande in modo che il limite $T(n) \leq c\,n\,log_2\,n$ sia valido anche per le condizioni al contorno. Quindi per $n = 1$ (condizione al contorno), il limite $T(n) \leq c\,n\,log_2\,n$ diventa $T(1) \leq c\,log_2\,1 = 0$, che però è in contrasto con $T(1) = 1$: il caso base della dimostrazione induttiva non è valido!

Questo ostacolo nella dimostrazione può essere facilmente superato sfruttando la notazione asintotica, che richiede di provare che $T(n) \leq c\,n\,log_2\,n$ sia valida solamente dopo un certo n_0 in poi, scelto arbitrariamente: l'idea è quella di escludere la condizione al contorno dalla dimostrazione induttiva. Si osservi che, per $n \geq 3$, la ricorrenza non dipende direttamente da $T(1)$, quindi si può sostituire con $T(2)$ e $T(3)$, impiegati come casi base della dimostrazione induttiva. Inoltre, ponendo $n_0 = 2$, se $T(1) = 1$ allora $T(2) = 4$ e $T(3) = 5$. Basta quindi determinare una costante c tale per cui $T(2) = 4 \leq 2c\,log_2(2)$ e $T(3) = 5 \leq 3c\,log_2(3)$: le precedenti condizioni sono soddisfatte solo per $c \geq 2$.

Non esiste un metodo unico e generale per indovinare la soluzione corretta di una ricorrenza, ma è possibile formulare delle buone ipotesi tramite il metodo dell'albero di ricorsione. Inoltre, se una ricorrenza è simile ad una già risolta in precedenza, allora è possibile che anche la soluzione sia analoga. Un altro metodo per formulare un'ipotesi di soluzione consiste nel dimostrare dei limiti superiori e inferiori molto generali e larghi, per poi ridurre gradualmente il grado di incertezza, aumentando il limite inferiore e diminuendo il limite superiore.

Ci sono poi casi in cui la soluzione ipotizzata sembra essere corretta, ma i calcoli matematici non soddisfano il passo induttivo: solitamente, il problema risiede nel fatto che l'ipotesi induttiva non è abbastanza forte per dimostrare il limite esatto. In un caso del genere, spesso è necessario semplicemente correggere l'ipotesi sottraendo un termine di ordine inferiore per fare in modo che i calcoli soddisfino i requisiti.

ESEMPIO: Si calcoli la ricorrenza $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$ supponendo che la soluzione sia $T(n) = O(n)$. Si deve quindi dimostrare che $T(n) \leq cn$ per qualche c arbitraria. Sostituendo l'ipotesi all'interno della ricorrenza si ottiene:

$$\begin{aligned} T(n) &\leq c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 \\ &= c\,n + 1 \end{aligned}$$

che non implica che $T(n) \leq c\,n$ per qualunque valore di c. Sembrerebbe quindi che l'ipotesi fatta sia sbagliata, ma al contrario si può dimostrare che è corretta, formulando un'ipotesi induttiva più forte. Per affrontare tale problema, si sottrae un termine di ordine inferiore dalla precedente ipotesi, ad esempio,

un termine costante $d \geq 0$, ottenendo come nuova ipotesi $T(n) \leq c\,n - d$, che sostituita alla ricorrenza:

$$\begin{aligned} T(n) &\leq (c\lfloor n/2 \rfloor - d) + (c\lceil n/2 \rceil - d) + 1 \\ &= cn - 2d + 1 \\ &\leq cn - d \end{aligned}$$

che diventa valida per ogni $d \geq 1$. Come prima, la costante c deve essere scelta arbitrariamente grande affinchè siano soddisfatte le condizioni al contorno.

Infine, ci sono casi in cui tramite una piccola manipolazione algebrica è possibile rendere una ricorrenza ignota simile ad una più familiare.

ESEMPIO: Si calcoli la ricorrenza $T(n) = 2T(\lfloor \sqrt{n} \rfloor) + log_2(n)$. Tale ricorrenza sembra molto complessa da risolvere, ma è possibile semplificarla ponendo $m = log_2 n$, ottenendo così
$T(2^m) = 2T(2^{m/2}) + m$. Chiamando $S(m)$ la ricorrenza appena ottenuta, è possibile scrivere $S(m) = 2S(m/2) + m$, simile alla precedente ricorrenza analizzata $T(n) = 2T(\lfloor n/2 \rfloor) + n$; in effetti, la soluzione della ricorrenza $S(m)$ è la stessa ottenuta in precedenza. Dunque, la soluzione è $S(m) = m\,log_2\,m$ e, ripristinando i termini con la sostituzione $m = log_2 n$, si ottiene che $T(n) = O(log_2 n \cdot log_2(log_2 n))$.

8.4.2 Metodo dell'Albero di Ricorsione

Dato che spesso è complesso fomulare un'ipotesi di soluzione per una data ricorrenza, è possibile utilizzare il metodo dell'albero di ricorsione, in cui ogni nodo rappresenta il costo di un singolo sottoproblema. Sommando i costi dei nodi di ogni livello, si ottengono i costi relativi a quel livello e, sommando tali costi, si ottiene il costo generale della ricorrenza, che rappresenta l'ipotesi da verificare con il metodo della sostituzione. Utilizzando questo metodo, si tollera un certo livello di approssimazione, in quanto è interessante analizzare solamente il comportamento asintotico della ricorrenza: si possono quindi eliminare gli operatori 'ceil' e 'floor' e fare delle ipotesi blande per semplificare i calcoli.

ESEMPIO: Si calcoli la ricorrenza $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. Come detto, si può approssimare la ricorrenza eliminando l'operatore floor, ottenendo $T(n) = 3T(n/4) + cn^2$, per una data costante $c > 0$. Per comodità, si suppone anche che n sia una potenza di 4, in modo tale che ogni livello dell'albero abbia dimensione intera. Si ottiene così il seguente albero delle ricorrenze:

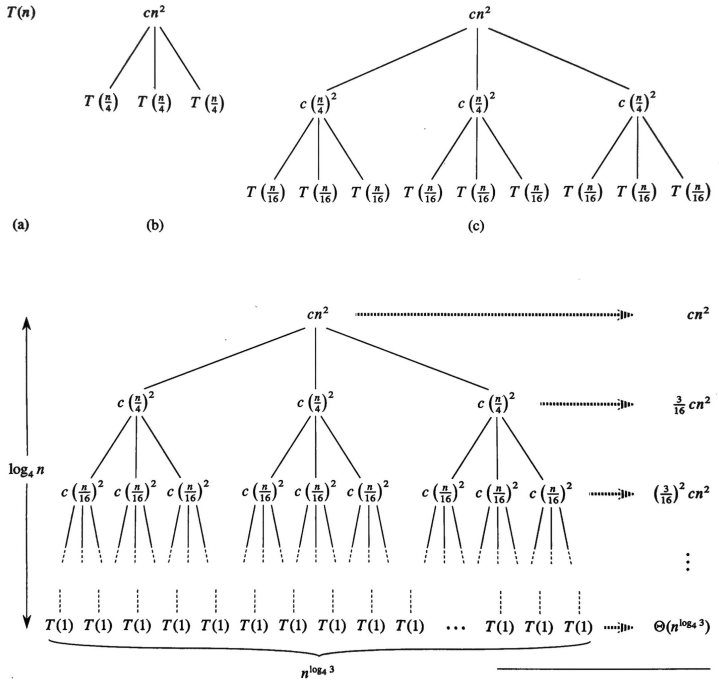


Figura 8.1: Albero della ricorrenza $T(n) = 3T(n/4) + cn^2$

La parte (a) della figura mostra $T(n)$, che viene espanso nella parte (b) in un albero equivalente che rappresenta la ricorrenza. Il termine cn^2 nella radice di quest'albero rappresenta il costo al livello

più alto della ricorsione, mentre i tre sottoalberi rappresentano i costi richiesti dai tre sottoproblemi di dimensione $n/4$. La parte (c) mostra l'espansione dei nodi di costo $T(n/4)$ dalla parte (b), in cui ogni nodo figlio ha costo $c(n/4)^2$. Tale processo viene ripetuto più e più volte fino ad ottenere i casi base, rappresentati nella parte (d) con $T(1)$.

La dimensione dei sottoproblemi per i nodi alla profondità i è di $n/4^i$, quindi la dimensione del sottoproblema diventa 1 (dimensione delle foglie) quando $(n/4)^i = 1$, ovvero quando $i = log_4(n)$: dunque, l'albero della ricorrenza ha esattamente $log_4 n+1$ livelli. Ora, per determinare il costo di ogni livello, basti pensare che ogni nodo dell'albero genera tre sottonodi e che, dunque, il numero di nodi alla profondità i è 3^i. Moltiplicando il risultato appena ottenuto con il costo di un singolo nodo, si ottiene che ogni livello ha un costo di $3^i c(n/4^i)^2 = (3/16)^i cn^2$. L'ultimo livello dell'albero conta $n^{log_4 3}$ nodi, ognuno di costo $T(1)$, per un costo totale di $n^{log_4 3}T(1)$, ovvero $\Theta(n^{log_4 3})$.

A questo punto, si sommano i contributi di ogni livello, ottenendo:

$$\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + ... + \left(\frac{3}{16}\right)^{log_4 n-1} cn^2 + \Theta(n^{log_4 3}) \\
&= \sum_{i=0}^{log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{log_4 3}) \\
&\overset{*}{<} \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{log_4 3}) \\
&= \frac{1}{1-(3/16)}cn^2 + \Theta(n^{log_4 3}) \\
&= \frac{16}{13}cn^2 + \Theta(n^{log_4 3}) \\
&= O(n^2).
\end{aligned}$$

Il passaggio segnato con * rappresenta una piccola approssimazione: la $\sum_{i=0}^{log_4 n-1} \left(\frac{3}{16}\right)^i cn^2$ ammette come limite superiore $\sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2$, che rappresenta una serie geometrica decrescente infinita. In questo modo è possibile proseguire con agilità i calcoli, ottenendo come ipotesi $T(n) = O(n^2)$, che dovrà essere verificata con il metodo della sostituzione.

8.4.3 Metodo dell'Esperto

Il metodo dell'esperto è impiegato per la risoluzione di ricorrenze del tipo $T(n) = aT(n/b) + f(n)$, con $a \geq 1, b > 1$ costanti ed $f(n)$ una funzione asintoticamante positiva. Una ricorrenza di questo tipo rappresenta il tempo di esecuzione di un algoritmo che divide il problema di dimensione n in a sottoproblemi di dimensione n/b, mentre la funzione $f(n)$ rappresenta il costo di divisione del problema e di combinazione delle soluzioni. Il metodo dell'esperto dipende dal seguente teorema:

Teorema 8.4.1 (Master Theorem)**.** *Date le costanti $a \geq 1$, $b > 1$ e la funzione $f(n)$, se la ricorsione $T(n)$ si presenta nella forma $T(n) = aT(n/b) + f(n)$, allora può essere limitata asintoticamente nei seguenti modi:*

1. *Se $f(n) = O(n^{log_b a-\varepsilon})$ per qualche $\varepsilon > 0$, allora $T(n) = \Theta(n^{log_b a})$;*
2. *Se $f(n) = \Theta(n^{log_b a})$, allora $T(n) = \Theta(n^{log_b a} log_2(n))$;*
3. *Se $f(n) = \Omega(n^{log_b a+\varepsilon})$ per qualche $\varepsilon > 0$ e se $af(n/b) \leq cf(n)$ per qualche $c < 1$ e per ogni n sufficientemente grande, allora $T(n) = \Theta(f(n))$.*

Si osservi che in ciascuno dei tre casi, si confronta la funzione $f(n)$ con la funzione $n^{log_b a}$: intuitivamente, la soluzione della ricorrenza è determinata dalla funzione polinomialmente [3] più grande. Se la funzione $n^{log_b a}$ è più grande polinomialmente, come nel caso uno, allora sarà soluzione della ricorrenza, altrimenti la soluzione sarà $f(n)$, come enunciato nel caso tre, in cui si deve anche verificare la condizione di regolarità della funzione. Nel caso due, in cui le due funzioni sono asintoticamente uguali, si moltiplicano entrambi i membri per un fattore logaritmico e la soluzione sarà $T(n) = \Theta(n^{log_b a} log_2(n)) = \Theta(f(n) log_2(n))$.

