Appunti di "Algoritmi e Principi dell'informatica"

Alexandru Gabriel Bradatan

19 settembre 2021

Indice

1 Introduzione ai modelli **2**

2 I linguaggi **2**
2.1 Costruzione di un linguaggio . . . 2
2.2 Operazioni sui linguaggi . . . 3
2.3 A cosa utilizzeremo i linguaggi? . . . 3

3 Modelli operazionali **4**
3.1 Automi a stati finiti . . . 4
3.1.1 Automi a stati finiti traduttori . . . 5
3.1.2 Analisi degli automi a stati finiti . . . 6
3.1.3 Proprietà di chiusura degli automi a stati finiti . . . 7
3.2 Automi a stati finiti con pila . . . 8
3.2.1 Analisi degli automi a pila . . . 10
3.3 Macchine di Turing . . . 10
3.3.1 Proprietà di chiusura delle macchine di Turing . . . 12
3.3.2 Modelli equivalenti di macchine di Turing . . . 12
3.4 Modelli operazionali non deterministici . . . 12
3.4.1 Automi a stati finiti non deterministici . . . 13
3.4.2 Automi a pila non deterministici . . . 13
3.4.3 Macchine di Turing non deterministiche . . . 14

4 Modelli descrittivi **15**
4.1 Le grammatiche . . . 15
4.2 Espressività di una grammatica . . . 16
4.3 Legame tra grammatiche e automi . . . 17

5 Altri modelli utili **19**
5.1 I pattern . . . 19
5.2 Le espressioni regolari . . . 19

6 Modelli dichiarativi **19**
6.1 La logica . . . 19
6.1.1 Precondizioni e postcondizioni . . . 20
6.2 La logica monadica del primo ordine . . . 21
6.2.1 Proprietà . . . 22
6.3 Logica monadica del secondo ordine . . . 22
6.3.1 Trasformare da MSO a FSA . . . 23

7 Teoria della computazione **23**
7.1 Formalizzazione di problema . . . 23
7.2 Gli algoritmi . . . 24
7.3 Enumerazione algoritmica . . . 24
7.4 Macchina di Turing universale . . . 25
7.5 Funzioni calcolabili e problemi definibili . . . 25
7.5.1 Specializzazioni e generalizzazioni . . . 26
7.5.2 Il problema delle funzioni totali . . . 26
7.6 Decidibilità e semidecidibilità . . . 27
7.7 Stabilire decidibilità e semidecidibilità di un problema . . . 29
7.7.1 La tecnica di riduzione di un problema . . . 31
7.8 Problemi notoriamente decidibili o indecidibili . . . 32

8 Complessità del calcolo **32**
8.1 Complessità degli altri modelli visti fino ad ora . . . 34
8.2 Teoremi di accelerazione lineare . . . 35
8.3 Un nuovo modello di calcolo: la macchina RAM . . . 35
8.4 Limiti del criterio di costo . . . 36
8.5 Legami tra le complessità dei vari modelli di calcolo . . . 37

9 Studio della complessità di algoritmi **39**
9.1 La sintassi dello pseudocodice . . . 39
9.2 Studio di algoritmi ricorsivi . . . 40
9.2.1 Metodo di sostituzione . . . 41
9.2.2 Studio dell'albero di ricorsione . . . 42
9.2.3 Teorema dell'esperto . . . 42
9.3 Algoritmi di ordinamento . . . 43
9.3.1 Insertion sort . . . 43
9.3.2 Merge sort . . . 44
9.3.3 Quicksort . . . 45
9.3.4 Counting sort . . . 46

10 Strutture dati **47**
10.1 Vettori . . . 47
10.2 Liste semplicemente connesse . . . 48
10.3 Pile o stack . . . 48
10.4 Code o queues . . . 48
10.5 Mazzo (deque) . . . 49
10.6 Dizionari . . . 49
10.6.1 L'approccio ingenuo . . . 49
10.6.2 Tabelle Hash . . . 50
10.7 Alberi binari . . . 52
10.7.1 Alberi binari di ricerca . . . 53
10.7.2 Alberi RB . . . 55
10.7.3 Mucchi (heap) . . . 59
10.8 Grafi . . . 61

A Costo delle istruzioni RAM in criterio di costo logaritmico **66**

B Riassunto sulle proprietà di chiusura dei linguaggi **66**

C Somme e approssimazioni notevoli **67**

1 Introduzione ai modelli

I modelli sono fondamentali nell'ingegneria. I modelli sono talvolta fisici e spesso sono modelli formali, ossia oggetti matematici che fungono da rappresentazioni astratte di entità reali complesse. Un modello è adeguato se i risultati ottenuti riflettono le proprietà che ci interessano del sistema fisico entro i limiti della nostra approssimazione. I modelli dell'informatica si basano principalmente sulla matematica discreta. Definiamo i due tipi di modelli che costruiremo.

Definizione 1.1 (Modello operazionale)**.** È un modello basato sul concetto di stato e di meccanismo per la sua evoluzione.

Definizione 1.2 (Modello descrittivo)**.** È un modello che formula le proprietà desiderate o no del sistema piuttosto che il suo funzionamento.

Le differenze tra questi due tipi di modellizzazione non sono spesso molto ben definite. Le fasi dell'ingegneria del software ricalcano quelli della modellizzazione di un problema:

- Analisi dei requisiti: Stesura della specifica del sistema
- Progetto: Architettura del software
- Implementazione: Scrittura effettiva del codice

Esempio 1.1. Modellizziamo lo stesso problema, l'ordinamento di una sequenza di interi, secondo i due criteri enunciati sopra:

- Modello operazionale: Calcola il minimo in tutto l'array e mettilo al primo posto. Continua ad eseguire l'operazione finché l'array non è in ordine.
- Modello descrittivo: Individua una permutazione degli elementi dell'array tale che $\forall i a[i] \leq a[i+1]$.

2 I linguaggi

Il meta-modello fondamentale che useremo sarà il linguaggio. Il termine linguaggio è un termine che conosciamo già ed è utilizzabile a diversi ambiti diversi come la linguistica, l'informatica, la grafica e la musica.

2.1 Costruzione di un linguaggio

Iniziamo a definire i vari elementi un linguaggio. Il primo elemento formale di un linguaggio è l'alfabeto o vocabolario. In matematica essi sono sinonimi anche se in italiano naturale non lo sono.

Definizione 2.1 (Alfabeto)**.** Si dice alfabeto un insieme finito A di simboli base.

Una volta definito il concetto di alfabeto possiamo anche definire il concetto di stringa.

Definizione 2.2 (Stringa)**.** Si dice stringa una sequenza ordinata e finita di elementi dell'alfabeto A.

Naturalmente ogni stringa possiede una lunghezza $|a|$ pari al numero di elementi dell'alfabeto contenuti al suo interno. Definiamo anche la stringa nulla ϵ tale che $|\epsilon| = 0$. Definiamo infine $A^\star$ l'insieme di tutte le stringhe scrivibili con un certo alfabeto.

Nota. L'operatore $\star$ è detto stella, star, iterazione o stella di Kleene. L'insieme $A^\star$ è infinito numerabile.

Sulle stringhe possiamo definire l'operazione di concatenazione:

Definizione 2.3 (Concatenazione di stringhe)**.**

$$\begin{array}{rccc} . : & A^\star \times A^\star & \to & A^\star \\ & (x, y) & \mapsto & x.y \end{array} \tag{1}$$

Nota. La scrittura dell'operatore di concatenazione può essere omessa, scrivendo al posto di $z = x.y$ $z = xy$.

L'operazione di concatenazione gode della proprietà associativa ma non della commutativa e ha come elemento neutro la stringa vuota. Possiamo quindi definire il monoide non commutativo delle stringhe rispetto alla concatenazione $\langle A^\star, . \rangle$.

Definito tutti questi elementi possiamo finalmente definire un linguaggio.

Definizione 2.4 (Linguaggio)**.** Chiamiamo un linguaggio un insieme L tale che: $L \subseteq A^\star$.

Notiamo che L può anche essere infinito. Inoltre, poiché L è un insieme, le operazioni insiemistiche sono tutte ben definite. Un insieme di linguaggi che condividono le stesso proprietà è detto famiglia di linguaggi.

2.2 Operazioni sui linguaggi

Definizione 2.5 (Concatenazione di linguaggi)**.**

$$L_1.L_2 = \{x.y : x \in L_1, y \in L_2\} \tag{2}$$

Definizione 2.6 (Potenze di linguaggi)**.**

$$\begin{aligned} L^0 &= \{\epsilon\} \\ L^i &= L^{i-1}.L \end{aligned} \tag{3}$$

Definizione 2.7 (Star (Linguaggi))**.**

$$L^\star = \bigcup_{i=0}^{\infty} L^i \tag{4}$$

Definizione 2.8 (Plus (Linguaggi))**.**

$$L^+ = \bigcup_{i=1}^{\infty} L^i \tag{5}$$

Nota. La differenza tra scrivere $L^\star$ e L^+ è che in L^+ è assente il linguaggio vuoto!

2.3 A cosa utilizzeremo i linguaggi?

Il nostro principale utilizzo del concetto di linguaggio sarà per definire in maniera astratta il concetto di problema informatico. Il nostro primo problema informatico sarà:

$$x \in A^\star, L \subseteq A^\star \quad x \in L? \tag{6}$$

Questa semplice questione è capace di modellizzare una grande varietà di problemi diversi, scelti A ed L in modo adatto. Un secondo problema sarà quello di trovare una traduzione, ossia una funzione così definita:

Definizione 2.9 (Traduzione)**.**

$$\begin{array}{rccc} \tau: & L_1 & \to & L_2 \\ & x & \mapsto & \tau(x) \end{array} \tag{7}$$

3 Modelli operazionali

3.1 Automi a stati finiti

Gli automi a stati finiti (FSA) sono il modello operazionale più semplice. Essi sono caratterizzati da un insieme finito di stati e da un insieme di regole di transizione. Gli stati possono essere di accettazione o no. Le regole di transizione permettono al nostro automa di passare da uno stato all'altro in base a ciò che forniamo come input. Quando un automa riceve un input lo elabora e produce un output. L'elaborazione inizia in uno stato iniziale e consiste nel leggere l'input e spostarsi da uno stato all'altro secondo le leggi di transizione. Se alla fine della lettura l'automa si trova in uno stato di accettazione, diremo che esso ha accettato l'input, altrimenti diremo che l'ha rifiutato.

Definizione 3.1 (Automa a stati finiti)**.** Un automa a stati finiti $\mathcal{A}$ è una quintupla $\langle Q, I, \delta, q_0, F\rangle$ dove:

1. Q è l'insieme finito non vuoto di tutti gli stati;
2. I è l'alfabeto di ingresso;
3. $\delta : Q \times I \to Q$ è la funzione (relazione) di transizione;
4. $q_0 \in Q$ è lo stato iniziale;
5. $F \subseteq Q$ è l'insieme degli stati di accettazione.