[3] Una funzione è polinomialmente più grande rispetto ad un'altra funzione se la prima è asintoticamente più grande della seconda di un fattore n^ε per qualche $\varepsilon > 0$.

I tre casi, sfortunatamente, non coprono tutte le funzioni $f(n)$ possibili, in quanto ci sarà un intervallo fra i casi 1 e 2, in cui la funzione $f(n)$ è minore di $n^{log_b a}$, ma non polinomialmente, mentre ci sarà anche un intervallo fra i casi 2 e 3, in cui la funzione $f(n)$ è maggiore di $n^{log_b a}$, ma non polinomialmente. In questi casi, il teorema dell'esperto non può essere applicato.

Per utilizzare il teorema enunciato, bisogna semplicemente determinare in quali dei tre casi rientra la funzione $f(n)$ e confrontarla con la funzione $n^{log_b a}$.

ESEMPIO: Si determini la soluzione della ricorrenza $T(n) = 9T(n/3) + n$. In questo caso, si ha che $a = 9, b = 3$ ed $f(n) = n$ e quindi $n^{log_b a} = n^{log_3(9)} = \Theta(n^2)$. Dato che $f(n) = O(n^{log_3(9)-\varepsilon})$, con $\varepsilon = 1$ (in quanto $f(n) = n$), si può applicare il caso 1 del teorema dell'esperto e concludere immediatamente che la soluzione della ricorrenza è $T(n) = \Theta(n^2)$, in quanto n^2 è polinomialmente più grande di n.

ESEMPIO: Si determini la soluzione della ricorrenza $T(n) = 2T(n/2) + n\,log_2\,n$. In questo caso, si ha che $a = 2, b = 2$ ed $f(n) = n\,log_2\,n$ e quindi $n^{log_b a} = n^{log_2(2)} = n$. Si potrebbe erroneamente pensare di essere nel terzo caso del teorema dell'esperto, ma le due funzioni non sono polinomialmente comparabili quindi non si può applicare il teorema. La ricorrenza, dunque, deve necessariamente essere risolta con l'utilizzo dei metodi precedentemente analizzati.

8.5 Heap Sort

Analizzando l'algoritmo Merge Sort si è constatato che è efficiente dal punto di vista temporale, ma non dal punto di vista spaziale, in quanto occupa una grande quantità di memoria. A questo proposito, si analizza ora l'algoritmo **Heap Sort**, che effettua un ordinamento sul posto degli elementi utilizzando una struttura dati detta Heap ('mucchio'), per la gestione delle informazioni.

Un Heap (binario) è una struttura dati ad albero binario quasi completo [4], in cui ogni nodo rappresenta un elemento dell'array da ordinare. Nello specifico, `A[1]` è la radice dell'albero, e per ogni elemento `A[i]`, `A[2i]` e `A[2i+1]` rappresentano i figli del nodo, mentre `A[`$\lfloor n/2 \rfloor$`]` rappresenta il nodo padre. Si possono quindi definire le seguenti procedure:

```
1 parent(i):
2   return ⌊i/2⌋
```

```
1 left(i):
2   return 2i
```

```
1 right(i):
2   return 2i + 1
```

Oltre all'attributo `A.length`, che ne ritorna la lunghezza, l'array `A` possiede in questo caso anche l'attributo `A.heapSize`, che indica il numero degli elementi dell'heap che sono registrati nell'array `A`. In altre parole, anche se l'array contiene n elementi, con n=`A.length`, soltanto gli elementi in `A[1..A.heapSize]`, con $0 \leq$ `A.heapSize` $\leq$ `A.length`, sono elementi validi dell'heap.

Esistono, inoltre, due tipologie di heap binari: max-heap e min-heap. Il primo, più importante, è costruito in modo tale che ogni nodo rispetti la condizione per cui `A[parent(i)]` $\geq$ `A[i]`; dunque, il valore di un nodo è al massimo il valore del nodo padre e, di conseguenza, l'elemento più grande di un max-heap è memorizzato alla sua radice. Il secondo, meno utilizzato, è costruito in modo tale che ogni nodo rispetti la condizione per cui `A[parent(i)]` $\leq$ `A[i]`.

Per implementare l'algoritmo heapsort si fa utilizzo del max-heap; per poterne mantenere le proprietà si utilizza la procedura di supporto `maxHeapify`, un algoritmo che prende in input un array `A` e un suo indice `i`, e restituisce l'array ordinato in modo tale da rappresentare il max-heap. Quando tale procedura viene invocata, essa assume che gli alberi binari di radici `left(i)` e `right(i)` siano dei max-heap, ma assume anche che `A[i]` possa essere più piccolo dei suoi figli, violando la proprietà fondamentale. La procedura, quindi, ha il compito di far 'scendere' il valore `A[i]` in modo tale che il sottoalbero con radice di indice i diventi un max-heap.

[4] Un albero binario quasi completo è una struttura dati ad albero in cui ogni livello è completo, eccetto per al più l'ultimo livello, che potrebbe essere completo solo fino ad un certo punto da sinistra

In pseudocodice:

```
maxHeapify(A, i):
  l := left(i)
  r := right(i)
  if l <= A.heapSize and A[l] > A[i]:
    max := l
  else:
    max := i
  if r <= A.heapSize and A[r] > A[max]:
    max := r
  if max != i:
    swap A[i] with A[max]
    maxHeapify(A, max)
```

A ogni passo viene determinato il più grande degli elementi `A[i]`, `A[left(i)]` e `A[right(i)]` e il suo indice viene memorizzato nella variabile `max`. Se `A[i]` è l'elemento più grande, allora il sottoalbero è già un max-heap e la procedura termina la propria esecuzione, altrimenti uno dei due figli contiene l'elemento più grande e `A[i]` viene scambiato con `A[max]` (***swap with***); in questo modo, il nodo di indice i e i suoi figli soddisfano la proprietà di max-heap. Il nodo con indice massimo, però, presenta il valore originale di `A[i]` e, quindi, il sottoalbero di radice `max` potrebbe violare la proprietà fondamentale: quindi, la procedura viene chiamata ricorsivamente sul sottoalbero, fino a raggiungere le foglie.

Questa procedura viene eseguita in un tempo $O(h)$, con h l'altezza dell'albero. Essendo l'albero quasi completo, $h = O(log\, n)$, quindi $T(n) = O(log\, n)$.

Ora, tramite la procedura `maxHeapify` è possibile convertire un array `A[1..n]` (con n=`A.length`) in un max-heap. Prima di procedere, è importante osservare come tutti gli elementi `A[`$\lfloor n/2 \rfloor + 1$ `.. n]` siano foglie dell'albero e, quindi, ciascuno di essi è un heap di un solo elemento, che si può utilizzare come punto di partenza per la costruzione dell'heap. Si introduce, dunque, la procedura `buildMaxHeap`, che attraversa i nodi restanti dell'albero ed esegue la procedura `maxHeapify` in ciascuno di essi.

In pseudocodice:

```
buildMaxheap(A):
  A.heapSize := A.length
  for i := ⌊A.length / 2 ⌋ down to 1:
    maxHeapify(A, i)
```

Si può dimostrare che tale procedura impiega un tempo di esecuzione $T(n) = O(n)$.

A questo punto, è possibile scrivere l'algoritmo `heapSort`. In pseudocodice:

```
heapSort(A):
  buildMaxheap(A)
  for i := A.length down to 2:
    swap A[1] with A[i]
    A.heapSize := A.heapSize - 1
    maxHeapify(A, 1)
```

Questo algoritmo si basa sul fatto che, una volta riordinato l'array in maniera che rappresenti un max-heap, l'elemento più grande dell'array si trova in `A[1]`: questo elemento può quindi essere inserito nella posizione finale corretta scambiandolo con `A[n]`. Se ora si toglie il nodo n dall'heap, diminuendo `A.heapSize`, si nota che i figli della radice restano max-heap, ma la nuova radice potrebbe violare la proprietà del max-heap. Questo problema può essere rimosso chiamando la procedura `maxHeapify( A, 1)`, che lascia un max-heap in `A[1..n-1]`. Questa operazione viene ripetuta fino ad un heap di dimensione 2, già ordinato per definizione.

Come detto in precedenza, la procedura `buildMaxHeap` impiega un tempo di esecuzione lineare ($O(n)$), mentre le $n - 1$ chiamate alla procedura `maxHeapify` impiegano ciascuna un tempo $O(log\, n)$. Pertanto il tempo di esecuzione dell'`heapSort` impiega un tempo di esecuzione $T(n) = O(n\, log\, n)$.

8.6 Quick Sort

Quick sort è un algoritmo di ordinamento divide et impera, il cui tempo di esecuzione nel caso peggiore è $O(n^2)$. Nonostante un tempo di esecuzione molto lento nel caso peggiore, quick sort è uno degli algoritmi più utilizzati perchè ha un tempo medio atteso $\Theta(n\,log\,)$ e i fattori costanti nascosti dalla notazione asintotica sono pressochè nulli. Inoltre, è un algoritmo di ordinamento sul posto, che lo rende utilizzabile anche in calcolatori con memoria limitata.

L'idea generale di questo algoritmo si basa sui tre tipici passi del metodo divide et impera, per un sottoarray `A[p..r]`:

1. Divide: partiziona l'array `A[p..r]` in due sottoarray `A[p..q-1]` e `A[q+1..r]` (eventualmente vuoti) tali che ogni elemento di `A[p..q-1]` sia minore o uguale di `A[q]` che, a sua volta, è minore o uguale di ogni elemento di `A[q+1..r]`.
2. Impera: ordina i due sottoarray `A[p..q-1]` e `A[q+1..r]` chiamando ricorsivamente sè stesso.
3. Combina: nessuna operazione di ricombinazione necessaria in quanto l'array `A[p..r]` è già ordinato.

La procedura `quickSort` è implementata tramite il seguente pseudocodice:

```
1 quickSort(A, p, r):
2   if p < r:
3     q = partition(A, p, r)
4     quickSort(A, p, q-1)
5     quicksort(A, q+1, p)
```

Per pter ordinare un intero array `A`, la chiamata iniziale a tale algoritmo è `quickSort(A, 1, A.length)`. Si noti che all'interno di tale algoritmo viene chiamata la sottoprocedura `partition`, definita come segue in pseudocodifica:

```
1 partition(A, p, r):
2   x := A[r]
3   i := p - 1
4   for j := p to r - 1:
5     if A[j] <= x:
6       i := i + 1
7       swap A[i] with A[j]
8   swap A[i + 1] with A[r]
9   return i + 1
```

Questo algoritmo riarrangia il sottoarray `A[p..r]` sul posto selezionando un elemento `x = A[r]` come pivot, intorno a cui partizionare l'array. All'inizio di ogni iterazione del ciclo ***for*** di riga 4, per qualsiasi indice k si individuano quattro regioni:

1. Se $p \leq k \leq i$, allora `A[k]` $\leq$ `x`;
2. Se $i+1 \leq k \leq j-1$, allora `A[k]` $\geq$ `x`;
3. Se $k = r$, allora `A[k]=x`;
4. Gli indici k tali che $j \leq k \leq r-1$ non hanno una particolare relazione con il pivot x.

Le ultime due righe della procdeura, inveceinseriscono il pivot al suo posto nel mezzo dell'array, scambiandolo con l'elemento più a sinistra, che è maggiore di x, e restituisce un nuovo indice di pivot. Il tempo di esecuzione dell'algoritmo `partition` con input il sottoarray `A[p..r]` è $\Theta(n)$, con $n = r-p+1$.

Il tempo di esecuzione dell'algoritmo `quickSort` dipende solamente da come viene partizionato l'array (in maniera bilanciata o meno) che, a sua volta, dipende da quali elementi vengono utilizzati per il partizionamento. Se il partizionamento è bilanciato, l'algoritmo ha un tempo di esecuzione $\Theta(n\,log\,n)$, mentre nel caso peggiore, quando il partizionamento è sbilanciato, l'algoritmo converge ad una solzuione in un tempo $\Theta(n^2)$.

Il comportamento nel caso peggiore si verifica quando la subroutine `partition` produce un sottoproblema con $n-1$ elementi e uno vuoto. Per calcolare il tempo di esecuzione, si suppone che questo

sbilanciamento si verifichi per ogni chiamata ricorsiva. Il partizionamento costa un tempo di esecuzione $\Theta(n)$ e, dato che uno dei due array è vuoto e làaltro conta $n-1$ elementi, si ha un tempo totale:

$$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(1) + \Theta(n)$$

Intuitivamente, e si sommano i costi ad ogni livello della ricorsione si ottiene una serie aritmetica, il cui valore è $\Theta(n^2)$. Questa situazione si verifica quando l'array di partenza è già completamente ordinato.

Il comportamento nel caso ottimo si verifica quando la subroutine `partition` produce due sottoproblemi di dimensione non maggiore di $n/2$: in questo caso il tempo di esecuzione dell'algoritmo è molto più rapido e avviene in un tempo totale $T(n) \leq 2T(n/2) + \Theta(n)$, che per il secondo caso del teorema dell'esperto, ha soluzione $T(n) = \Theta(n \log n)$. Si noti, inoltre, che nel caso in cui la partizione non fosse perfettamente bilanciata, l'algoritmo riuscirebbe comunque a riordinare l'array in un tempo $T(n) = \Theta(n \log n)$: questo dimostra che il `quickSort` è un algoritmo molto più vicino al caso ottimo che al caso pessimo, caso che si verifica in una sola istanza del problema (quando, appunto, è ordinato).

Il comportamento nel caso medio si verifica quando la subroutine `partition` produce una combinazione di partizioni 'buone' e 'cattive'. Si suppone, per semplicità, che le partizioni buone e cattive si alternino all'interno dell'albero di ricorsione e che quelle buone siano tutte nel caso migliore, mentre le quelle cattive siano nel caso pessimo. Si ipotizzi che nella radice dell'albero il costo di ripartizione è n e i sottoarray prodotti hanno dimensione $n-1$ e 0 (caso pessimo), mentre nel livello successivo il prtizionamento del sottoarray $n-1$ produca due array di dimensione $(n-1)/2$ e $(n-1)/2-1$ (caso migliore). Il costo di una divisione cattiva, seguito da una divisione buona è comunque $\Theta(n)$, ovvero lo stesso costo di una divisione buona: intuitivamente, quindi, la coppia divisione buona/cattiva impiega lo stesso tempo totale di esecuzione $\Theta(n \log n)$, con l'unica differenza che cambiano le costanti moltiplicative, eclissante nella notazione asintotica.

8.7 Counting Sort

Gli algoritmi analizzati fino ad ora, seppur differenti fra loro, condividono un'importante proprietà: l'ordinamento che effettuano è basato soltanto su confronti fra gli elementi di input. Questi algoritmi sono detti di ordinamento per confronti e, dati due elementi a_i e a_j, eseguono uno dei test $a_i < a_j, a_i \leq a_j, a_i = a_j, a_i \geq a_j$ o $a_i > a_j$ per determinare il loro ordine relativo.

Gli algoritmi di ordinamento per confronti possono essere visti in termini di alberi di decisione, ovvero alberi binari completi che rappresentano i confronti fra gli elementi effettuati da un determinato algoritmo di ordinamento. Ora, l'esecuzione di un algoritmo di ordinamento corrisponde ad indicare su tale albero un cammino semplice che collega la radice dell'albero con una foglia (nello specifico, una delle foglie che rappresentano le permutazioni ordinate dell'array di ingresso). Ogni nodo interno di tale albero rappresenta un confronto: il sottoalbero sinistro corrisponde a confronti del tipo $a_i \leq a_j$, mentre quello destro corrisponde a confronti del tipo $a_i > a_j$. Si noti che sulle foglie sono presenti tutte le possibili permutazioni della sequenza di input: dunque un albero di decisione può presentare più di $n!$ foglie, in quanto alcune permutazioni potrebbero comparire più volte, ma meno di 2^h (con h, l'altezza dell'albero). La lunghezza del cammino semplice più lungo dalla radice di un albero di decisione ad una delle sue foglie rappresenta il numero di confronti che un determinato algoritmo di ordinamento deve svolgere nel caso peggiore: questo numero è equivalente all'altezza dell'albero stesso. Si introduce quindi il seguente teorema, che determina un limite inferiore sul tempo di esecuzione degli algoritmi di ordinamento per confronti:

Teorema 8.7.1. *Qualsiasi algoritmo di ordinamento per confronti richiede $\Omega(n \log n)$ confronti nel caso peggiore.*

da cui deriva anche:

Teorema 8.7.2. *Ogni albero di decisione di un algoritmo di ordinamento di n elementi ha altezza $\Omega(n \log n)$.*

Una volta determinato il limite inferiore del tempo di esecuzione degli algoritmi di ordinamento, si introduce qui l'algoritmo **counting sort**, che riordina un array di dimensione n in un tempo lineare $\Theta(n)$. Tale algoritmo suppone che ciascuno degli elementi di input sia un numero intero compreso nell'intervallo

da 0 a $k \in \mathbb{N}$ e determina per ciascun di essi il numero di elementi minori; utilizza poi questa informazione per inserire l'elemento corrente direttamente nella giusta posizione nell'array di output. Ad esempio, se l'elemento x è più grande di 5 altri elementi, allora verrà inserito nella posizione 6 dell'array di output.

Nel codice di counting sort, si suppone che l'input sia un array `A[1..n]`, con n=`A.length`. Occorrono altri due array: l'array `B[1..n]`, che contiene l'output ordinato, e l'array `C[0..k]` fornisce la memoria temporanea di lavoro. In pseudocodice:

```
1  countingSort(A, B, k):
2    C := [0 .. k]
3    for i := 0 to k:
4      C[i] := 0
5    for j := 1 to A.length:
6      C[A[j]] := C[A[ji]] + 1
7    for i:= 1 to k:
8      C[i] := C[i] C[i - 1]
9    for j := A.length down to 1
10     B[C[A[j]]] := A[j]
11     C[A[j]] := C[A[j]] - 1
```

Dopo che il ciclo ***for*** in riga 3 inizializza a zero tutti gli elementi dell'array `C`, ogni elemento dell'input viene esaminato con il secondo ciclo ***for*** (in riga 5): se il valore di un elemento è i, viene incrementato il valore di `C[i]`. Dunque, dopo la riga 6, l'array `C` contiene il numero di elementi uguagli ad i, per ogni $i = 0, 1, ..., k$. Le righe 7 e 8 determinano, per ogni $i = 0, 1, ..., k$, quanti elementi di input sono minori o uguali a i. Infine, il ciclo ***for*** di riga 9 inserisce l'elemento corrente nella posizione corretta all'interno dell'array di output `B`.

Il primo ciclo ***for*** impiega un tempo di esecuzione $\Theta(k)$, il secondo un tempo $\Theta(n)$, il terzo un tempo $\Theta(k)$ e, infine, il ciclo ***for*** di riga 9 impiega un tempo di esecuzione lineare $\Theta(n)$. Dunque, il tempo di esecuzione totale dell'algoritmo `countingSort` è $T(n) = \Theta(n + k)$. Di solito, questa procedura viene utilizzata quando $h = \Theta(n)$, facendo si che il tempo totale di esecuzione sia un $\Theta(n)$.

L'algoritmo appena analizzato batte il limite inferiore di tempo $\Omega(n \log n)$ per gli algoritmi di ordinamento per confronti perchè non confronta nessun elemento dell'input, ma utilizza il valore di ogni elemento come indice di un array `C` di appoggio.

Capitolo 9

Strutture Dati

Gli insiemi manipolati dagli algoritmi, a differenza di quelli matematici, possono essere modificati inserendo o rimuovendo elementi. Questi insiemi sono detti dinamici e giocano un ruolo importante in informatica, perchè modellano le strutture utilizzate per memorizzare in modo ordinato i dati.

In una tipica implementazione di un insieme dinamico, ogni elemento è rappresentato da un oggetto, i cui attributi possono essere esaminati e manipolati a piacimento dagli algoritmi. In molte strutture dati, l'oggetto dispone di una chiave identificativa (spesso univoca, ma non necessariamente) e ovviamente di dati satelliti che si vogliono memorizzare ordinatamente in memoria. Oltre a questi due attributi, l'oggetto può anche contenere altri dati specifici per una determinata struttura dati, in modo da rendere più semplice e veloce la loro manipolazione.

Le tipiche operazioni che si possono svolgere sulle strutture dati sono suddivise in due categorie: le query (interrogazioni), che hanno il solo scopo di estrapolare informazioni dall'insieme dinamico, e le operazioni di modifica, che hanno il compito di modificare l'insieme. Di seguito sono elencate le istruzioni più comuni:

- `search(S, k)`: è un'operazione di query che, dato un insieme S e un valore chiave k, restituisce `NIL` se tale elemento non appartiene all'insieme.
- `insert(S, x)`: è un'operazione di modifica che inserisce all'interno dell'insieme S l'elemento puntato da x.
- `delete(S, x)`: è un'operazione di modifica che, dato un puntatore x ad un elemento dell'insieme S, rimuove x da S.
- `minimum(S)`: è un'operazione di query che ritorna l'elemento dell'insieme S con la chiave più piccola.
- `maximum(S)`: è un'operazione di query che ritorna l'elemento dell'insieme S con la chiave più grande.
- `successor(S, x)`: è un'operazione di query che, dato un elemento x la cui chiave appartiene ad un insieme totalmente ordinato S, restituisce un puntatore all'elemento successivo più grande di S oppure `NIL` se x è il più grande degli elementi.
- `predecessor(S, x)`: è un'operazione di query che, dato un elemento x la cui chiave appartiene ad un insieme totalmente ordinato S, restituisce un puntatore all'elemento precedente più piccolo di S oppure `NIL` se x è il più piccolo degli elementi.

9.1 Stack

Gli **stack** sono insiemi dinamici dove l'elemento da rimuovere tramite l'operazione `delete` è predeterminato. In questa struttura dati, l'elemento cancellato è quello inserito per ultimo, secondo la politica LIFO (Last In, First Out). Nello specifico, le operazioni di `insert` e `delete` prendono rispettivamente il nome di `push` e `pop` [1]: la prima inserisce in cima alla pila l'elemento passato come argomento, mentre la seconda operazione elimina l'unico elemento accessibile dalla pila, ovvero la cima.

[1] Questa operazione non prende nessun argomento, in quanto l'elemento da eliminare è predeterminato

Questa struttura dati può essere implementata tramite un array con un massimo di n elementi `S[1..n]`, che presenta lo specifico atrtibuto `S.top`, ovvero l'indice tramite cui accedere all'ultimo elemento inserito. L'array è dunque composto dagli elementi `S[1..S.top]`, dove `S[1]` rappresenta l'elemento in fondo alla pila, mentre `S[S.top]` rappresenta l'elemento in cima. Ovviamente, se `S.top` $= 0$, si dice che la pila è vuota, in quanto non contiene nessun elemento. In questo caso, se si tenta di estrarre un elemento dallo stack, si ottiene un errore di underflow dello stack, mentre se si cerca di inserire un elemento sulla pila piena (che conta quindi di n elementi), si ottiene un errore di overflow dello stack.

Le operazioni dello stack possono essere implementate molto semplicemente in pseudocodifica come segue:

```
1 push(S, x):
2   if S.top = S.length:
3     error "overflow"
4   else:
5     S.top := S.top + 1
6     S[S.top] := x
```

```
1 pop(S):
2   if iS.top = 0:
3     error "underflow"
4   else:
5     S.top := S.top - 1
6     return S[S.top + 1]
```

Si noti come, in questo caso, l'operazione di `pop` ritorna l'elemento appena eliminato dallo stack. Entrambe le procedure vengono eseguite in tempo costante $\Theta(1)$.