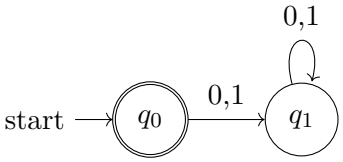

Figura 1: Esempio di diagramma di stato di FSA.

Se δ è una funzione l'automa si dice completo, altrimenti non completo. Gli automi non completi possono facilmente resi completi "riempiendo" le transizioni mancanti con delle transizioni verso uno stato di "sink" dal quale non si può più uscire.

Un automa processa l'input tramite una sequenza di transizioni di stato effettuate iterando su ogni carattere della stringa e muovendosi allo stato corrispondente a $\delta(q, i)$. Definiamo formalmente la sequenza di mosse eseguita da un automa a stati finiti.

Definizione 3.2 (Sequenza di mosse)**.** Sia un FSA $\mathcal{A}$, la sequenza di mosse è una funzione:

$$\delta^\star : Q \times I^\star \to Q \tag{8}$$

Definita induttivamente da:

1. $\delta^\star(q, \epsilon) = q$
2. $\delta^\star(q, y.i) = \delta(\delta^\star(q, y), i)$ con $y \in I^\star, i \in I$

Gli FSA rappresentano dei linguaggi, più nello specifico la famiglia dei linguaggi regolari (**REG**). Se L è un linguaggio accettato da $\mathcal{A}$ allora diremo che esso è il linguaggio di $\mathcal{A}$ e lo indichiamo come $L(\mathcal{A})$. Usando la definizione di sequenza di mosse possiamo scrivere $L(\mathcal{A})$ come:

$$L(\mathcal{A}) = \{x \in I^\star : \delta^\star(q_0, x) \in F\} \tag{9}$$

Per rappresentare il linguaggio accettato da un FSA possiamo usare la pura notazione insiemistica oppure, visto che i linguaggi sono regolari, delle espressioni regolari. Definiamo la sintassi delle espressioni regolari.

Definizione 3.3 (Espressioni regolari)**.** Dato una alfabeto A, siano le seguenti operazioni:

- $R + S = R \cup S$ con $R \subseteq A^\star$ e $S \subseteq A^\star$ (notazione equivalente: $R|S$)
- $\star$ la star di Kleene: $R^\star = \{x^n : n \in \mathbb{N}, x \in R\}$
- $R^+ = \{x^n : n \in \mathbb{N}^*, x \in R\}$
- la concatenazione: $RS = \{xy : x \in R, y \in S\}$

Allora, dato un alfabeto A e un insieme di simboli $\{+, \star, (,), ., \emptyset\}$ si dice espressione regolare su A la stringa $R \in A \cup \{+, \star, (,), ., \emptyset\}$ che rende vera una delle seguenti condizioni:

1. $R = \emptyset$;
2. $R \in A$;
3. $R = S + T$, $R = ST$, $R = S^\star$, $R = S^+$ con S, T espressioni regolari su A.

Gli automi a stato finito sono modelli di calcolo con una memoria finita pari al numero di stati. Infatti ogni stato rappresenta una istantanea della situazione in cui si trova il sistema in un dato istante. Il fatto di possedere una memoria finita, come vedremo in 3.1.2, è uno degli aspetti più limitanti degli automi a stato finito e ci costringeranno a costruire modelli più sofisticati.

3.1.1 Automi a stati finiti traduttori

Un FSA può anche essere usato come traduttore tra un linguaggio ed un altro. Ci basta aggiungere la capacità di dare un output al nostro automa.

Un semplice FSA si limita semplicemente ad interpretare un input senza produrre nessun output. Se ad un FSA aggiungiamo la possibilità di produrre un output otteniamo un trasduttore. Un particolare tipo di trasduttore che ci interessa è il traduttore.

Prima di parlare di traduttori, però, formalizziamo come un automa produce un output.

Definizione 3.4 (Funzione di transizione con uscita)**.** Chiamiamo $\delta(q, i/w)$ con $i \in I$ e $w \in O$, I, O alfabeti, una funzione di transizione tra stati che restituisce il nuovo stato del FSA e un simbolo complesso w.

Ora possiamo definire il traduttore.

Definizione 3.5 (Automa a stati finiti traduttore)**.** Sia $\mathcal{A} = \langle Q, I, \delta, q_0, F \rangle$ un automa a stati finiti con funzione di transizione $\delta(q, i/w)$, definiamo automa a stati finiti traduttore $\mathcal{T}$ la terna $\langle \mathcal{A}, O, \eta \rangle$ dove:

- O è l'alfabeto di uscita di δ;
- $\eta : Q \times I \to O^\star$ funzione di traduzione.

Analogamente a quanto fatto in 3.2 possiamo iterare la funzione di traduzione usando la stella di Kleene, così da poter finalmente enunciare la traduzione come:

$$\tau(x) = \eta^\star(q_0, x) \tag{10}$$

Se il traduttore conclude la propria esecuzione su uno stato di accettazione, allora possiamo dire che la stringa di input e corretta e la sua traduzione nel linguaggio di output è $\tau(x)$.

3.1.2 Analisi degli automi a stati finiti

Gli automi a stati finiti sono un modello molto semplice ed intuitivo, applicato a molti settori. Abbiamo già visto in 3.1 che sono adatti a modellare linguaggi regolari e che sono modelli a memoria finita. Studiamone ora nel dettaglio le proprietà ed eventuali limitazioni.

Innanzitutto esiste qualche condizione affinché un automa a stati finiti sia o no accettore di un linguaggi finito o infinito (il caso di linguaggio vuoto è banale)? Si dimostra che esistono due condizioni necessarie e sufficienti per l'accettazione dei linguaggi, una per il caso finito e un per l'infinito.

Teorema 3.1 (Condizione di accettazione di un linguaggio finito)**.** *Condizione sufficiente e necessaria affinché un automa a stati finiti accetti un linguaggio finito non vuoto è che possa accettare una stringa di lunghezza inferiore al numero di stati ($|x| < |Q|$).*

Teorema 3.2 (Condizione di accettazione di un linguaggio infinito)**.** *Condizione sufficiente e necessaria affinché un automa a stati finiti accetti un linguaggio infinito è che possa accettare una stringa tale che: $|Q| \leq |x| < 2|Q|$.*

Questo teorema deriva dall'osservazione che se un automa accetta un linguaggio infinito, nel suo grafo saranno presenti dei cicli che potranno essere percorsi un numero arbitrario di volte. Nel caso peggiore verranno ripercorsi tutti gli stati dell'automa tranne l'ultimo che deve essere di accettazione.

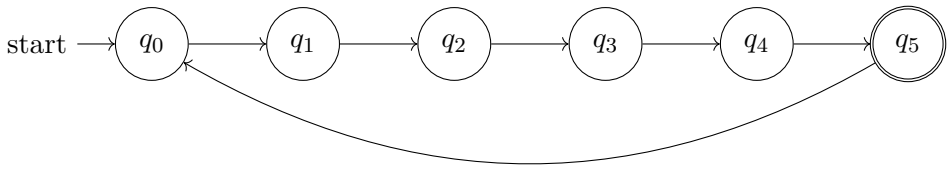

Figura 2: Peggior caso del teorema 3.2.

Possiamo quindi vedere una stringa di un linguaggio infinito come composta da 3 parti: un preambolo al ciclo, il ciclo e l'epilogo. Il ciclo può ripetersi un numero infinito di volte, generando la seguente espressione regolare:

$$x = x_i x_c^\star x_f \tag{11}$$

Questa è una condizione necessaria affinché una stringa di un linguaggio infinito sia riconosciuta da un FSA e rappresenta il contenuto del "Pumping lemma":

Lemma 3.1 (Pumping lemma)**.** *Sia un automa a stati finiti $\mathcal{A}$ che accetta un linguaggio $L(\mathcal{A})$. Allora per ogni $x \in L$ con $|x| > |Q|$ esiste $q \in Q$ e $w \in I^+$ tali da verificare una di queste condizioni:*

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

Come conseguenza si ha: $\forall n \in \mathbb{N} \geq 0, yw^n z \in L$.

Il Pumping lemma, quindi, ci mette dei paletti sui tipi di linguaggi infiniti che un FSA può accettare. Infatti consideriamo il seguente linguaggio infinito:

$$L = \{a^n b^n : n \in \mathbb{N}\} \tag{12}$$

Supponiamo che un FSA sia in grado di accettarlo, allora consideriamo la stringa $x = a^m b^m$ con $m > |Q|$ e applichiamo il Pumping lemma. Otterremo 3 casi possibili:

1. $w = a^p$ e quindi dovrebbe essere:

$$x = a^m b^m = a^r a^p a^s b^m = a^r w a^s b^m \in L \text{ con } r + p + s = m \tag{13}$$

   Se il Pumping lemma dovesse valere, allora si dovrebbe avere che $a^r w^k b^m \in L$ che ci porta ad un assurdo.

2. $w = b^p$ analogo al precedente.

3. $w = a^p b^q$ e quindi dovrebbe essere:

$$x = a^m b^m = a^r a^p b^q b^s = a^r w b^s \in L \text{ con } r + p = q + s = m \tag{14}$$

Se il Pumping lemma dovesse valere, allora si dovrebbe avere che $a^r w^k b^s \in L$ che ci porta ad un assurdo.

Ciò significa che considerare L come accettato da una FSA è un assurdo. Il linguaggio L appena costruito ci dimostra quindi la necessità di costruire modelli di calcolo più potenti. Infatti per contare fino a n non basta la memoria finita degli FSA ma servirebbe una memoria infinita!

3.1.3 Proprietà di chiusura degli automi a stati finiti

Il concetto di chiusura di un insieme rispetto ad un'operazione o proprietà è un concetto già affrontato nel precedente corso di logica e algebra lineare. In questo contesto ci occupiamo della chiusura della famiglia dei linguaggi regolari.

La famiglia dei linguaggi regolari è chiusa rispetto a tutte le operazioni insiemistiche, alla concatenazione, alla star di Kleene e praticamente tutte le altre viste fino ad ora. Proviamo a costruirne qualcuno.