9.2 Queue

Le **code** sono insiemi dinamici dove l'elemento da rimuovere tramite l'operazione `delete` è predeterminato. In questa struttura dati, l'elemento cancellato è quello inserito per primo, secondo la politica FIFO (First In First Out). Nello specifico, la coda presenta un inizio detto `head` e una fine detta `tail`, e le operazioni di `insert` e `delete` prendono rispettivamente il nome di `enqueue` e `dequeue` [2]: la prima inserisce in fondo alla fila l'elemento passato come argomento, mentre la seconda operazione elimina il primo elemento della fila.

Questa struttura dati può essere implementata tramite un array di n elementi `Q[1..n]`, che contiene un massimo di $n - 1$ elementi, per ragioni che verranno chiarite in seguito. L'attributo `Q.head` punta all'inizio della coda, mentre l'attributo `Q.tail` punta alla posizione in cui l'ultimo elemento che dovrà essere inserito prenderà posto (ovvero alla posiziove vuota successiva all'ultimo elemento della coda). Gli elementi della coda, quindi, occupano le posizioni `Q.head`, `Q.head + 1`, ..., `Q.tail - 1`. Alla fine dell'array la posizione 1 della queue segue immediatamete la posizione n secondo un ordine circolare. Se `Q.head = Q.tail` allora la coda è vuota. All'inizio le posizioni `Q.head` e `Q.tail` combaciano e sono entrambe inizializzate al valore 1.

Come per gli stack, se la coda è vuota, il tentativo di rimuovere un elemento provoca un errore di underflow, mentre se `Q.head = Q.tail + 1` la coda è piena e il tentativo di inserire un nuovo elemento provoca un errore di overflow.

Le operazioni della queue possono essere implementate molto semplicemente in pseudocodifica come segue:

```
1 enqueue(Q, x):
2   Q[Q.tail] := x
3   if Q.tail = Q.length:
4     Q.tail := 1
5   else:
6     Q.tail := Q.tail + 1
```

[2] Anche in questo caso, questa operazione non prende nessun argomento, in quanto l'elemento da eliminare è predeterminato

```
dequeue(Q, x):
  x := Q[Q.head]
  if Q.head = Q.length:
    Q.head := 1
  else:
    Q.head := Q.head + 1
  return x
```

Entrambe le procedure vengono eseguite in un tempo costante $\Theta(1)$.

9.3 Linked List

Una **lista concatenata** è una struttura dati i cui oggetti sono disposti in ordine lineare, determinato da un puntatore in ogni oggetto. Una lista doppiamente concatenata è una lista in cui ogni oggetto presenta, oltre ad una chiave `key`, anche un puntatore all'elemento successivo `next` e un puntatore a quello precedente `prev`. Se `x.prev = NIL`, allora l'elemento x è il primo elemento della lista e si dice essere la testa (o head) della lista. Se, invece, `x.next = NIL`, allora x è l'ultimo elemento della lista e si dice essere la coda (o tail) della lista. Intuitivamente, l'attributo `L.head` punta alla testa della lista, che sarà vuota se `L.head = NIL`.

Questa struttura dati può presentare varie forme: può essere doppiamente concatenata o singolarmente concatenata, oppure può essere circolare o non. Una lista si dice singolarmente concatenata se i suoi oggetti non sono dotati di puntatore all'elemento precedente, mentre si dicono circolari se l'ultimo elemento possiede un puntatore alla testa della lista, che a sua volta possiede un puntatore alla coda se la lista è circolare. Una lista concatenata può anche essere ordinata o non: si dice ordinata quando la disposizione lineare degli elementi corrisponde con la disposizione crescente delle chiavi degli elementi e, in tal caso, la testa della lista conterrà l'elemento minimo, mentre la coda l'elemento massimo.

Nel seguito si fa riferimento a liste non ordinate doppiamente concatenate per lo sviluppo degli algoritmi che le manipolano.

Ricerca La prima procedura che sia analizza è `listSearch(L, k)`, che trova il primo elemento con la chiave k nella lista L, restituendo un puntatore a tale oggetto. Se nessun oggetto con chiave k è presente nella lista, allora viene restituito il valore `NIL`. In pseudocodifica:

```
listSearch(L, k):
  x := L.head
  while x != NIL and x.key != key:
    x := x.next
  return x
```

Si noti quindi che l'algoritmo `listSearch` cerca l'elemento di chiave k tramite una ricerca lineare sulla list L di n elementi. Dunque, l'algoritmo impiega un tempo $\Theta(n)$ nel caso peggiore, in quanto potrebbe essere necessario scorrere l'intera lista.

Inserimento La seconda procedura analizzata è `listInsert(L, x)` che inserisce l'elemento x di attributo key (già inizializzato) in testa alla lista. In pseudocodifica:

```
listInsert(L, x):
  x.next := L.head
  if L.head != NIL:
    L.head.prev := x
  L.head := x
  x.prev := NIL
```

Questa procedura impiega un tempo costante $\Theta(1)$ per la sua esecuzione.

Rimozione L'ultima procedura analizzata per le linked list è `listDelete(L, x)`, che rimuove l'elemento x dalla lista L. Per poter eliminare tale elemento è prima necessario chiamare la funzione `listSearch` per ottenere il puntatore all'elemento desiderato. In pseudocodifica:

```
listDelete(L, x):
  if x.prev != NIL:
    x.prev.next := x.next
  else:
    L.head := x.next
  if x.next != NIL:
    x.next.prev := x.prev
```

Anche questa procedura impiega un tempo di esecuzione $\Theta(1)$.

9.4 Hash Table

9.4.1 Indirizzamento diretto

Prima di procedere con l'introduzione alle tavole di hash, è prima necessario introdurre il concetto di indirizzamento diretto, una tecnica molto efficiente nel caso in cui l'insieme da cui vengono acquisite le chiavi, detto insieme universo $U = \{0, 1, 2..., m-1\}$, è un insieme ragionevolmente piccolo. Si suppone, inoltre, che due elementi distinti non possano avere chiavi coincidenti. Per rappresentare un tale insieme dinamico si utilizza un array, oppure una tavola ad indirizzamento diretto, indicata con $T[0..m-1]$, dove ogni cella k di tale tabella punta all'elemento dell'insieme di chiave k. Se la k-esima cella non contiene nessun elemento, viene inizializzata con il valore `NIL`.

Le operazioni di dizionario sono semplici da implementare in pseudocodifica e impiegano tutte un tempo costante $O(1)$ (nel caso peggiore):

```
directAddressSearch(T, k):
  return T[k]
```

```
directAddressInsert(T, x):
  T[x.key] := x
```

```
directAddressDelete(T, x):
  T[x.key] := NIL
```

In alcune implementazioni è possibile memorizzare l'elemento di chiave k direttamente all'interno della tabella, anzichè in un oggetto esterno, risparmiando spazio in memoria.

9.4.2 Introduzione alle Tavole di Hash

La difficoltà nell'implementazione di una tale struttura dati è evidente: l'insieme universo U, nella maggior parte dei casi, è troppo grande per essere memorizzato in una tavola T di dimensione $|U|$. Inoltre, l'insieme K delle chiavi effettivamente memorizzate è molto più piccolo dell'inisieme U delle chiavi disponibili e, dunque, la maggior parte dello spazio allocato per la tavola T non verrebbe mai utilizzato. A questo proposito si introduce una nuova struttura dati, detta **tavola di hash**, che utilizza una memoria proporzionale al numero delle chiavi effettivamente memorizzate nel dizionario, riducendo lo spreco di memoria.

Quando l'insieme K delle chiavi memorizzate in un dizionario è molto più piccolo dell'universo U di tutte le chiavi possibili, utilizzando una tavola di hash si può ridurre lo spazio richiesto fino a $\Theta(|K|)$, rdiucendo però l'efficienza temporale a $O(1)$ nel caso medio (anzichè pessimo). Nella tabella di hash, l'elemento di chiave k non viene memorizzato direttamente nella cella k, ma viene utilizzata una cosiddetta funzione di hash $h(k)$ che calcola l'indice k della cella. La funzione di hash $h(k)$ associa ad ogni chiave dell'universo U una specifica chiave della tavola di hash $T[0..m-1]$. Formalmente:

$h(k) : U \to \{0, 1, ..., m-1\}, \quad m << |U|$

Si dice che $h(k)$ è il valore hash della chiave k.

9.4.3 Hashing Concatenato

Il problema principale con la tecnica di indirizzamento appena analizzata è che, riducendo l'intervallo degli indici da $|U|$ ad $m << |U|$, è molto probabile che due chiavi vengano mappate nella stessa cella: in tal caso si dice che avviene una collisione. Per evitare un evento simile è possibile, in prima analisi, implementare una funzione di hash totalmente deterministica il più randomica possibile, in modo da minimizzare le collisioni. Si dimostra però che un evento di collisione è impossibile da evitare in quanto $|U| > m$ e quindi, dopo l'm-esima chiamata alla funzione di hash, avverà sicuramente una collisione.

Si rende necessario, dunque, implementare un meccanismo che gestisca tali eventi. Nello specifico, la tecnica più utilizzata è il concatenamento (o chaining), tramite cui, tutti gli elementi associati ad una stessa cella k sono posti in una lista concatenata. La cella k, in questo caso, punta al nodo di testa della lista che contiene gli elementi mappati in tale cella, oppure ha valore `NIL`, nel caso in cui la cella non contenga nessun elemento.

Le operazioni di dizionario su una tavola di hash T sono facili da implementare in pseudocodifica nel caso di gestione delle collisioni tramite concatenamento:

```
1 chainedHashInsert(T, x):
2   listInsert(T[h(x.key)], x)
```

```
1 chainedHashSearch(T, x):
2   listSearch(T[h(x.key)], x)
```

```
1 chainedHashDelete(T, x):
2   listDelete(T[h(x.key)])
```

In cui le procedure `listInsert`, `listSearch` e `listDelete`, sono le stesse analizzate nella sezione delle liste concatenate e hanno il compito, rispettivamente, di inserire in testa alla lista un nodo, cercare un nodo nella lista ed eliminare un nodo dalla lista.

Si passa ora all'analisi del tempo di esecuzione di tali procedure: nel caso peggiore, l'inserimento in lista di un nodo è $O(1)$, la ricerca avviene in tempo proporzionale alla lunghezza della lista, quindi $O(n)$, mentre l'eliminazione di un nodo dalla lista avviene, sempre nel caso peggiore, in tempo $O(1)$ se la lista è doppiamente concatenata.

Si noti che la funzione `chainedHashDelete` prende come input un elemento x, non la sua chiave k, quindi non occorre cercare prima l'elemento x. Se la tavola di hash supporta la cancellazione, allora le sue liste dovrebbero essere doppiamente concatenate in modo che la cancellazione di un elemento sia più rapida. Se le liste fossero singolarmente concatenate, per cancellare l'elemento x, si dovrebbe prima trovare x nella lista $T[h(x.key)]$ in modo da poter aggiornare l'attributo `next` dell'elemento precedente in lista, assegnandogli il valore `NIL`.

9.4.4 Analisi della Funzione di Hash

Data una tavola di hash T, che conta m celle in cui sono memorizzati n elementi, si definisce il fattore di carico α della tavola T come il rapporto n/m, ossia il numero medio di elementi memorizzati in una lista. Il caso peggiore nell'hashing si verifica quando tutte le n chiavi sono associate alla stessa cella, creando una lista di lunghezza n. Il tempo di esecuzione della ricerca diventa quindi $\Theta(n)$ a cui si aggiunge il tempo di esecuzione della funzione di hashing. Ovviamente, un caso del genere è molto improbabile nel caso in cui la funzione di hash sia ben progettata. Per il momento si suppone che ogni elemento ha uguale probabilità di essere mappato in una qualsiasi delle m celle, indipendentemente dalle celle in cui sono stati mappati gli altri elementi. Tale ipotesi viene definita hashing uniforme semplice. Per ogni $j = 0, 1, ..., m-1$, si indica con n_j la lunghezza della lista $T[j]$, ottenendo quindi il numero di elementi totali memorizzati in tabella è $n = n_0 + n_1 + ... + n_{m-1}$. Il valore atteso di ogni n_j sarà $E[n_j] = \alpha = n/m$, quindi il tempo medio per la ricerca di un elemento di chiave k non presente nella lista (caso pessimo) è $\Theta(1 + \alpha)$, che si dimostra essere anche il tempo di ricerca dello stesso elemento, questa volta presente tabella. Nella pratica, se il numero di celle nella tavola di hash è almeno proporzionale al numero di elementi della tavola, si ottiene che $n = O(m)$ e quindi $\alpha = n/m = O(m)/m = O(1)$. Pertanto, la ricerca di un elemento della tavola richiede un tempo costante.Ogni operazione di dizionario può essere svolta, in media, in un tempo $O(1)$.

9.4.5 Funzione di Hash

Per progettare una buona tabella di hash è necessario implementare una funzione di hash che sia altamente efficiente. A questo proposito si introducono tre possibili schemi di implementazione: hashing per divisione (uristico), hashing per moltiplicazione (euristico) e hashing universale (aleatorio, non analizzato in questa sezione). In generale, una buona funzione di hashing deve soddisfare approssimativamente la condizione di hashing uniforme semplice: ogni chiave deve avere la stessa probabilità di essere mappata in una qualsiasi cella della tabella. Di solito non è possibile verificare questa condizione, in quanto non è nota la distribuzione delle probabilità secondo cui vengono estratte le chiavi. Quindi, nella pratica, spesso si utilizzano delle tecniche euristiche per la realizzazione di tali funzioni.

La maggior parte delle funzioni di hashing suppone che l'universo delle chiavi sia l'insieme dei numeri naturali $\mathbb{N}$ e quindi, se nella struttura dati progettata, la chiave non è un numero naturale ma, ad esempio, una stringa, è necessario studiare un metodo di conversione: nei calcolatori questo metodo è tipicamente già implementato, in quanto ogni informazione analogica viene convertita in una stringa binaria di bit (appartenente ad $\mathbb{N}$). Nello studio dei tre metodi di hashing si suppone che le chiavi siano numeri naturali.

Metodo della Divisione Quando si applica il metodo della divisione per creare una funzione di hash, una chiave k viene associata a una delle m celle prendeno il resto della divisione fra k ed m. Formalmente, la funzione di hash è così definita:

$h(k) = k \ mod \ m$

Il vantaggio principale di questo metodo è che si può implementare molto rapidamente e richiede un tempo di esecuzione costante.

Quando si utilizza il metodo della divisione, si cerca di evitare alcuni valori di m. Nello specifico si evitano le potenze di 2, in quanto se $m = 2^p$, allora $h(k)$ rappresenta proprio i p bit meno significativi di k: infatti, sarebbe più corretto far dipendere la funzione di hash da tutti i bit della chiave. Inoltre, si evita di scegliere $m + 2^p - 1$ quando k è una stringa di caratteri interpretata in base 2^p, in quanto la permutazione dei caratteri di k non cambia il suo valore di hash. Una buona scelta di m, invece, potrebbe essere un numero primo non troppo vicino a una potenza esatta di 2.

Metodo della Moltiplicazione Il metodo della moltiplicazione per la creazione di funzioni di hash consiste in due passi. Nel primo passaggio si moltiplica la chiave k per una costante A, tale che $0 < A < 1$, per poi estrarre la parte frazionaria del numero appena ottenuto. Nel secondo passaggio si moltiplica questo valore per m e si prende la parte intera inferiore del risultato. Formalmente, la funzione di hash è così definita:

$h(k) = \lfloor m \, (k \cdot A \ mod \ 1) \rfloor$

dove $k \cdot A \ mod \ 1$ rappresenta la parte frazionaria di kA, ovvero $kA - \lfloor kA \rfloor$.

Il vantaggio principale di questo metodo è che il valore di m non è critico. Tipicamente si sceglie un valore di m tale per cui sia una potenza di 2, in modo da rendere più semplice implementare tale funzione in un calcolatore reale. Si upponga, infatti, che la dimensione di una parola nel calcolatore sia w bit e che il numero k sia contenuto in una sola parola. Si scelga poi un valore di A che sia una frazione nella forma $s/2^w$, con s intero nell'intervallo $0 < s < 2^w$. A questo punto, si moltiplica k per $s = A \cdot 2^w$: il risultato sarà un numero di $2w$ bit $r_1 2^w + r_0$, in cui r_1 rappresenta la parte più significativa del prodotto ed r_0 la parte meno significativa. Il valore hash desiderato di p bit è formato dai p bit più significativi di r_0. Sebbene queste operazioni funzionino con qualsiasi valore della costante A, la scelta spesso adoperata è $A \approx (\sqrt{5} - 1)/2 \approx 0.6180339887...$

9.4.6 Indirizzamento Aperto

Un altro metodo per evitare le collisioni è tramite l'indirizzamento aperto, in cui tutti gli elementi sono memorizzati nella tavola hash, ovvero ogni cella contiene un elemento dell'insieme dinamico o la costante `NIL` se non contiene nessun elemento. Quando si cerca un elemento, si esamina sistematicamente la tabella fino a quando non si trova l'elemento desiderato, oppure finchè non ci sono più elementi da controllare (l'elemento non è nell'array). Nell'indirizzamento aperto, a differenza degli altri metodi, la tavola hash può riempirsi fino a quando non può più fisicamente contenere altri elementi. Una conseguenza di questo design è che il fattore di carico α non supera mai il valore 1. Il vantaggio di questo metodo sta nel fatto che elimina completamente l'utilizzo dei puntatori, in quanto calcola la sequenza delle celle da esaminare, e libera quindi una notevole quantità di memoria, utilizzata per incrementare la capacità della tabella, riducendo il rischio di collisioni.

Inserimento Per effettuare un insierimento mediante il metodo dell'indirizzamento aperto, si esamina in successione le posizioni della tavola hash fino a che non si trova una cella libera in cui inserire la chiave. L'efficienza di questo metodo consiste nel calcolare una nuova sequenza di accesso alla tabella in base alla chiave dell'oggetto da inserire, anzichè seguire sempre lo stesso cammino di accessi, che impiegherebbe un tempo di esecuzione $\Theta(n)$. Per determinare quali celle esaminare durante la fase di ispezione, si estende la funzione di hash in modo da includere l'ordine di ispezione (a partire da 0), come secondo input. Formalmente, la funzione di hash modificata è definita come segue:

$h(k,i) : U \times \{0,1,...,m-1\} \rightarrow \{0,1,...,m-1\}$

Si richiede, inoltre, che per ogni chiave k la sequenza di ispezione $< h(k,0), h(k,1), ..., h(k,m-1) >$ sia una permutazione della sequenza $< 0,1,...,m-1 >$, in modo tale che ogni cella della tavola possa essere considerata come possibile cella in cui inserire una nuova chiave. In pseudocodifica:

```
1  hashInsert(T, k):
2    i := 0
3    repeat
4      j := h(k, i)
5      if T[j] = NIL or T[j] = DELETED:
6        T[j] := k
7        return j
8      else:
9        i := i + 1
10   until i = m
11   error "hash table overflow"
```

Questa procedura prende in input una tavola di hash T e una chiave k da inserire in tabella, e ritorna l'indice della cella in cui è stato inserito l'elemento, oppure ***error*** di overflow se non è presente nessuna cella libera. Si noti nella condizione in riga 5, che viene verificato se la cella $T[k]$ contiene il valore `DELETED`: questo valore speciale verrà ripreso e analizzato più avanti assieme alla procedura di eliminazione delle chiavi dalla tabella.

Ricerca Per la ricerca di una determinata chiave k esamina la stessa sequenza di celle che ha esaminato l'algoritmo di inserimento quando ha inserito l'elemento di chiave k. Dunque, la procedura di ricerca potrebbe terminare la propria esecuzione senza successo quando trova una cella vuota, in quanto la chiave k sarebbe stata inserita in quella posizione (si presuppone che le chiavi non possano essere eliminate dalla tavola). In pseudocodifica:

```
1  hashSearch(T, k):
2    i := 0
3    repeat
4      j := h(k, i)
5      if T[j] = k:
6        return j
7      i := i + 1
8    until T[j] = NIL or i = m
9    return NIL
```

Questa procedura, come la precedente, prende in input una tavola di hash T e una chiave k da ricercare in tabella, e ritorna l'indice della cella in cui è stato trovato l'elemento, oppure `NIL` se non è presente nessuna cella che contiene l'elemento oppure se nella sequenza di ricerca si trova una cella libera (per le ragioni precedentemente analizzate).

Rimozione La procedura di cancellazione di una chiave dalla tavola di hash ad indirizzamento aperto è un'operazione molto complessa, in quanto non è possibile semplicemente cancellare il contenuto della cella i, assegnandone il valore `NIL`, in quanto sarebbe impossibile ricercare qualsiasi elemento in tabella per come è stata definita la procedura di ricerca. Dunque, si rende necessario marcare la cella il cui contenuto è stato eliminato con il valore speciale `DELETED`, anzichè `NIL`. È per questo motivo che in riga 5 della procedura di inserimento viene anche verificato se la cella ispezionata contanga il valore `DELETED`. Si noti inoltre che, utilizzando questa nuova notazione, il tempo di esecuzione della procedura

di ricerca non dipende più dal fattore di carico α. Per questo motivo, nella pratica si preferisce spesso utilizzare il metodo della concatenazione quando è necessario che la tabella di hash supporti l'operazione di cancellazione delle chiavi.

9.4.7 Hashing Uniforme

Nell'analisi delle tabelle hash con indirizzamento aperto si ipotizza hashing uniforme: si suppone, infatti, che ogni chiave abbia la stessa probabilità di avere come sequenza di ispezione una delle $m!$ permutazioni di $< 0, 1, ..., m-1 >$. L'hashing uniforme estende il concetto di hashing uniforme semplice, impiegato più volte precedentemente, al caso in cui la funzione di hash produce, non un singolo numero, ma un'intera sequenza di ispezione. Nella pratica non è possibile ottenere una funzione di hash uniforme, ma si utilizzano delle approssimazioni accettabili. Si esaminano nel seguito tre tecniche utilizzate per calcolare le sequenze di ispezione richieste nell'indirizzamento aperto: ispezione lineare, ispezione quadratica e doppio hashing. Tali tecniche grantiscono che la sequenza $< h(k,0), h(k,1), ..., h(k,m-1) >$ sia una permutazione di $< 0, 1, ..., m-1 >$ per ogni chiave k, ma nessuna di loro può garantire l'ipotesi di hashing uniforme, in quanto nessuna di esse è in grado di generare più di m^2 sequenze di ispezioni differenti (invece delle $m!$ sequenze richieste).

Ispezione Lineare Data una funzione di hash ordinaria $h' : U \to \{0, 1, ..., m-1\}$, detta funzione di hash ausiliaria, il metodo di ispezione lineare utilizza la funzione di hash

$h(k,i) = (h'(k) + i) \ mod \ m$

per $i = 0, 1, ..., m-1$. Data la chiave k, la prima cella esaminata è $T[h'(k)]$, che è la cella data dalla funzione di hash ausiliaria, la seconda cella è $T[h'(k)+1]$ e così via fino alla cella $T[m-1]$. Poi, l'ispezione riprende dalle celle $T[0], T[1], ..., T[h'(k)-1]$. Poichè la prima cella ispezionata determina l'intera sequenza di ispezioni, ci sono soltanto m sequenze di ispezione distinte. Questa tecnica è facile da implementare ma presenta un problema noto come addensamento primario: si formano lunghe file di celle occupate che aumentano il tempo medio di ricerca. Tale fenomeno si presenta perchè una cella vuota preceduta da i celle piene ha probabilità $(i+1)/m$ di essere la prossima ad essere occupata e le lunghe file di celle occupate tendono, dunque, a diventare sempre più lunghe.