Intersezione Proviamo a costruire l'intersezione di due automi. Il risultato sarà un automa che accetta solo stringe solo stringhe accettate da entrambi gli automi di partenza. Dati i due automi di partenza:

$$A^1 = \langle Q^1, I, \delta^1, q_0^1, F^1 \rangle \tag{15}$$
$$A^2 = \langle Q^2, I, \delta^2, q_0^2, F^2 \rangle \tag{16}$$

Possiamo scrivere l'automa intersezione come:

$$\langle A^1, A^2 \rangle = \langle Q^1 \times Q^2, I, \delta, \langle q_0^1, q_0^2 \rangle, F^1 \times F^2 \rangle \tag{17}$$
$$\delta(\langle q^1, q^2 \rangle, i) = \langle \delta(q^1, i), \delta(q^2, i) \rangle \tag{18}$$

Con una semplice induzione si può dimostrare che il linguaggio di un automa così definito è $L(\langle A^1, A^2 \rangle) = L(A^1) \cap L(A^2)$.

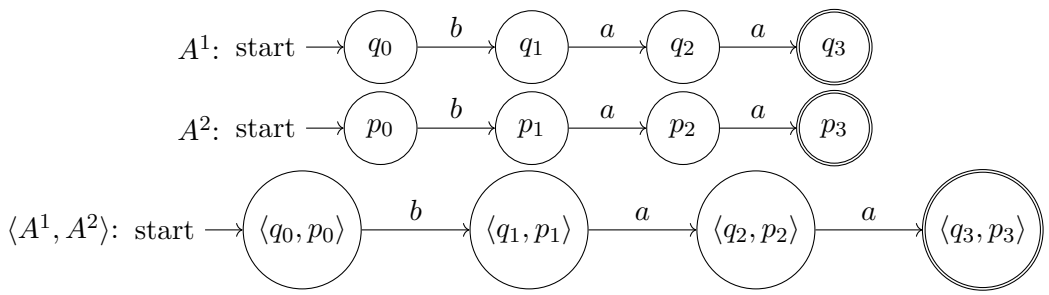

Figura 3: Due automi e la loro intersezione.

Unione Per l'unione il tutto funziona in modo analogo. L'automa risultante sarà un automa che accetterà stringhe accettate da almeno uno dei due automi di partenza. Eseguendo un procedimento simile a quello effettuato per ottenere 17 abbiamo:

$$\langle A^1, A^2 \rangle = \langle Q^1 \times Q^2, I, \delta, \langle q_0^1, q_0^2 \rangle, F^1 \times Q^2 \cup Q^1 \times F^2 \rangle \tag{19}$$

Questo approccio presenta, però, un problema: non funziona se l'automa di partenza non è completo. Con l'intersezione non avevamo problemi perché l'automa intersezione accettava solo stringhe accettate da entrambi gli automi iniziali, rimuovendo il problema della non completezza. Con l'unione invece otterremo un automa che accetta anche quando uno degli automi iniziali non accetta, rendendo problematico il caso di un errore dovuto alla parzialità della funzione di transizione. Bisogna, quindi, ricordarsi di completare gli automi aggiungendo eventuali stati di errore prima di eseguirne l'unione.

Complemento Per il complemento la situazione è analoga all'unione: anche qui dobbiamo stare attenti alla completezza dei due automi di partenza. L'automa complemento sarà praticamente uguale all'automa di partenza, solo che gli stati di accettazione ora saranno di non accettazione e viceversa.

$$\neg A = \langle Q, I, \delta, q_0, Q \setminus F \rangle \tag{20}$$

3.2 Automi a stati finiti con pila

Arricchiamo ora i nostri automi a stati finiti con della memoria. Questa memoria sarà strutturata come una pila (stack) di dimensione potenzialmente illimitata. L'automa può manipolare la pila tramite le due operazioni fondamentali delle pile: push e pop. L'automa così strutturato si chiama "automa a pila", in breve PDA (dall'inglese "Push Down Automata"). Indicheremo convenzionalmente l'inizio della pila con Z_0.

A grandi linee, una mossa dell'automa a pila è strutturata in diversi passi:

1. Leggi un simbolo (o nulla) dall'input (d'ora in poi lo chiameremo nastro d'ingresso);
2. Esegui una pop di un elemento dalla stack;
3. Cambia stato;
4. Sposta di una posizione il puntatore del carattere corrente (la testina del nastro d'ingresso);
5. Esegui una push di una serie di caratteri (anche nulla);
6. Se è un automa traduttore, scrivi una stringa (anche nulla) sull'output (nastro d'uscita).

Come per gli FSA, la stringa in ingresso viene riconosciuta se l'automa la scandisce completamente e termina su uno stato di accettazione. Lo stato della pila non è rilevante. Se l'automa è traduttore, allora se accetta la stringa l'output corrisponde alla stringa tradotta, altrimenti viene detta indefinita e lo indichiamo con $\tau(x) = \perp$. In generale useremo il simbolo $\perp$ anche con la funzione δ per indicare una transizione indefinita.

Poiché le transizioni di stato sono più ricche di quelle degli FSA, dobbiamo adottare una notazione ben definita per indicare le varie azioni compiute:

$$a, A/B \ldots, c \tag{21}$$

- a il carattere letto in input;
- A il carattere letto dalla pila tramite stack;
- $B \ldots$ i caratteri reinseriti nella pila tramite la push;
- c il carattere scritto sull'output.

Nel caso in cui l'automa non leggesse nulla dal nastro d'ingresso diciamo che l'automa ha effettuato una ϵ-mossa e la indichiamo riportando ϵ come carattere letto dal nastro d'ingresso. Vale lo stesso anche con il carattere della push.

Nota. Un automa a pila non può non leggere nessun carattere dalla pila! Scrivere, quindi, $a, \epsilon/B$ è sbagliato.

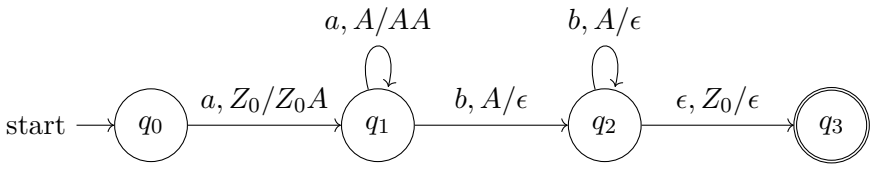

Figura 4: Esempio di un automa a pila.

Formalizziamo ora i concetti introdotti.

Definizione 3.6 (Automa a stati finiti a pila)**.** Definiamo un automa a stati finiti con pila $\mathcal{A}$ una eptupla $\langle I, \Gamma, \delta, q_0, Z_0, F\rangle$ dove:

- Q, I, q_0, F sono definiti alla stessa maniera di un FSA;
- Γ è l'alfabeto di pila;
- Z_0 è il simbolo iniziale di pila;
- $\delta : Q \times (I \cup \epsilon) \times \Gamma \to Q \times \Gamma^\star$ è la funzione di transizione.

Definizione 3.7 (Automa a stati finiti a pila traduttore)**.** Definiamo un automa a stati finiti con pila $\mathcal{A}$ una ennupla $\langle I, \Gamma, \delta, q_0, Z_0, F, 0, \eta\rangle$ dove:

- Q, I, q_0, F, O sono definiti alla stessa maniera di un FSA;
- Γ è l'alfabeto di pila;
- Z_0 è il simbolo iniziale di pila;
- $\delta : Q \times (I \cup \epsilon) \times \Gamma \to Q \times \Gamma^\star$ è la funzione di transizione;
- $\eta : Q \times (I \cup \epsilon) \times \Gamma \to O^\star$ è la funzione di traduzione.

Nota. La funzione η è definita ovunque lo è anche δ.

La funzione di δ deve essere parziale perché l'esistenza delle ϵ-mosse causerebbe non-determinismo in caso di completezza dell'automa: non possono esistere una ϵ-mossa tra due stati q, q' se tra questi due stati esiste già un'altra mossa che legge lo stesso simbolo dalla pila. Un automa a stati finiti non può decidere deterministicamente tra una ϵ-mossa e una regolare. Ciò che abbiamo detto fino ad ora può essere formalizzato come:

$$\forall q, A\delta(q, \epsilon, A) \neq \bot \implies \forall i\delta(q, i, A) = \bot \tag{22}$$

Il concetto intuitivo di stato che avevamo introdotto con gli FSA oramai non è più adeguato in quanto è troppo semplicistico. Negli automi a pila, infatti, si aggiunge anche lo stato della pila che anch'esso contribuisce allo stato generale dell'automa. Formalizziamo, quindi, la generalizzazione dello stato: la configurazione.

Definizione 3.8 (Configurazione)**.** Chiamiamo configurazione di un automa a pila la tripla (se traduttore quadrupla) $c = \langle q, x\gamma\rangle$ ($\langle q, x\gamma, z\rangle$) dove:

- $q \in Q$ è lo stato dell'organo di controllo;

- x è la stringa ancora da leggere nel nastro d'ingresso;
- γ è la stringa di caratteri nella pila (rappresentati usando la convenzione "alto-destra basso-sinistra").
- z la stringa già scritta sul nastro di uscita.

Tra le configurazioni di un automa a pila è presente una relazione di transizione indicata con $\vdash$. Possiamo esprimere il passaggio di stato come:

$$c = \langle q, x, \gamma \rangle \vdash c' = \langle q', x', \gamma' \rangle \tag{23}$$

Indichiamo con $\vdash^\star$ la chiusura transitiva e riflessiva di $\vdash$. Essa non è altro che l'insieme delle nostre vecchie sequenze di mosse. Usando $\vdash^\star$ possiamo anche formalizzare l'accettazione di una stringa (e la sua eventuale traduzione) come:

$$c_0 = \langle q_0, x, Z_0, (\epsilon) \rangle \vdash^\star c_f = \langle q, \epsilon, \gamma, (z) \rangle \tag{24}$$

Nota. D'ora in poi indicheremo la chiusura transitiva e riflessiva di una relazione A con $A^\star$. La chiusura solo transitiva sarà A^+.

3.2.1 Analisi degli automi a pila

Come anche per gli FSA, è facile trovare un linguaggi non riconosciuti dagli automi a pila. Uno di questi è:

$$L = \{a^n b^n c^n\} \tag{25}$$

L'incapacità di riconoscere il linguaggio 25 può essere ricondotta al fatto che la pila è una memoria distruttiva: bisogna distruggere ciò che ci è salvato dentro per leggerla (pop). Quanto appena detto si dimostra con una estensione al Pumping lemma. Noi non la tratteremo.

Studiamo ora la chiusura della famiglia dei linguaggi riconosciuti da un automa a pila. Prendiamo due linguaggi $L_1 = \{a^n b^n\}, L_2 = \{a^n b^{2n}\}$. Essi sono individualmente riconosciuti dagli automi a pila, la loro unione però non lo è (ovviamente si può dimostrare). Ciò ci indica che la famiglia dei linguaggi riconosciuti dagli automi a pila non è chiusa rispetto all'unione e (per motivi analoghi) neanche rispetto all'intersezione. Per quanto riguarda invece il complemento? Usiamo la stessa idea degli automi a stati finiti: scambiamo gli stati di accettazione con quelli di non accettazione. Anche in questo caso δ va "completata" cercando di evitare di cadere nel non-determinismo. Le ϵ-mosse possono ancora causare problemi:

- Si può creare un ciclo di ϵ-mosse che fa entrare l'automa in blocco;
- Può esserci una sequenza di ϵ-mosse in cui si alternano stati di accettazione e di non accettazione.