Ispezione Quadratica Data la funzione di hash ausiliaria $h' : U \to \{0, 1, ..., m-1\}$, il metodo di ispezione qudratica utilizza la funzione di hash

$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \ mod \ m$

per $i = 0, 1, ..., m-1$, con $c_1, c_2 \neq 0$ costanti ausiliarie. Data la chiave k, la prima cella esaminata è $T[h'(k)]$, mentre le successive posizioni esaminate sono distanti dalle precedenti di quantità che dipendono in modo quadratico dal numero d'ordine di ispezione i. Questa tecnica funziona meglio della precedente, ma i valori c_1, c_2 ed m non possono essere scelti in maniera arbitraria, ma devono essere tali per cui si possa percorrere l'intera tabella. Inolte, se due chiavi hanno la stessa posizione iniziale di ispezione, allora le due sequenze di ispezione saranno identiche portando al cosiddetto addensamento secondario.

Doppio Hashing Date la funzioni di hash ausiliarie h_1 ed h_2, il metodo di ispezione lineare utilizza la funzione di hash

$h(k,1) = (h_1(k) + i h_2(k)) \ mod \ m$

per $i = 0, 1, ..., m-1$. Il doppio hashing è il metodo migliore disponibile per l'indirizzamento aperto, in quanto le permutazioni prodotte hanno molte caratteristiche comuni con le permutazioni casuali. Data la chiave k, la prima cella esaminata è $T[h_1(k)]$, mentre le successive posizioni sono distanziate dalle precedenti di quantità $h_2(k) \ mod \ m$. Differentemente dai precedenti, il metodo del doppio hashing produce sequenze che dipendono in due modi dalla chiave k, in quanto possono variare sia la posizione iniziale della sequenza di ispezione, sia la distanza fra due posizioni successive.

Inoltre, il valore $h_2(k)$ deve essere relativamente primo con la dimensione m della tavola hash perchè venga ispezionata l'intera tabella. Un modo pratico per garantire tale condizione è scegliere m potenza di due e definire h_2 in modo che produca sempre un numero dispari. Un altro modo è scegliere m primo e definire h_2 in modo che generi sempre un numero intero positivo minore di m. In questo contesto, il doppio hashing è migliore delle precedenti tecniche in quanto utilizza $\Theta(m^2)$ sequenze di ispezione, anzichè $\Theta(m)$, perchè ogni possibile coppia di h_1, h_2 produce una distinta sequenza di ispezione.

9.5 Alberi Binari

Un albero binario di ricerca è una struttura dati ad albero[3] che supporta molte operazioni degli insiemi dinamici, come `search`. `delete`, `insert`, `minimum`, `maximum`, `successor` e `predecessor`. Tali alberi sono rappresentati da strutture concatenate in cui ogni nodo è un oggetto, che oltre ad avere una chiave e i dati satellite, contengono anche gli attributi `left`, `right` e `parent`, che puntano rispettivamente al figlio sinistro, al figlio destro e al padre. Se manca un figlio o un padre, i corrispondenti attributi sono inizializzati a `NIL`. Il nodo radice, detto anche root, è l'unico nodo nell'albero il cui attributo padre è inizializzato con `NIL`.

Le chiavi dell'albero binario di ricerca sono memorizzate in maniera da rispettare sempre la proprietà fondamentale per cui se x è un nodo in un albero binario di ricerca ed y è il nodo nel sottoalbero sinistro di x, allora $y.key \leq x.key$, mentre se y è un nodo nel sottoalbero destro di x, allora $y.key \geq x.key$. Tale proprietà permette di elencare ordinatamente tutte le chiavi di un albero binario di ricerca con un semplice algoritmo ricorsivo di attraversamento simmetrico di un albero, tramite cui la chiave della radice di un sottoalbero viene stampata nel mezzo tra la stampa dei valori nel sottoalbero sinistro e la stampa dei valori nel sottoalbero destro. In pseudocodice:

```
1 inorderTreeWalk(x):
2   if x != NIL:
3     inorderTreeWalk(x.left)
4     print x.key
5     inorderTreeWalk(x.right)
```

Per attraversare un albero binario di ricerca costituito da n nodi, è necessario un tempo $\Theta(n)$, in quanto, dopo la chiamata iniziale, la procedura viene chiamata ricorsivamente esattamente due volte per ogni nodo dell'albero.

Ricerca Un'altra tipica operazione svolta su un albero binario di ricerca è quella di cercare una chiave memorizzata nell'albero: oltre all'operazione `search`, gli alberi binari supportano anche query come `minimum`, `maximum`, `successor` e `predecessor`.

Dato un puntatore alla radice dell'albero e una chiave k da cercare, la procedura `treeSearch` restituisce il puntatore al nodo di chiave k, se esiste, oppure `NIL`, se la chiave non è presente nell'albero. In pseudocodifica:

```
1 treeSearch(x, k):
2   if x = NIL or k = x.key:
3     return x
4   if k < x.key:
5     return treeSearch(x.left, k)
6   else:
7     return treeSearch(x.right, k)
```

Questo algoritmo inizia la sua ricerca dalla radice dell'albero e ad ogni iterazione confronta il valore k passato come argomento con $x.key$. Se le due chiavi sono uguali la ricerca termina e viene restituito il puntatore al nodo, mentre se la chiave è minore della chiave corrente analizzata, per la proprietà fondamentale degli alberi binari, si chiama ricorsivamente la procedura sul sottoalbero sinistro, altrimenti sul sottoalbero destro. Il tempo di esecuzione di tale algoritmo è $O(h)$, dove h rappresenta l'altezza dell'albero.

Minimo e Massimo Un elemento con chiave minima in un albero binario di ricerca può sempre essere trovato seguendo, a partire dalla radice, i puntatori `left`, fino a quando non viene incontrato un valore `NIL`, sempre per la proprietà fondamentale degli alberi binari di ricerca. In pseudocodifica:

```
1 treeMinimum(x):
2   while x.left != NIL:
3     x := x.left
4   return x
```

[3] consultare appendice A per approfondire la definizione di struttura dati ad albero.

La procedura per trovare l'elemento massimo, invece, è simmetrica alla procedura appena analizzata. In pseudocodifica:

```
1 treeMaximum(x):
2   while x.right != NIL:
3     x := x.right
4   return x
```

Entrambe queste procedure vengono eseguite in un tempo $O(h)$, in un albero di altezza h.

Successore e Predecessore Dato un nodo in un albero binario di ricerca, spesso è necessario trovare il suo successore, nell'ordine stabilito da un attraversamento simmetrico. Il successore di un nodo x è il nodo con la più piccola chiave maggiore di $x.key$. La struttura di un albero binario consente di trovare tale nodo senza il bisogno di confrontare le chiavi. In pseudocodifica:

```
1 treeSuccessor(x):
2   if x.right != NIL:
3     return treeMinimum(x.right)
4   y := x.parent
5   while y != NIL and x = y.right:
6     x := y
7     y := y.parent
8   return y
```

Se il sottoalbero destro del nodo x non è vuoto, allora il successore di x è proprio il nodo più a sinistra nel sottoalbero destro, che viene trovato chiamando la procedura precedentemente analizzata `treeMinimum`. Altrimenti, se il sottoalbero destro del nodo x è vuoto e x ha un successore y, allora y è l'antenato più prossimo di x il cui figlio sinistro è anche antenato di x. Per trovare il nodo y, è necessario risalire l'albero partendo da x, fino a quando non si trova un nodo che è figlio sinistro di suo padre. Questa operazione è svolta in un tempo $O(h)$, con h l'altezza dell'albero.

La procedura per trovare il successore è simmetrica alla procedura appena anlizzata e viene svolta nello stesso tempo.

Inserimento Per inserire un nuovo valore v all'interno dell'albero binario di ricerca T, si utilizza la seguente procedura, che riceve in input un nodo z, tale per cui $z.key = v$:

```
1  treeInsert(T, z):
2    y := NIL
3    x := T.root
4    while x != NIL:
5      y := x
6      if z.key < x.key:
7        x := x.left
8      else:
9        x := z.right
10   z.parent := y
11   if y = NIL:
12     T.root = z
13   else if z.key < y.key:
14     y.left := z
15   else:
16     y.right := z
```

La procedura appena vista inizia dalla radice dell'albero e il puntatore x traccia un cammino semplice in discesa cercando un `NIL` da sostituire con l'elemento di input z. La procedura mantiene anche un puntatore y detto inseguitore che punta sempre al padre di x. Le righe 4-9 del ciclo ***while*** spostano questi due puntatori verso il basso, andando a sinistra o a destra a seconda dell'esito del confronto fra $z.key$ e $x.key$, finchè a x non viene assegnato il valore `NIL`. A questo punto si rende necessario il puntatore y perchè, quando si arriva a trovare il valore `NIL` da sostituire con il nodo z, la ricerca è andata un passo

oltre il nodo che deve essere modificato. Le righe successive inseriscono il nodo passato come argomento all'interno dell'albero binario di ricerca. In generale, questo algoritmo viene eseguito in un tempo $O(h)$, in un albero di altezza h.

Cancellazione La procedura per la cancellazione di un nodo z da un albero binario di ricerca si suddivide in tre casi:

1. se il nodo z non ha figli, si modifica il nodo $z.parent$ in modo che non punti più a z, ma a `NIL`;
2. Se il nodo z ha un solo figlio, si eleva il figlio di tale nodo in modo che occupi la posizione di z nell'albero, modificando il padre di z affinchè punti al figlio di z;
3. Se il nodo z ha due figli, si trova prima di tutto il successore y di z, che deve necessariamente trovarsi nel sottoalbero destro di z, per poi fare in modo che y assuma la posizione di z nell'albero. La parte restante del sottoalbero destro originale diventa il nuovo sottoalbero destro di y e il sottoalbero sinistro di z diventa il nuovo sottoalbero sinistro di y.

Per poter spostare i sottoalberi all'interno dell'albero binario di ricerca si deve prima definire una procedura che sostituisca il sottoalbero figlio di suo padre con un altro sottoalbero. Questa procedura sosituisce il sottoalbero con radice nel nodo u con il sottoalbero con radice nel nodo v, in modo che il padre del nodo u diventa il padre del nodo v. In pseudocodifica:

```
1  transplant(T, u, v):
2    if u.parent = NIL:
3      T.root := v
4    else if u = u.parent.left:
5      u.parent.left := v
6    else:
7      u.parent.right := v
8    if v != NIL:
9      v.parent := u.parent
```

Una volta introdotto questo algoritmo, è possibile ora analizzare la procedura di eliminazione:

```
1  treeDelete(T, z):
2    if z.left = NIL:
3      transplant(T, z, z.right)
4    else if z.right = NIL:
5      transplant(T, z, z.left)
6    else:
7      y := treeMinimum(z.right)
8      if y.parent != z:
9        transplant(T, y, y.right)
10       y.right := z.right
11       y.right.parent := y
12     transplant(T, z, y)
13     y.left := z.left
14     y.left.parent := y
```

Questa procedura gestisce i tre casi analizzati precedentemente. Le righe 2-3 gestiscono il caso in cui il nodo z non ha figlio sinistro, le righe 4-5 gestiscono il caso in cui il nodo z ha un figlio sinistro, ma non un figlio destro, mentre le restanti righe gestiscono il caso in cui il nodo ha sia figlio destro che sinistro. Nello specifico, la riga 7 trova il nodo y, successore di z: poichè z ha un sottoalbero destro non vuoto, il suo successore deve essere il nodo di quel sottoalbero con la chiave più piccola (per questo motivo viene invocata la procedura `treeMinimum`). Come già detto, y non ha figlio sinistro, quindi bisogna staccare y dalla sua posizione corrente e metterlo al posto di z. Se y è figlio destro di z, le ultime tre righe sostituiscono z con y, per poi sostituire il figlio sinistro di y con il figlio sinistro di z. Se y non è figlio sinistro di z, le righe 9-11 sostituiscono y con il figlio destro di y e cambiano il figlio destro di z nel figlio destro di y. Tale procedura richiede un tempo di esecuzione nell'ordine di $O(h)$, in un albero binario di ricerca alto h.

9.6 Alberi Red Black

Gli alberi rosso nero sono strutture dati ad albero di ricerca binaria con un bit aggiuntivo di memoria per ogni nodo, che rappresenta il colore di tale nodo. Questa struttura dati, garantisce che nessun cammino semplice dalla radice dell'albero fino ad una sua qualsiasi foglia si più lungo del doppio di qualsiasi altro cammino: si dice, quindi, che l'albero è approssimativamente bilanciato. Ogni nodo dell'albero contiene ora gli attributi `key`, `left`, `right`, `parent` e `color`. Se un nodo non contiene riferimenti a figli o al nodo padre, il corrispondente attributo viene inizializzato con il valore `NIL`. In questo caso, i puntatori a `NIL` vengono trattati come puntatori a nodi (vuoti) esterni dell'albero, mentre i nodi che contengono informazioni sono trattati come nodi interni all'albero: dunque, tutte le foglie dell'albero sono nodi che non contengono nessuna informazione. In realtà, per semplificare le condizioni al contorno nello pseudocodice, si è soliti utilizzare un unico nodo sentinella inizializzato a `NIL`, in modo che per ogni albero T, le sue foglie puntano al nodo $T.nil$ di color nero.

I nodi dell'albero rosso nero vengono colorati tramite le seguenti regole:

1. Ogni nodo è rosso o nero;
2. La radice è nera;
3. Ogni foglia (vuota - `NIL`) è nera;
4. Se un nodo è rosso, allora entrambi i suoi figli sono neri;
5. Per ogni nodo, tutti i cammini semplici, che vanno dal nodo alle sue foglie discendenti, contengono lo stesso numero di nodi neri.

Si definisce a questo proposito, altezza nera di un nodo x, indicato con $bh(x)$, il numero di nodi neri lungo un cammino semplice che inizia dal nodo x (non incluso). Per la proprietà degli alberi rosso nero, il concetto di altezza nera è ben definito in quanto tutti i cammini semplici che scendono dal nodo hanno lo stesso numero di nodi neri. In generale:

Teorema 9.6.1. *L'altezza massima di un albero rosso nero con n nodi è $2\,log_2(n+1)$.*

La conseguenza immediata di questo teorema è che le operazioni sugli insiemi dinamici possono tutte essere implementate in un tempo $O(log_2\, n)$ negli alberi rosso neri, perchè possono essere eseguite nel tempo $O(h)$ in un albero binario di ricerca di altezza h e qualsiasi albero rosso nero di n nodi è un albero binario di ricerca di altezza $O(log_2\, n)$.

Rotazione Le operazioni di `treeInsert` e `treeDelete` sugli alberi rosso nero, potrebbero violare le proprietà della struttura, proprio perchè modificano l'albero. Si rende quindi necessario ricalcolare i colori di qualche nodo dell'albero e anche la struttura dei puntatori, dopo la chiamata a una delle due procedure. La struttura dei puntatori viene modificata tramite una rotazione, ovvero un'operazione locale che preserva le proprietà degli alberi binari di ricerca. Quando si esegue una rotazione sinistra in un nodo x, supponendo che il nodo $y = x.right$ sia diverso da `NIL`, si fa "perno" sul collegamento tra x e y: il nodo $y$diventa la nuova radice del sottoalbero, con x come figlio sinistro di y e il figlio sinistro di y come figlio destro di x. In pseudocodifica:

```
1  leftRotate(T, x):
2    y := x.right
3    x.right := y.left
4    if y.left != T.nil:
5      y.left.parent := x
6    y.parent := x.parent
7    if x.parent = T.nil:
8      T.root := y
9    else if x = x.parent.left:
10     x.parent.left := y
11   else:
12     x.parent.right := y
13   y.left := x
14   x.parent := y
```

Il codice per la procedura `rightRotate` è simmetrico a quello appena analizzato. Entrambe le procedure vengono eseguite nel tempo $O(1)$.

Inserimento L'inserimento di un nodo in un albero rosso nero viene eseguito tramite una versione leggermente modificata della procedura di inserimento analizzata per gli alberi binari di ricerca. Inoltre, per poter preservare le condizioni degli alberi rosso neri, è necessario utilizzare una seconda procedura che ricolora i nodi dell'albero in seguito alla chiamata della procedura di inserimento. In pseudocodifica:

```
1  RBInsert(T, z):
2    y := T.nil
3    x := T.root
4    while x != T.nil:
5      y := x
6      if z.key < x.key:
7        x := x.left
8      else:
9        x := x.right
10   z.parent := y
11   if y = T.nil:
12     T.root := z
13   else if z.key < y.key:
14     y.left := z
15   else:
16     y.right := z
17   z.left := T.nil
18   z.right := T.nil
19   z.color = RED
20   RBFixup(T, z)
```

Dunque, tramite questa procedura, il nodo z viene inserito nella posizione corretta all'interno dell'albero rosso nero, inizializzando gli attributi `left` e`right` a `NIL` (per rispettare la struttura dell'albero) e il suo colore a rosso. Dato che l'inserimento di questo nodo potrebbe aver causato la violazione delle proprietà fondamentali degli alberi rosso neri, viene chiamata la procedura di ripristino `RBFixup`, implementata con la seguente pseudocodifica:

```
1  RBInsertFixup(T, z):
2    while z.parent.color = RED:
3      if z.parent = z.parent.parent.left:
4        y := z.parent.parent.right
5        if y.color = RED:
6          z.parent.color := BLACK                    // caso 1
7          y.color := BLACK                           // caso 1
8          z.parent.parent.color := RED               // caso 1
9          z := z.parent.parent                       // caso 1
10       else:
11         if z = z.parent.right:
12           z := z.parent                            // caso 2
13           leftRotate(T, z)                         // caso 2
14         z.parent.color := BLACK                    // caso 3
15         z.parent.parent.color := RED               // caso 3
16         rightRotate(T, z.parent.parent)            // caso 3
17     else:
18       come righe 4-16 con "left" e "right" scambiati
19   T.root.color := BLACK
```

Una volta chiamata la funzione di inserimento, sicuramente la proprietà 1 (ogni nodo è rosso o nero) e la proprietà 3 (ogni foglia è nera) sono rispettate in quanto i figli del nodo rosso appena inserito sono $T.nil$. Anche la proprietà 5 (per ogni nodo, tutti i cammini semplici che vanno dal nodo alle sue foglie discendenti contengono lo stesso numero di nodi neri) è rispettata proprio perchè il nodo inserito

nell'albero è di color rosso. Le uniche due proprietà che potrebbero non essere rispettate sono la 2 (la radice è nera) e la 4 (se un nodo è rosso, allora entrambi i suoi figli sono neri). Entrambe le possibili violazioni sono dovute al fatto che il nodo inserito viene colorato di rosso: infatti, se l'albero è vuoto, il nodo viene inserito alla radice violando la proprietà 2, mentre se il padre del nodo inserito è rosso, viene violata la proprietà 4.

Dunque, nel caso ci fossero violazioni delle proprietà degli alberi rosso nero, ce ne sarebbe solo una e riguarderebbe le due proprietà di cui si è appena discusso. Esistono 3 casi possibili, segnati anche all'interno dello pseudocodice nei commenti, per la ricolorazione dei nodi:

Caso 1 Lo zio y di z è rosso: il caso 1 (righe 6-9) viene eseguito quando $z.parent$ e y sono entrambi rossi. Poichè $z.parent.parent$ è nero, è possibile colorare di nero $z.parent$ e y risolvendo il problema per cui z e $z.parent$ sono entrambi rossi. Si ricolora poi di rosso $z.parent.parent$ per conservare la proprietà 5 e si ripete il ciclo ***while*** con $z.parent.parent$ come nuovo nodo z.

Caso 2 Lo zio y di z è nero e z è figlio destro: nel caso 2 (righe 12-13), lo zio di z è di colore nero e il nodo z è figlio destro. In questo caso è necessaria una semplice rotazione a sinistra per riportarsi immediatamente al caso 3. Una rotazione del genere non influisce nè sull'altezza dell'albero nè sulla proprietà 5.

Caso 3 Lo zio y di z è nero e z è figlio sinistro: nel caso 3 (righe 14-16), lo zio di z è sempre nero, ma il nodo z è figlio sinistro anzichè destro. In questo caso si applicano alcune modifiche sui colori e alcune rotazioni aggiuntive a destra per poter preservare le proprietà degli alberi rosso nero.

Rimozione La procedura di rimozione di un determinato nodo dall'albero rosso nero si basa sulla subroutine `transplant`, che deve essere opportunamente modificata in modo da adattarsi alla struttura degli alberi rosso nero. In pseudocodifica:

```
1  RBTransplant(T, u, v):
2    if u.parent = T.nil:
3      T.root := v
4    else if u = u.parent.left:
5      u.parent.left := v
6    else:
7      u.parent.right := v
8    v.parent := u.parent
```

Una volta introdotta questa procedura, è possibile analizzare l'algoritmo di rimozione, simile a quella analizzata per gli alberi di ricerca binaria, con l'unica differenza che si tiene traccia del nodo y, che potrebbe violare le proprietà degli alberi rosso nero. Per evitare tali violazioni, si utilizza una seconda procedura di supporto che ha il compito di ricalcolare i colori dell'albero, in modo da rispettare le sue proprietà. In pseudocodifica:

```
1  RBDelete(T, z):
2    y := z
3    yOriginalColor := y.color
4    if z.left = T.nil:
5      x := z.right
6      RBTransplant(T, z, z.right)
7    else if z.right = T.nil
8      x := z.left
9      RBTransplant(T, z, z.left)
10   else:
11     y := treeMinimum(z.right)
12     yOriginalColor := y.color
13     x := y.right
14     if y.parent = z:
15       x.parent := y
16     else:
17       RBTransplant(T, y, y.right)
18       y.right := z.right
```

```
      y.right.parent := y
    RBTransplant(T, z, y)
    y.left := z.left
    y.left.parent := y
    y.color := z.color
  if yOriginalColor = BLACK:
    RBDeleteFixup(T, x)
```

```
RBDeleteFixup(T, x):
  while x != T.root and x.color = BLACK:
    if x = x.parent.left:
      w := x.parent.right
      if w.color = RED:
        w.color := BLACK                                    // caso 1
        x.parent.color := RED                               // caso 1
        leftRotate(T, x.parent)                             // caso 1
        w := x.parent.right                                 // caso 1
      if w.left.color = BLACK and w.roght.colr = BLACK:
        w.color := RED                                      // caso 2
        x := x.parent                                       // caso 2
      else:
        if w.right.color = BLACK:
          w.left.color = BLACK                              // caso 3
          w.color = RED                                     // caso 3
          rightRotate(T, w)                                 // caso 3
          w := x.parent.right                               // caso 3
        w.color := x.parent.color                           // caso 4
        x.parent.color := BLACK                             // caso 4
        w.right.color := BLACK                              // caso 4
        leftRotate(T, x.parent)                             // caso 4
        x := T.root                                         // caso 4
    else:
      come righe 4-23 con "left" e "right" scambiati
  x.color := BLACK
```

9.7 Grafi

Un grafo G è una coppia (V, E), in cui V è l'insieme finito dei vertici del grafo, mentre E è l'insieme degli archi di G. Un arco è una relazione binaria su V che collega due vertici detti adiacenti: se un arco e connette due vertici u e v, allora può essere rappresentato come la coppia (u, v) dei vertici che connette. Da questo si ottiene che l'insieme $E \subseteq V^2$ e quindi che $0 \leq |E| \leq |V|^2$.

Esistono due tipologie di grafi, ovvero i grafi orientati e non orientati. Un grafo si dice non orientato quando la coppia (u, v) è la stessa di (v, u) e non c'è quindi la nozione di direzione da un nodo all'altro. D'altra parte, in un grafo orientato, la coppia (u, v) è differente dall'arco (v, u), in quanto si tiene conto della direzione di attraversamento.