In entrambi i casi si possono costruire automi equivalenti che eliminano questi problemi, nel primo caso eliminando i cicli, nel secondo forzando l'accettazione alla fine di una sequenza di ϵ-mosse.

3.3 Macchine di Turing

L'ultimo automa che tratteremo sarà la macchina di Turing. Partiamo dalla versione a k-nastri, un po' più semplice di quella originaria, ma che gode delle stesse proprietà.

Come indica il nome, la macchina di Turing a k-nastri è analoga a un automa a stati finiti al quale aggiungiamo k nastri di memoria. Le testine dei nastri possono muoversi in ambo le direzioni arbitrariamente. Come anche per gli altri automi avremo i soliti stati e alfabeti. Per convenzione storica, i nastri sono rappresentati da sequenze infinite di celle invece che da stringhe finite. Per rappresentare una cella inutilizzata usiamo il simbolo speciale "blank", rappresentato da uno spazio vuoto, _ o $\not b$. Assumeremo che ogni nastro contenga solo un numero finito di celle non contenenti blank. Come nel caso degli automi a pila indichiamo l'inizio del nastro con Z_0.

La mossa della macchina di Turing è simile a quella dell'automa a pila ma un po' più articolata. Possiamo suddividerla in due "fasi":

1. Lettura:
   - Legge il carattere in corrispondenza della testina del nastro d'ingresso;
   - Legge i k caratteri dai nastri;
   - Valuta lo stato dell'organo di controllo.
2. Scrittura:
   - Cambia stato;
   - Scrittura di un carattere sui nastri di memoria;
   - Eventuale scrittura di un carattere sul nastro di uscita;
   - Spostamento delle testine di una posizione.

Introduciamo anche la nuova notazione per le transizioni (tra parentesi è riportata la parte aggiuntiva nel caso di presenza di output):

$$i, \langle A_1, \ldots, A_k \rangle / (o), \langle A'_1, \ldots, A'_k \rangle, \langle M_0, \ldots, M_k, (M_{k+1}) \rangle \tag{26}$$

- i: carattere letto dal nastro di ingresso;
- $\langle A_1, \ldots, A_k \rangle$: le letture dai vari nastri;
- o: il carattere scritto sul nastro di uscita;
- $\langle A'_1, \ldots, A'_k \rangle$: le scritture sui vari nastri;
- $\langle M_0, \ldots, M_k, (M_{k+1}) \rangle$: i movimenti effettuati dai vari nastri dove:
  - M_0 è il movimento della testina di ingresso;
  - M_{k+1} è il movimento della testina di output.

Le nuove funzioni di transizione e traduzioni saranno le seguenti:

$$\delta : Q \times I \times \Gamma^k \to Q \times \Gamma^k \times \{R, L, S\}^{k+1} \tag{27}$$

$$\eta : Q \times I \times \Gamma^k \to Q \times \Gamma^k \times \{R, L, S\}^{k+1} \times O \times \{R, S\} \tag{28}$$

Lo stato iniziale di una macchina di Turing è come ce lo immagineremmo: Z_0 seguito da blank nei nastri, uscita tutta blank, testine in posizione 0 per ogni nastro, organo di controllo nello stato iniziale e stringa iniziale scritta sul nastro iniziale a partire dalla posizione 0 seguita da blank. Le condizioni di terminazione, e quindi di accettazione, sono leggermente diverse da quelle viste fino ad ora:

- Gli stati di accettazione sono sempre $F \subseteq Q$;
- Per convenzione la $\delta(\eta)$ non è definita a partire dagli stati finali:

$$\forall q \in F \delta(q, \ldots) = \perp, (\eta(q, \ldots) = \perp) \tag{29}$$

- La macchina si ferma in uno stato q quando $\delta(q, \ldots) = \perp$;
- La stringa in ingresso è accettata se e solo se dopo un numero finito di transizioni la macchina si ferma in uno stato di accettazione.

Ciò significa che una stringa in ingresso non è accettata se la macchina si ferma in uno stato di accettazione o se la macchina non si ferma.

3.3.1 Proprietà di chiusura delle macchine di Turing

La famiglia dei linguaggi riconosciuti dalle macchine di Turing è chiusa per:

- $\cup$ simulazione di esecuzione "in parallelo";
- $\cap$ simulazione di esecuzione "in serie";
- . simile a $\cap$;
- $\star$ come concatenazione.

Per quanto riguarda il complemento la situazione è diversa. Il principale problema sta nel fatto che non esistono macchine di Turing loop-free, ossia macchine di Turing equivalenti ad una data ma con eventuali cicli rimossi (come si poteva fare per PDA e FSA). A causa di ciò il rischio di generare computazioni infinite non è eliminabile. La dimostrazione di ciò verrà vista in seguito nella parte di computabilità del corso.

3.3.2 Modelli equivalenti di macchine di Turing

Noi abbiamo introdotto il modello della macchina di Turing a k nastri. Si può dimostrare, però, che esistono diverse formulazioni equivalenti della macchina di Turing. Queste diverse tipologie di macchina sono funzionalmente equivalenti, con unica differenza la complessità dell'organo di controllo.

TM a singolo nastro Il primo modello, e anche il più vecchio, che enunceremo è quello della TM a singolo nastro. Essa, come implica il nome, possiede un singolo nastro di memoria infinito sul quale la testina si può muovere in entrambe le direzioni. Questo unico nastro viene usato come memoria, input ed eventualmente output. Attenzione a non confonderla con la TM con $k = 1$ nastri (chiamata "ad un nastro di memoria").

TM con nastro di memoria bidimensionale Un altro modello interessante è quello in cui il nastro di memoria è organizzato come una tabella e la testina si può muovere in entrambe e 4 le direzioni cardinali sequenzialmente. Si ricorda che l'accesso randomico alla memoria non è possibile con una semplice TM.

Macchine di von Neumann La macchina di von Neumann è il modello astratto di un computer con processore, memoria ad accesso randomico e periferiche. Le macchine di Turing possono simulare anche questo modello già più complesso. Il risultato però è una macchina molto più complessa e lenta. Ciò vale in generale: si possono costruire modelli più prestanti e complessi della macchina di Turing, ma essi non ne aumentano la capacità espressiva.

3.4 Modelli operazionali non deterministici

I modelli visti fino ad ora sono detti deterministici: in un certo stato e con certi ingressi la mossa eseguita è sempre la stessa. Se neghiamo questa ipotesi creiamo dei modelli detti non deterministici. Ciò significa che per un certo stato l'automa potrà eseguire più mosse diverse ed è lui a scegliere quale di queste eseguire. Si vengono a creare, quindi, dei fili (thread) di esecuzione che l'automa può seguire. Il modello dell'esecuzione di un automa non deterministico può visto come:

Parallelo L'automa esegue tutte le strade contemporaneamente. L'automa si ferma quando tutti le fili di esecuzione hanno terminato o per esaurimento dell'input o per la parzialità delle funzione di transizione.

Sequenziale L'automa esegue solo un filo, ma ad ogni biforcazione sceglie uno dei fili di esecuzione in modo casuale.

Di solito i modelli non deterministici tendono ad essere più compatti sacrificando l'operatività. In un contesto pratico essi possono essere realizzati utilizzando più esecutori, seguendo il modello parallelo. Vediamo le varianti non deterministiche dei modelli già noti.

3.4.1 Automi a stati finiti non deterministici

Iniziamo con il primo modello, e il più semplice gli automi a stati finiti non deterministici o NFA ("non-deterministic finite automata"). Diciamo che un FSA è non deterministico se esiste $\delta(q, a) = \{q_1, q_2\}$ ossia associati ad uno stato e ad un carattere è associato un insieme di transizioni. Sarà quindi necessario ridefinire δ e $\delta^\star$.

$$\delta : \ Q \times I \to \wp(Q) \tag{30}$$

$$\delta^\star(q, x) = \begin{cases} \delta^\star(q, \epsilon) = \{q\} & x = \epsilon \\ \delta^\star(q, y.i) = \bigcup_{q' \in \delta^\star(q,y)} \delta(q', i) & x = y.i, i \in I \end{cases} \tag{31}$$

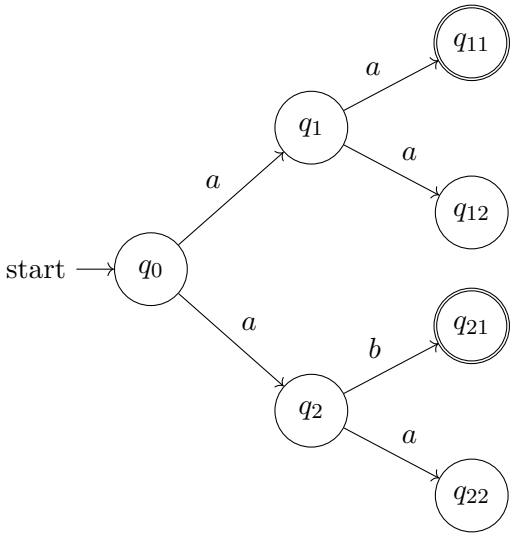

Figura 5: Un esempio di NFA.

Una stringa $x \in L$ appartiene al linguaggio modellato da un NFA se:

$$\delta^\star(q_0, x) \cap F \neq \emptyset \tag{32}$$

Ossia c'è almeno una sequenza di mosse che ci porta allo stato finale. È possibile anche considerare $\delta^\star(q_0, x) \subseteq F$.

È importante sottolineare che gli NFA non possiedono un potere riconoscitivo maggiore dei semplici FSA. Infatti possiamo sempre sintetizzare un FSA equivalente ad uno non deterministico. Intuitivamente, possiamo trasformare un automa non deterministico in uno deterministico semplicemente unendo in un singolo stato l'insieme di arrivo della δ e mantenendo gli archi. Possiamo anche operare nell'altro senso, ossia costruire un automa non deterministico a partire da uno deterministico.

3.4.2 Automi a pila non deterministici

Seguendo l'evoluzione svolta nella parte che tratta gli automi deterministici, introduciamo gli automi a pila non deterministici o NPDA. Il concetto di non determinismo è analogo a quello trattato in 3.4.1. L'avevamo già incontrato quando avevamo parlato delle ϵ-mosse. La nuova funzione di transizione è:

$$\delta : Q \times (I \cup \{\epsilon\}) \times \Gamma \to \wp_f(Q \times \Gamma^\star) \tag{33}$$

Il pedice f dell'insieme delle parti sta per "finito". Infatti i possibili sottoinsiemi di $Q \times \Gamma^\star$ sono infiniti, ma noi consideriamo solamente quelli ottenibili nell'immagine di δ, ottenendo un insieme delle parti finito. Un NPDA accetta se esiste una sequenza di mosse tale che:

$$c_0 \vdash^\star \{c_0, \ldots, c_n\}, \quad c_0 = \langle q_0, \epsilon, Z_0 \rangle, c_1 = \langle q, \epsilon, \gamma \rangle, \ldots, q \in F \tag{34}$$

Quindi la relazione $\vdash$ non è più univoca!

Una semplice costruzione come quella in figura 6 ci permette di costruire sempre l'unione di due NPDA e quindi di dimostrare la chiusura degli NPDA rispetto all'unione. Questa proprietà non è condivisa dagli PDA. Gli NPDA non sono, però, chiusi rispetto all'intersezione. Ciò implica che gli NPDA hanno un potere riconoscitivo maggiore della controparte deterministica.

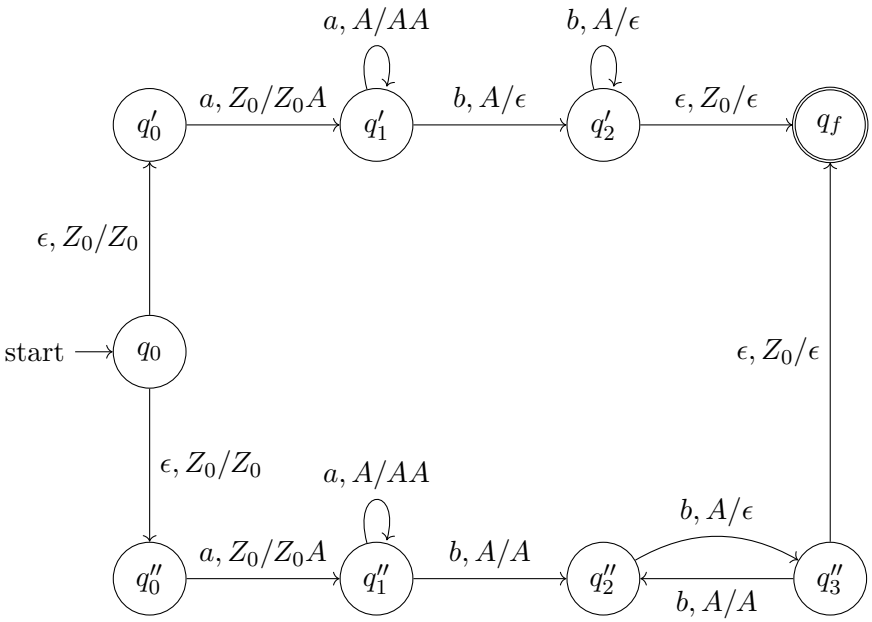

Figura 6: NPDA unione di due PDA. Lo NPDA riconosce $L = \{a^n b^n\} \cup \{a^n b^{2n}\}$. Da notare la ϵ-mossa non deterministica che parte da q_0 e lega i due automi, ognuno corrispondente ad uno dei due rami.

Se la famiglia dei linguaggi riconosciuti dagli NPDA è chiusa rispetto a $\cup$ ma non rispetto a $\cap$ non può esserlo rispetto al complemento a causa delle leggi di De Morgan.

Proviamo a dimostrare il risultato sopra. Procedendo con le stesse modalità del caso deterministico, possiamo ottenere una computazione che termina e accetta il linguaggio complemento. Se però consideriamo il caso:

$$\langle q_0, x, z_0 \rangle = c_0 \vdash^\star \{\langle q_1, \epsilon, \gamma_1 \rangle, \langle q_2, \epsilon, \gamma_2 \rangle\}, \; q_1 \in F, q_2 \notin F \tag{35}$$

La stringa x è accettata anche se scambio F con $Q \setminus F$, rendendo impossibile la costruzione del complemento.

3.4.3 Macchine di Turing non deterministiche

Definiamo il concetto di non determinismo in una macchina di Turing in modo analogo ai casi precedenti, ossia la capacità di poter assumere diversi stati contemporaneamente. Definiamo la relazione di transizione per le NTM (dall'inglese "nondeterministic Turing machines"):

$$\delta : Q \times I \times \Gamma^k \to \wp(Q \times \Gamma^k \times \{L, S, R\}) \tag{36}$$

Non ripeteremo la definizione di configurazione, transizione, sequenza di transizioni e accettazioni poiché la loro definizione è invariata.

Come nel caso degli NFA, le NTM non aggiungono capacità riconoscitiva. Per dimostrare ciò, costruiamo un algoritmo che permetta ad una macchina di Turing deterministica di emulare il comportamento di una non deterministica. Poiché una stringa è accettata da una NTM solo se esiste un calcolo che termina in uno stato di accettazione, rappresentiamo la computazione sotto forma di un albero chiamato appunto albero delle computazioni.

Per emulare una NTM con una TM ci basterà, quindi, percorrere in larghezza questo albero, scandendo le varie configurazioni finché non ne troveremo una di accettazione. Per eseguire questa operazione in alberi tradizionali esistono degli algoritmi ben consolidati ("breadth first search").

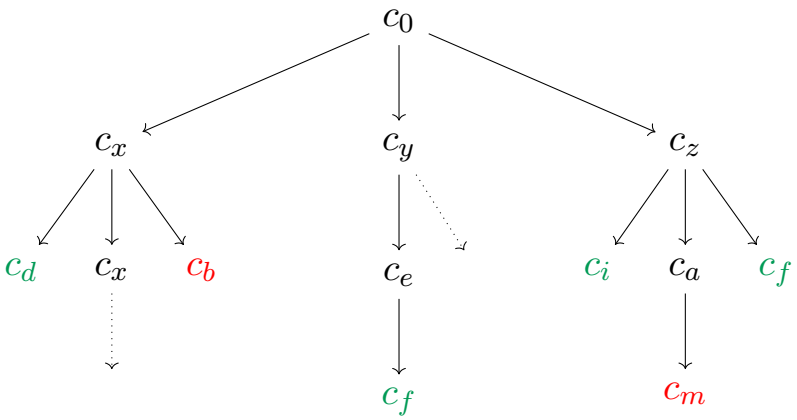

Figura 7: Un esempio di albero delle computazioni. In verde sono indicati le configurazioni di accettazione, in rosso quelle di non accettazione e con le linee tratteggiati i percorsi che porterebbero ad una computazione infinita.

4 Modelli descrittivi

4.1 Le grammatiche

Le grammatiche sono un altro tipo di modello che possiamo usare per modellare i linguaggi. A differenza degli automi, le grammatiche sono un modello generativo più che riconoscitivo. Una automa, infatti, legge una stringa in ingresso, la elabora e determina se appartiene al linguaggio; una grammatica, invece, descrive una serie di regole per generare le stringhe del linguaggio.

Come anticipato, possiamo intuitivamente definire le grammatiche come un insieme di regole usato per costruire le "frasi", sinonimo di stringhe, del linguaggio di interesse. In modo analogo alle grammatiche dei normali meccanismi linguistici, una grammatica formale genera le stringe di un linguaggio attraverso un processo di riscrittura. Esse descrivono un oggetto principale (la frase) come un insieme ordinato di componenti, a loro volta descritti come composti da altri componenti fino ad arrivare agli elementi fondamentali dell'alfabeto considerato.

Definizione 4.1 (Grammatica)**.** Si definisce grammatica G la quadrupla (V_n, V_t, P, S) dove:

- V_n è l'alfabeto o vocabolario non terminale;
- V_t è l'alfabeto o vocabolario terminale;
- $S \in V_n$ è l'assioma o simbolo iniziale;
- L'insieme di riscrittura o delle produzioni:

$$P \subseteq V_n^+ \times (V_t \cup V_n)^\star = \{S \to \alpha, \alpha \to \beta, \ldots\} \tag{37}$$

Nota. Per comodità indicheremo con V l'insieme $V_t \cup V_n$.

Definizione 4.2 (Relazione di derivazione immediata)**.** Definiamo la relazione di derivazione immediata $\underset{G}{\Longrightarrow}$ per una grammatica $G = (V_t, V_n, P, S)$ come $\alpha \underset{G}{\Longrightarrow} \beta$ se e solo se dati $\alpha \in V^+$, $\beta \in V^\star$:

1. $\alpha = \alpha_1\alpha_2\alpha_3$;
2. $\beta = \alpha_1\beta_2\alpha_3$;
3. $\alpha_2 \to \beta_2 \in P$.

Nota. Dove non ambiguo ometteremo il pedice indicante la grammatica per alleggerire la notazione. Indicheremo inoltre con $\overset{\star}{\Longrightarrow}$ la chiusura riflessiva e transitiva di $\Rightarrow$.

Definizione 4.3 (Linguaggio generato da una grammatica)**.** Definiamo $L(G)$ il linguaggio generato dalla grammatica $G = (V_t, V_n, P, S)$ un linguaggio tale che:

$$L(G) = \{x \in V_t^\star : S \overset{\star}{\Rightarrow} x\} \tag{38}$$

Esempio 4.1. Definiamo una prima grammatica semplice: $V_t = \{a, b, c\}$, $V_n = \{S, A, B, C\}$ con assioma S e produzioni $P = \{S \to A, A \to Aa, A \to B, B \to bB, B \to C, C \to cC, C \to \epsilon\}$. Una possibile derivazione consiste in $S \Rightarrow A \Rightarrow aA \Rightarrow aaA \Rightarrow aaB \Rightarrow aaC \Rightarrow aacC \Rightarrow aaccC \Rightarrow aacccC \Rightarrow aaccc$ oppure $S \Rightarrow A \Rightarrow B \Rightarrow bB \Rightarrow bC \Rightarrow b$. Evidentemente il linguaggio modellato è $L(G) = \{a^*b^*c^*\}$.

Esempio 4.2. Definiamo una grammatica un po' più elaborata: $V_t = \{a, b\}$, $V_n = \{S\}$, assioma S e produzioni $P = \{S \to aSbS, S \to \epsilon\}$. Studiando le varie derivazioni si vede che il linguaggio modellato è quello delle coppie di a, b "ben parentetizzate".

Esempio 4.3. Definiamo, infine, un'ultima grammatica ancora più complessa: $V_t = \{a, b, c\}$, $V_n = \{S, A, B, C, D\}$, assioma S e produzioni $P = \{S \to aACD, A \to aAC, A \to \epsilon, B \to b, CD \to BDc, CB \to BC, D \to \epsilon\}$. Il linguaggio generato da questa grammatica sarà $L(G) = \{a^n b^n c^n\}$.