9.7.1 Rappresentazione

Esistono due metodi principali per la rappresentazione dei grafi: la prima consiste in una collezione di liste di adiacenza, mentre la seconda in una matrice di adiacenza. Entrambi i metodi possono essere applicati sia ai grafi orientati che non orientati.

Rappresentazione con liste di adiacenza Tale rappresentazione di un grafo $G = (E, V)$ consiste in un array Adj di $|V|$ liste concatenate, una per ogni vertice V. Per ogni $u \in V$, la lista di adicenza $Adj[u]$ include tutti i vertici adiacenti a u in G. Poichè le liste di adiacenza rappresentano gli archi di un grafo, nello pseudocodice si tratta l'array Adj come un attributo del grafo G.

Inoltre, Se G è un grafo orientato, la somma delle lunghezze di tutte le liste di adiacenza è esattamente $|E|$, perchè un arco nella forma (u, v) è rappresentato inserendo v in $G.Adj[u]$. Al contrario, se G è un grafo non orientato, la somma delle lunghezze di tutte le liste di adiacenza è $2|E|$, perchè se (u, v) rappresenta un arco non orientato, allora u appare nella lista di adiacenza di v e viceversa. Per i grafi orientati e non, la rappresentazione con liste di adiacenza ha la proprietà di occupare una quantità di memoria $\Theta(V + E)$.

Lo svantaggio principale di questo metodo di rappresentazione risiede nel fatto che il metodo di ricerca è abbastanza inefficiente: infatti, per poter determinare se un determinato arco (u, v) è presente nel grafo, è necessario cercare v nella lista di adiacenza $G.Adj[u]$, operazione che richiede un tempo lineare.

Rappresentazione con matrice di adiacenza Tale rappresentazione di un grafo $G = (E, V)$ consiste in una matrice $A = (a_{ij})$ di dimensione $|V| \times |V|$ tale per cui:

$$a_{i,j} = \begin{cases} 1 & if\ (i, j) \in E \\ 0 & else \end{cases}$$

Per poter procedere con tale rappresentazione, è prima necessario che i vertici siano numerati in maniera arbitraria da 1 a $|V|$. Si noti che, la matrice di adiacenza A per un grafo non orientato è uguale alla sua trasposta A^T, per la proprietà principale dei grafi non orientati, per cui $(u, v) = (v, u)$.

Lo svantaggio maggiore nell'impiego di una matrice di adiacenza è l'utilizzo intensivo di memoria, nell'ordine di $\Theta(|V|^2)$. Nonostante questo, per grafi di dimensione relativamente piccole, si preferisce utilizzare questo metodo a quello precedente, per la sua semplicità di utilizzo. In generale conviene utilizzare il metodo delle liste di adiacenza quando $|E| \neq \Theta(|V|^2)$, cioè quando il grafo è sparso (ovvero quando il numero di nodi connessi non è molto grande), mentre se il grafo è completo (ovvero quando ogni nodo è connesso con i restanti), conviene utilizzare una matrice di adiacenza.

9.7.2 Visita in Ampiezza

La visita in ampiezza (o breadth-first search) è l'algoritmo più semplice di ricerca nei grafi. Dato un grafo G è un vertice distinto s, detto sorgente, la visita in ampiezza ispeziona sistematicamente gli archi di G per scoprire tutti i vertici che sono raggiungibili da s, calcolando la distanza da s a ciascun vertice raggiungibile. L'algoritmo è così chiamato perchè espande la frontiera fra i vertici scoperti e quelli ancora da scoprire in maniera uniforme lungo l'ampiezza della frontiera: l'algoritmo, infatti, scopre tutti i vertici che si trovano a distanza k da s, prima di scoprire tutti i ivertici che si trovano a distanza $k + 1$.

Per tenere traccia del lavoro, la visita colora i vertici di bianco, grigio oppure nero. Inizialmente, tutti i vertici sono di colore bianco e possono diventare grigi e poi neri. Un vertice viene scoperto quando viene incontrato per la prima volta durante la visita, cessando di essere bianco. Tutti i vertici adiacenti ai vertici neri sono di colore grigio, il che significa che sono stati scoperti dall'algoritmo. I vertici grigi, invece, potrebbero presentare alcuni vertici adiacenti bianchi, che rappresentano la frontiera da scoprire: tali vertici sono tenuti in memoria in una coda (gestita dalla politica FIFO) e, ad ogni iterazione, si elimina dalla coda un elemento u, visitando i vertici a lui adiacenti che sono ancora bianchi. Si noti che, se $u.dist$ è la distanza del vertice u da s, allora la distanza dei nodi bianchi adiacenti ad u è $u.dist + 1$.

In pseudocodifica:

```
1  breadthFirstSearch(G, s):
2    for each u in G.V - {s}:
3      u.color := WHITE
4      u.dist := ∞
5    s.color := GRAY
6    s.dist := 0
7    Q := ∅
8    enqueue(Q, s)
9    while Q != ∅:
10     u := dequeue(Q)
11     for each v in u.Adj:
12       if v.color = WHITE:
13         v.color := GRAY
14         v.dist := u.dist + 1
15         enqueue(q, v)
16     u.color := BLACK
```

Informalmente, le righe 2-4 inizializzano ogni vertice, colorandolo di bianco, eccetto il vertice s che in riga 5 viene colorato di grigio, in quanto è il vertice sorgente già visitato ad inizio ricerca, e a cui viene assegnata la distanza 0. Le righe 7 e 8 inizializzano la coda Q inserendo il solo vertice s. Il ciclo ***while*** di riga 10-16 viene reiterato fin tanto che ci sono ancora vertici grigi, la cui frontiera protrebbe ancora presentare vertici di colore bianco. La riga 10 determina il vertice grigio u che si trova in testa alla coda, rimuovendolo dalla coda stessa (`dequeue`), in modo da poter accedere ai suoi vertici adiacenti. Il ciclo ***for***, di righe 11-15, esamina ciascun vertice v della lista di adiacenza di u: se questo vertice è bianco, allora non è ancora stato scoperto e, quindi, l'algoritmo lo colora di grigio e lo inserisce nella coda dopo avergli assegnato il valore di distanza $u.dist + 1$. Dopo aver esaminato ogni vertice nella lista di adiacenza di u, tale vertice viene colorato di nero in riga 16.

Dato un grafo $G = (V, E)$, il tempo di esecuzione dell'algoritmo BFS è $O(|E| + |V|)$.

9.7.3 Visita in Profondità

La visita in profondità (o depth-first search) è un algoritmo che adotta una strategia differente rispetto al precedente: consiste, infatti, nel visitare il grafo sempre più in profondità se possibile. Nella visita in profondità, gli archi vengono ispezionati a partire dall'ultimo vertice v scoperto, che ha ancora archi non ispezionati nella sua lista di adiacenza. Questo processo continua fino a quando non sono stati scoperti tutti i vertici che sono raggiungibili dal vertice v da cui inizia la ricerca. Se rimane ancora qualche vertice da ispezionare, allora se ne sceglie uno nuovo come sorgente e si reitera il processo.

Dunque, si può affermare che se l'algoritmo BFS si basa sulla strategia di visitare i vertici secondo una politica LIFO, l'algoritmo DFS si basa su una politica di tipo FIFO: una volta inserito un nodo in cima alla pila, si visitano tutti i vertici adiacenti relativi a quel nodo, prima di procedere alla visita dei vertici adiacenti del successivo.

Come per il precedente algoritmo, anche in questo caso i vertici vengono colorati durante la loro visita, per indicarne lo stato. Inizialmente tutti i vertici sono colorati di bianco e nel momento in cui un determinato vertice viene visitato per la prima volta, questo viene colorato di grigio. Infine, un vertice è colorato di nero nel momento in vui sono ispezionati tutti i suoi vertici adiacenti. Oltre alla lista di adiacenza e al colore, i vertici all'interno di un grafo visitato in profondità, contengono anche due informazioni temporali: la prima, memorizzata in $v.d$, registra il momento in cui il vertice viene scoperto e colorato di grigio, mentre le seconda, memorizzata in $v.f$, registra il momento in cui la visita completa l'ispezione della lista di adiacenza di v e viene colorato di nero. Quindi, un vertice v è bianco prima del tempo $v.d$, grigio fra il tempo $v.d$ e $v.f$, nero successivamente. Queste informazioni aggiuntive servono per agevolare la scrittura di procedure e sono utilizzate anche da parte di algoritmi più avanzati.

Si introduce di seguito la procedura in pseudocodifica che realizza la strategia di visita in profondità:

```
1 depthFirstSearch(G):
2   for each u in G.V:
3     u.color := WHITE
4   time := 0
5   for each u in G.V:
6     if u.color = WHITE:
7       DFSVisit(G, u)
```

La procedura imposta, in righe 2-3, il colore di tutti i vertici di bianco, per poi inizializzare una variabile globale `time`,tramite cui si memorizzano le informazioni temporali precedentemente discusse. Le righe successive servono ad ispezionare in profondità tutti i vertici che hanno colore bianco, tramite la procedura di supporto `DFSVisit`, introdotta di seguito.

```
1  DFSVisit(G, u):
2    time := time + 1
3    u.d := time
4    u.color := GREY
5    for each v in G.Adj[u]:
6      if v.color = WHITE:
7        DFSVisit(G, v)
8    u.color := BLACK
9    time := time + 1
10   u.f := time
```

In ogni chiamata della procedura `DFSVisit`, il vertice u passato come argomento è inizialmente bianco, quindi gli si assegna subito il colore grigio dopo aver assegnato al vertice il tempo di scoperta ($u.d$) il tempo incrementato di una unità (righe 2-4). Subito dopo, il ciclo ***for*** di righe 5-7 esplora tutti i vertici v di colore bianco adiacenti ad u e chiama ricorsivamente la procedura in modo da visitare il vertice v e i vertici a lui adiacenti, sempre se di colore bianco. Dopo aver terminato la visita in profondità di tutti i vertici a partire dal vertice sorgente u, il colore di tale vertice viene cambiato in nero e si assegna al tempo di fine esplorazione $u.f$ il valore temporale, dopo averlo nuovamente incrementato di una unità (righe 8-10).

Anche in questo caso, come per il procedimento precedente, il tempo medio di esecuzione dell'algoritmo di ispezione in profondità è $\Theta(|V| + |E|)$.

9.7.4 Ordinamento Topologico

La visita in profondità può essere utilizzata per eseguire l'ordinamento topologico di un determinato grafo orientato aciclico (detto anche DAG - Directed Acyclic Graph). Un ordinamento topologico di un dag $G = (V, E)$ è un ordinamento lineare di tutti i suoi vertici, tale che se G contiene un arco (u, v), allora u appare prima di v nell'ordinamento. Un ordinamento topologico di un grafo può essere visto come un ordinamento dei suoi vertici lungo una linea orizzontale ion modo che tutti gli archi orientati siano diretti da sinistra verso destra: intuitivamente, questo tipo di ordinamento, serve a rappresentare la precedenza di determinati eventi.

In pseudocodifica:

```
1 topologicalSort(G):
2   L := ∅
3   for each u in G.V:
4     u.color := WHITE
5   for each u in G.V:
6     if u.color = WHITE:
7       topologicalSortVisit(L, u)
8   return L
```

```
1 topologicalSortVisit(L, u):
2   u.color := GRAY
3   for each v in G.Adj[u]
4     if v.color = WHITE:
5       topologicalSortVisit(L, v)
6   x.key := u
7   listInsert(L, x)
8   u.color := BLACK
```

L'idea di questo algoritmo è quella di visitare il dag con l'algoritmo di vista DFS e, quando un vertice viene colorato di nero, lo si inserisce in testa alla lista L che rappresenta tutti i nodi ordinati. Una volta completata la visita di ogni vertice, la lista ottenuta rappresenta l'ordinamento topologico del grafo.

Anche in questo caso, l'algoritmo impiega un tempo di esecuzione medio $\Theta(|V| + |E|)$.