4.2 Espressività di una grammatica

Nella sezione precedente abbiamo mostrato con degli esempi che è possibile generare linguaggi che sappiamo essere riconosciuti da degli automi (usando la potenza minima): il primo da un FSA, il secondo da un PDA e il terzo da una TM. È possibile organizzare le grammatiche in una gerarchia in base al loro potere generativo e inoltre è possibile che ci sia una relazione tra le potenze riconoscitive dei vari autonomi e le grammatiche?

Alla prima domanda ha già risposto Noam Chomsky con la sua gerarchia delle grammatiche. Essa è divisa in 4 classi a seconda delle limitazioni imposte sulla forma delle produzioni $\alpha \to \beta$ (tabella 1).

Tipo	Nome	Forma delle produzioni
0	Non limitate	—
1	Contestuali	$\lvert\alpha\rvert \leq \lvert\beta\rvert$
2	Non contestuali	$\lvert\alpha\rvert = 1$
3	Regolari	$\lvert\alpha\rvert = 1 \vee \beta \in V_t.V$

Tabella 1: La gerarchia delle grammatiche di Chomsky.

Utilizzando questa gerarchia una grammatica più potente genera tutti i linguaggi di una meno potente, andando a creare una situazione come in figura 8. L'inclusione della famiglia di grammatiche meno potenti in quella di grammatiche più potenti è però stretta?

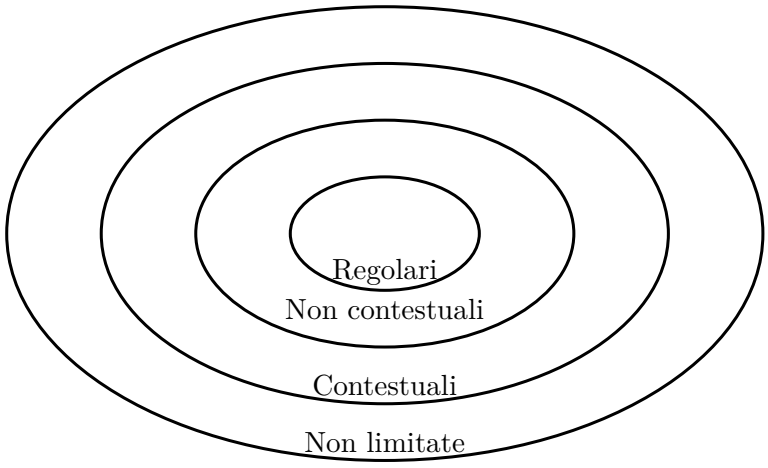

Figura 8: Diagramma di Venn rappresentante la gerarchia delle grammatiche.

4.3 Legame tra grammatiche e automi

Generalizziamo ora l'osservazione della corrispondenza grammatica-automa fatta negli esempi in 4.1.

Grammatiche regolari Come si potrebbe intuire, i linguaggi generati fa grammatiche regolari coincidono con i linguaggi riconosciuti dagli FSA (i linguaggi regolari). Poiché automa e grammatica sono in relazione di equivalenza possiamo passare da un all'altro (le due proposizioni si dimostrano con una semplice induzione):

Proposizione 4.1 (Passaggio da FSA a grammatica)**.** *Per ottenere la grammatica equivalente ad un FSA:*

- *Poniamo* $V_n = Q, V_t = I, S = \langle q_0 \rangle$*;*
- *Per ogni* $\delta(q, i) = q'$ *diciamo che* $\langle q \rangle \to i\langle q' \rangle \in P$*. Se* $q' \in F$ *aggiungiamo anche* $\langle q \rangle \to i \in P$*.*

Proposizione 4.2 (Passaggio da grammatica a NFA)**.** *Per ottenere lo NFA corrispondente alla grammatica:*

- *Poniamo* $Q = V_n \cup \{q_f\}, I = V_t, q_0 = S, F = \{q_f\}$*;*
- *Se* $A \to bC \in P$ *allora* $\delta(A, b) = C$*. Invece se* $A \to b \in P$ *allora* $\delta(A, b) = q_f$*.*

Grammatiche non contestali I linguaggi generati dalle grammatiche libere dal contesto coincidono con quelli riconosciuti dagli NPDA. La dimostrazione di questa affermazione non è affatto banale, perciò diamo solamente l'intuizione con la figura 9.

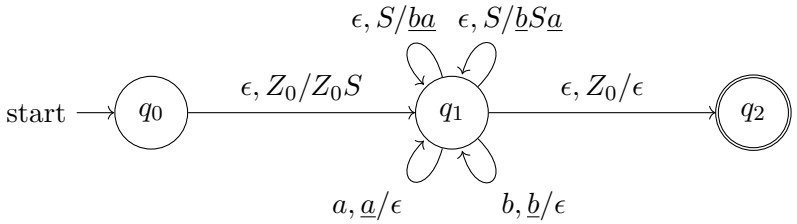

Figura 9: NPDA corrispondente a $(\{S\}, \{a, b\}\{S \to aSb, S \to ab\}, S)$.

Grammatiche non ristrette Le grammatiche non ristrette sono equivalenti alla macchine di Turing. Costruire una macchina di Turing (a nastro singolo) non deterministica a partire da una grammatica non ristretta G non è assai difficile: con la stringa x posizionata sul nastro d'ingresso eseguiamo un ciclo:

1. Scandiamo il nastro finché non troviamo una parte destra β di una produzione $\alpha \to \beta$ della grammatica;
2. Quando se ne trova una, scegliendola non deterministicamente, essa viene sostituita dalla corrispondente parte sinistra α.

Questo ciclo è chiamato "processo di riduzione" della stringa. Per come abbiamo strutturato il ciclo abbiamo che $\alpha \Rightarrow \beta$ se e solo se $\langle q, Z_0, \alpha \rangle \vdash^\star \langle q, Z_0, \beta \rangle$. Se e quando il contenuto del nastro d'ingresso diviene l'assioma S della grammatica, la stringa d'ingresso verrà accettata. Il procedimento descritto non garantisce che la macchina di Turing generata riesca a concludere sempre la computazione.

Grammatica equivalente a TM Eseguire il processo inverso è più complicato. Poiché una grammatica non può ricevere una stringa in input e può manipolare solo gli elementi non terminali di una stringa, dobbiamo fare in modo che essa generi tutte le possibili stringhe del tipo $x\blacklozenge X$ con $x \in V_t^\star$, $\blacklozenge \in V_n$ un separatore e X una "copia" di x fatta solo di elementi non terminali. Il nostro obiettivo è ottenere una derivazione $x\blacklozenge X \overset{\star}{\Rightarrow} x$ se e solo se x è accettata dalla TM. Innanzitutto la grammatica deve possedere le produzioni che generano $x\blacklozenge X$ (consideriamo $V_t = \{a, b\}$):

$$S \to SA'A, S \to SB'B, S \to \blacklozenge \quad \text{(genero coppie di simboli)} \tag{39}$$

$$AA' \to A'A, BA' \to A'B \quad \text{(scorro le } A' \text{ a sx)} \tag{40}$$

$$AB' \to B'A, BB' \to B'B \quad \text{(scorro le } B' \text{ a sx)} \tag{41}$$

$$\blacklozenge A' \to a\blacklozenge, B'\blacklozenge \to b\blacklozenge \quad \text{(quando scorro attraverso } \blacklozenge \text{ trasformo)} \tag{42}$$

Dobbiamo ora simulare ogni possibile mossa della macchina di Turing con una derivazione. Consideriamo una macchina di Turing con configurazione pari a quella rappresentata in figura 10. Tale configurazione è rappresentata dalla stringa $\blacklozenge\alpha BqAC\beta$. In base ai valori assunti da δ possiamo agire in 3 diversi modi:

1. $\delta(q, A) = \langle q', A', R\rangle$ aggiungo $qA \to A'q'$ alle produzioni;
2. $\delta(q, A) = \langle q', A', S\rangle$ aggiungo $qA \to q'A'$ alle produzioni;
3. $\delta(q, A) = \langle q', A', L\rangle$ aggiungo per ogni B nell'alfabeto della TM la produzione $BqA \to q'B'A'$.

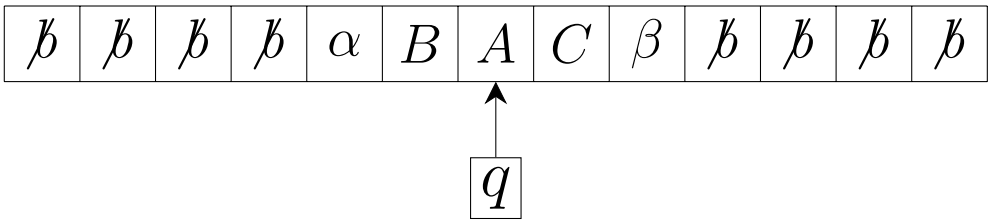

Figura 10: Configurazione della macchina di Turing in esame.

Per come abbiamo scritto le produzioni, i due stati in figura 11 sono in relazione se e solo se $\blacklozenge\alpha BqAC\beta \Rightarrow \blacklozenge\alpha BA'q'C\beta$. La costruzione della grammatica va, infine, completata aggiungendo le regole che cancellano tutto ciò che sta a destra del separatore $\blacklozenge$ (separatore incluso) se e solo se la configurazione della macchina di Turing è accettante, ad esempio $\blacklozenge\alpha Bq_fAC\beta$.

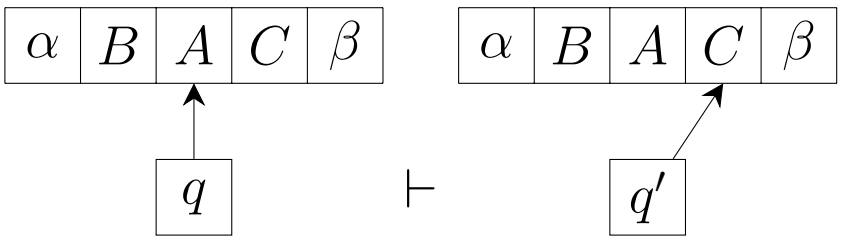

Figura 11: Due configurazioni della macchina di Turing in esame che sono legate dalla transizione.

Grammatiche monotone Possiamo notare che le produzioni si accorciano man mano che vengono "espanse". Quindi se costruiamo una macchina di Turing equivalente utilizzando i metodi descritti in questa sezione, essa userà al massimo la memoria occupata inizialmente dalla stringa in ingresso x. Possiamo, allora, limitare la potenza della nostra macchina fornendogli una memoria di dimensione finita, ottenendo così la macchina a minore potenza equivalente alle grammatiche monotone. Un automa così definito è detto automa lineare.

5 Altri modelli utili

Degli altri modelli molto usati per descrivere i linguaggi sono i pattern e le espressioni regolari. Non andremo per nulla del dettaglio e ci limiteremo a enunciare solo la definizione e al massimo qualche proprietà.

5.1 I pattern

Definizione 5.1 (Sistema di pattern)**.** Un sistema di pattern è una tripla $\langle A, V, p\rangle$ dove A è un alfabeto, V è un insieme di variabili disgiunto con A e p è una stringa su $A \cup V$ detta pattern.

5.2 Le espressioni regolari

Le espressioni regolari, o regex, le abbiamo già incontrate nella sezione 3.1 e definite con la definizione 3.3. Esse seguono la stessa idea generale dei pattern, ma hanno un potere espressivo molto diverso. Le regex hanno la stessa espressività dei linguaggi regolari e delle grammatiche regolari. Faremo solo una dimostrazione veloce di un solo senso dell'implicazione in quanto l'altro è più complesso e oggetto di altri corsi:

$\mathcal{L} \in \mathbf{RE} \implies \mathcal{L} \in \mathbf{REG}$. Guardando la definizione di espressione regolare si può notare che ciascuno dei casi base definisce un linguaggio regolare. Poiché le regex sono anche chiuse rispetto alle operazioni definite tutte le regex generalo linguaggi regolari. □

Con un semplice esempio si può anche vedere che la famiglia dei linguaggi descrivibili da una regex non coincide con quella dei linguaggi generati dai sistemi di pattern:

- $L = \{xx : x \in \{0,1\}\}$ non è un linguaggio regolare, ma può essere descritto con il pattern $\langle\{0,1\}, \{x\}, xx\rangle$;
- Il linguaggio generato da 0^*1^* è, ovviamente, regolare ma non è esprimibile con un pattern.

Quindi non è contenimento tra le due famiglie. Diciamo che esse sono non confrontabili.

6 Modelli dichiarativi

6.1 La logica

La logica è un elemento fondamentale dell'informatica. Esistono molti linguaggi logici, ognuno con diversi livelli di espressività e utilizzo. Nell'ambito di questo corso useremo la logica del primo ordine con due scopi: definizione di linguaggi e specifica della proprietà dei programmi. La logica, infatti, ci permetterà sia di definire i linguaggi specificandone le proprietà e di esprimere le condizioni che le nostre computazioni devono rispettare.

Esempio 6.1. Consideriamo il linguaggio $\{a^n b^n : n \geq 0\}$, come possiamo descriverlo con una formula del primo ordine? Una forma è:

$$\forall x(x \in L \iff \exists n(N \geq 0 \land a = a^n.b^n))$$

Definendo opportunamente i predicati $\in L$, $\geq$ e $=$ e le funzioni di concatenazione ed elevamento a potenza.

Esempio 6.2. Caratterizziamo un altro linguaggio: $L_1 = a^*b^*$. Uno strumento utile per caratterizzare un linguaggio con la logica è quello di pensare in modo induttivo. Ragionando induttivamente possiamo ricavare una forma logica corrispondente a L_1:

$$\forall x(x \in L_1 \iff (x = \epsilon) \lor \exists y(x = ay \land y \in L_1) \lor \exists y(x = yb \land y \in L_1))$$

Esempio 6.3. Consideriamo il linguaggio $L_2 = a^*b^*c^*$. Possiamo vederlo come $a^*b^*.b^*c^*$ dove entrambi questi sotto-linguaggi, chiamiamoli L_1 ed L_3, hanno struttura simile a L_1. Quindi una stringa appartiene ad L_2 se appartiene ad uno dei due linguaggi oppure se la prima sotto-stringa inizia con 'a' seguita da un suffisso y composto sia da 'a' e 'b' e la seconda sotto-stringa è composta da un prefisso y di 'b' e 'c' seguito da una c. Una formula del primo ordine che caratterizza la seguente è:

$$\forall x(x \in L_2 \iff x \in L_1 \vee x \in L_3 \vee \exists y(x = ay \wedge (y \in L_2 \vee y \in L_3)) \vee \exists y(x = yc \wedge (y \in L_2 \vee y \in L_3)))$$

Si può ridurre questa formula? Certo. Mantenendo la struttura come concatenazione possiamo scrivere:

$$\forall x(x \in L_2 \iff \exists z \exists y(y \in L_1 \wedge z \in L_3 \wedge x = y.z))$$

Esempio 6.4. Consideriamo l'ulteriore esempio del linguaggio:

$$L_4 = \{x \in \{a, b\} : \text{ numero di } a \text{ è uguale al numero di } b\}$$

Per indicare "il numero di" introduciamo la funzione di arietà 2 $\#(x, a)$ che indica il numero di occorrenze del carattere 'a' nella stringa x così formalmente definita:

$$\begin{aligned}
&\forall x \forall y((x = \epsilon \implies \#(x, a) = 0) \wedge \\
&\quad (x = a.y \implies \#(x, a) = \#(y, a) + 1) \wedge \\
&\quad (x = b.y \implies \#(x, a) = \#(y, a))) \wedge \\
&\quad \forall x \forall y((x = \epsilon \implies \#(x, b) = 0) \wedge \\
&\quad (x = b.y \implies \#(x, b) = \#(y, b) + 1) \wedge \\
&\quad (x = a.y \implies \#(x, b) = \#(y, b)))
\end{aligned}$$

Il nostro linguaggio in formula del primo ordine sarà $\forall x(x \in L_4 \iff \#(x, a) = \#(x, b))$.

6.1.1 Precondizioni e postcondizioni

Approfondiamo qui un uso della logica che ci interessa ai fini del corso: la specifica di condizioni che il nostri programma deve rispettare. Si preferisce specificare le condizioni iniziali e finali dell'ambiente invece di specificare il funzionamento specifico del programma perché di solito è più importante e flessibile definire correttamente cosa un programma faccia rispetto a definire una specifica implementazione. Infatti diverse implementazione possono rispettare le stesse precondizioni e postcondizioni, portando a programmi ottimizzati per diverse situazioni (uso minore di memoria, velocità di esecuzione maggiore, uso di particolari costrutti ecc...) che però sono funzionalmente equivalenti.

Per la specifica delle condizioni useremo la notazione di Hoare:

$$\begin{aligned}
&\{Precondizione\} \\
&Programma \\
&\{Postcondizione\}
\end{aligned} \tag{43}$$

Se valgono le precondizioni, affinché il programma sia considerato corretto dopo l'esecuzione dovranno valere le postcondizioni. Nella pratica la condizioni possono essere definite tramite linguaggio naturale nei commenti, funzioni di `assert` o linguaggi ad-hoc. Nel nostro contesto teorico le definiremo usando la logica del primo ordine arricchita dei necessari predicati.

Esempio 6.5. Sia un programma P che implementa la ricerca di un elemento x in un array ordinato di n elementi. La precondizione sarà che l'array sia ordinato e la postcondizione che la variabile logica `found` debba essere vera se e solo se l'elemento x esiste nell'array a. Notiamo che non ci interessa che tipo di algoritmo è implementato da P. Usando la notazione di Hoare, scriveremo:

$$\{\forall i(1 \leq i \leq n-1 \implies a[i] \leq a[i+1])\}$$
$$P$$
$$\{\texttt{found} \iff \exists i(1 \leq u \leq n \wedge a[i] = x)\}$$

Dove abbiamo definito "implicitamente" $a[i]$ l'i-esimo elemento di a.

6.2 La logica monadica del primo ordine

Consideriamo un frammento di logica del primo ordine che ci permette di descrivere parole su un alfabeto I. Chiamiamo questa logica "logica monadica del primo ordine" o MFO. La sintassi di questa logica è la seguente:

Definizione 6.1 (Sintassi della MFO)**.** Una formula ϕ fa parte della logica monadica del primo ordine se:

$$\phi \coloneqq a(x) \mid x < y \mid \neg\phi \mid \phi \wedge \phi \mid \forall x(\phi) \tag{44}$$

Con:

1. $a \in I$ e $a(x)$ un predicato per ogni singolo simbolo dell'alfabeto;
2. $<$ la relazione di minore;
3. $\mathbb{N}$ dominio delle variabili.

Data una parola $w \in I^+$ ed un simbolo $a \in I$, $a(x)$ è vera se e solo se l'x-esimo simbolo di w è a (il primo simbolo di w ha indice 0). Per esempio una formula che è vera su tutte e solo le parole il cui primo simbolo esiste ed è 'a' è: $\exists x(x = 0 \wedge a(x))$. Considereremo solo parole non vuote nelle definizioni poiché alleggerisce molto la notazione. I concetti possono, però, essere estesi anche a parole vuote.

Il resto dei predicati consueti viene definito con:

$$\phi_1 \vee \phi_2 \triangleq \neg(\neg\phi_1 \wedge \neg\phi_2) \tag{45}$$
$$\phi_1 \implies \phi_2 \triangleq \neg\phi_1 \vee \phi_2 \tag{46}$$
$$\exists x(\phi) \triangleq \neg\forall x(\neg\phi) \tag{47}$$
$$x = y \triangleq \neg(x < y) \wedge \neg(y < x) \tag{48}$$
$$x \leq y \triangleq \neg(y < x) \tag{49}$$

Possiamo anche definire altri predicati utili:

- La costante 0 tale che $x = 0 \triangleq \forall y(\neg(y < x))$;
- Il predicato "y successore di x" $\text{succ}(x, y) \triangleq x < y \wedge \neq \exists z(x < z \wedge z < y)$;
- Le costanti $1, 2, 3, \ldots$ come i successori di $0, 1, 2, \ldots$.

Per comodità possiamo definire altre abbreviazioni del tipo $y = x + 1$ per indicare $\text{succ}(x, y)$, $y = x + k$ per indicare lo spiazzamento di k, $y = x - 1$ per indicare $\text{succ}(y, x)$, analogamente $y = x - k$ e $\text{last}(x)$ per indicare l'ultima posizione.

Per definire la semantica introduciamo la definizione di assegnamento.

Definizione 6.2 (Assegnamento)**.** Siano $w \in I^+$ e V_1 l'insieme delle variabili. Un assegnamento è una funzione $\nu_1 : V_1 \to \{0, 1, \ldots, |w| - 1\}$ tale che:

- $w, \nu_1 \models a(x)$ se e solo se $w = uav$ e $|u| = \nu_1(x)$;

- $w, \nu_1 \models x < y$ se e solo se $\nu_1(x) < \nu_1(y)$;
- $w, \nu_1 \models \neg\phi$ se e solo se non $w\nu_1 \models \phi$;
- $w, \nu_1 \models \phi_1 \wedge \phi_2$ se e solo se $w, \nu_1 \models \phi_1$ e $w, \nu_1 \models \phi_2$;
- $w, \nu_1 \models a(x)$ se e solo se $w, \nu_1' \models \phi$ per ogni v_1' con $\nu_1'(y) = \nu_1(y)$, $y \neq x$.

Possiamo così definire il linguaggio corrispondente ad una formula chiusa ϕ come:

$$L(\phi) = \{w \in I^+ : \exists \nu_1 (w, \nu_1 \models \phi)\} \tag{50}$$

6.2.1 Proprietà

I linguaggi esprimibili tramite MFO sono chiusi rispetto a unione, intersezione e complemento come potrebbe far intendere la buona definizione di $\wedge$, $\vee$ e $\neg$. Essi però non sono chiusi rispetto alla star di Kleene. I linguaggi esprimibili da MFO sono perciò detti "star-free".

Un altra peculiarità delle formule MFO è il fatto che siano meno potenti degli FSA. Essi infatti non possono rappresentare un particolare linguaggio regolare L_p fatto di tutte e sole le parole di lunghezza pari su un alfabeto di una singola lettera. Quindi possiamo dire che i linguaggi "star-free" sono un sottoinsieme dei linguaggi regolari.

Enunciamo il seguente teorema che caratterizza i linguaggi star-free su alfabeto di una lettera.

Teorema 6.1 (Espressività della MFO)**.** *Ogni linguaggio $\mathcal{L}$ con alfabeto di una lettera è esprimibile tramite MFO (è star-free) se e solo se è finito o cofinito.*

Nota. Un linguaggio si dice cofinito se il suo complementare è finito.

6.3 Logica monadica del secondo ordine

Per permettere alla logica MFO di avere lo stesso potere espressivo degli FSA bisogna aumentarne la potenza. Un modo per fare ciò è permettere di quantificare i predicati monadici. Ammettiamo quindi formule del tipo $\exists X(\phi)$ dove X è una variabile il cui dominio è l'insieme dei predicati monadici.

In questo caso la semantica prevede anche un secondo assegnamento $\nu_2 : V_2 \to \wp(\{0, 1, \ldots, |w| - 1\})$ con le seguenti regole aggiuntive:

- $w, \nu_1, \nu_2 \models X(x)$ se solo se $\nu_2(x) \in \nu_2(X)$;
- $w, \nu_1, \nu_2 \models \exists X(\phi)$ se e solo se $w, \nu_1, \nu_2' \models \phi$ per qualche ν_2' con $\nu_2'(Y) = \nu_2(Y)$, $Y \neq X$.

Esempio 6.6. Possiamo allora descrivere il linguaggio L_p visto precedentemente (perdonate la lisp-syntax ma la formula era troppo lunga):

$$\begin{aligned}
&\exists P(\\
&\quad \forall x(a(x) \wedge \neg P(0) \wedge \\
&\qquad \forall y(y = x + 1 \implies (\neg P(x) \iff P(y))) \wedge \\
&\qquad (\text{last}(x) \implies P(x))))
\end{aligned}$$

Dove P indica un insieme di posizioni dispari.

6.3.1 Trasformare da MSO a FSA

In generale, grazie alle quantificazioni del secondo ordine, per ogni FSA, è possibile scrivere una formula MSO equivalente. In modo non rigoroso possiamo procedere così. Innanzitutto ogni predicato monadico quantificato corrisponde a ciascuno stato. Se nell'automa non è possibile essere in diversi stati contemporaneamente (determinismo), mettiamo i vari stati in esclusione mutua. Infine codifichiamo tutte le transizioni usando $last(x)$ e $\neg last(x)$ per caratterizzare gli stati finali e non finali rispettivamente.

Si può eseguire anche la trasformazione inversa tramite un processo ben definito, il teorema di Büchi-Elgot-Trakhtenbrot. Noi non lo vedremo poiché è un teorema molto tecnico.

Le due trasformazioni descritte ci premettono, quindi, di sancire l'equivalenza tra FSA e MSO.

7 Teoria della computazione

Automi, grammatiche e altri formalismi possono essere considerati modelli meccanici per risolvere problemi matematici. Alcuni di questi sono più potenti di altri, ad esempio le TM riconosco una famiglia di linguaggi che contiene linguaggi non riconoscibili dai PDA. Abbiamo inoltre affermato che nessun formalismo sarà più potente di una macchina di Turing. Questi modelli possono, però, catturare l'essenza di un generico solutore meccanico? Inoltre se un problema è stato formalizzato adeguatamente, possiamo sempre risolverlo mediante dispositivi meccanici?

7.1 Formalizzazione di problema

Prima di parlare di risoluzione e risolvibilità, dobbiamo definire bene il concetto di problema. Nei precedenti capitoli abbiamo già incontrato la formalizzazione di un problema come riconoscimento ($x \in L$) o traduzione ($y = \tau(x)$) di stringhe su un alfabeto. Con questi due formalismi è possibile descrivere tutti problemi con dominio numerabile. Infatti un problema con dominio numerabile può essere sempre ricondotto al calcolo di una funzione $f : \mathbb{N} \to \mathbb{N}$. Si può inoltre dimostrare che riconoscimento e traduzione possono ridotte l'una all'altra:

Teorema 7.1 (Equivalenza tra traduzione e riconoscimento)**.** *Se una macchina può risolvere tutti i problemi del tipo $y = \tau(x)$ allora può risolvere anche i problemi del tipo $x \in L$ e viceversa.*

Dimostrazione teorema 7.1. Dimostriamo i due versi dell'implicazione separatamente:

$\Longrightarrow$ Se ho una macchina che può risolvere tutti i problemi della forma $y = \tau(x)$ e voglio usarla per risolvere il problema $x \in L$ è sufficiente definire $\tau(x) = 1$ se $x \in L$ e 0 altrimenti.

$\Longleftarrow$ Se ho una macchina che può risolvere tutti i problemi della forma $x \in L$, posso definire un linguaggio $L_\tau = \{x \blacklozenge y : y = \tau(x)\}$. Per una x fissata, posso enumerare tutte le possibili stringhe y sull'alfabeto di uscita e per ognuna di esse posso chiedere alla macchina se $x \blacklozenge y \in L_\tau$.

□

La classe dei problemi che possono essere risolti da una TM è indipendente dall'alfabeto scelto, sempre che ci siano almeno due simboli. Prima abbiamo detto che le macchine di Turing sono il più potente formalismo di calcolo automatico. Questa è la cosiddetta tesi di Church. Esso non è un teorema poiché andrebbe verificato ogni volta che qualcuno inventa un nuovo modello computazionale. Nonostante ciò è largamente accettato dagli studiosi come teorema.

Proposizione 7.1 (Tesi di Church)**.** *Se un problema è umanamente calcolabile, allora esisterà una macchina di Turing in grado di risolverlo.*

Tutti i modelli che abbiamo esaminato sono discreti, in accordo con la tecnologia digitale nel quale vengono impiegati. Inoltre data una TM, si può costruire un programma in Java, C o FORTRAN che la simula e viceversa. Questi linguaggi di programmazione hanno, quindi, la stessa espressività delle macchine di Turing e vengono detti "Turing-completi". Esistono anche dei linguaggi che non sono Turing-completi come ad esempio il Datalog.

7.2 Gli algoritmi

Introduciamo un altro concetto fondamentale per l'informatica: l'algoritmo.

Definizione 7.1 (Algoritmo (intuitivo))**.** Con algoritmo intendiamo una procedura per risolvere problemi mediante un dispositivo di calcolo automatico.

Gli algoritmi godono di alcune importanti proprietà, anche queste enunciate in modo informale.

Proposizione 7.2. *Sia A un algoritmo in esecuzione su una macchina $\mathcal{M}$ dotata di un processore meccanico (digitale) e di una memoria utilizzabile dal processore in modo arbitrario. Si ha che:*

1. *La sequenza di istruzioni di A dev'essere finita;*
2. *Qualunque istruzione di A dev'essere eseguibile dal processore meccanico di $\mathcal{M}$;*
3. *La computazione è discreta, ossia l'informazione è codificata digitalmente e la computazione procede attraverso passi discreti;*
4. *A è eseguito in modo deterministico;*
5. *Non c'è limite sulla quantità di dati in ingresso e in uscita;*
6. *Non c'è limite sulla quantità di memoria richiesta per effettuare una computazione;*
7. *Non c'è limite sul numero di passi discreti richiesti per effettuare una computazione.*

Possiamo descrivere ogni problema calcolabile tramite un sistema di calcolo automatico con un algoritmo e quindi, per la tesi di Church, ogni algoritmo potrà essere eseguito da una macchina di Turing. Possiamo allora riformulare la tesi di Church come:

Proposizione 7.3 (Tesi di Church)**.** *Ogni algoritmo si può codificare con un macchina di Turing.*

Quali sono però i problemi che possiamo risolvere tramite un algoritmo? Una risposta un po' banale è: quelli risolvibili dalle macchine di Turing.

7.3 Enumerazione algoritmica

Definizione 7.2 (Insieme enumerabile algoritmicamente)**.** Un insieme S può essere enumerato algoritmicamente se possiamo trovare una biezione E tra S $\mathbb{N}$ calcolabile con un algoritmo.

Dimostriamo che l'insieme delle macchine di Turing è un insieme enumerabile algoritmicamente:

Proposizione 7.4 (Enumerazione delle TM)**.** *L'insieme di tutte le macchine di Turing a nastro singolo con alfabeto $A = \{0, 1, \not b\}$ è un insieme enumerabile algoritmicamente.*

Dimostrazione. Consideriamo le TM con solo due stati. Iniziamo con il calcolare la cardinalità dell'insieme di tutte le possibili funzioni di transizione realizzabili. Sappiamo che δ ha forma:

$$\delta : Q \times A \to Q \times A \times \{R, L, S\} \cup \{\bot\}$$

Sfruttando un risultato della teoria insiemistica che ci dice che il numero di funzioni $f : D \to R$ è $|R|^{|D|}$, otteniamo che il numero di funzioni sarà:

$$[|Q| * |A| * (3 + 1)]^{2*3} = 19^6$$

Considerando che abbiamo 4 possibili scelte di stati finali, otteniamo un numero di $4 * 19^6$ macchine di Turing a 2 stati. Iterando il procedimento, otterremo sempre un numero finito e al più numerabile di TM. Ora ci basta ordinare secondo un ordine arbitrario e quindi scrivere un algoritmo che le enumeri. □

Il numero $E(M)$ è detto numero di Gödel della TM $\mathcal{M}$ ed indica il suo indice nella enumerazione algoritmica di tutte le TM creabili. La E è detta la gödelizzazione dell'insieme delle TM.

7.4 Macchina di Turing universale

Chiediamoci ora se le macchine di Turing riescano a modellare i calcolatori programmabili. Una TM con queste caratteristiche è detta macchina di Turing universale (UTM). La UTM computa la funzione $g(y, x) = f_y(x)$ dove f_y è la funzione calcolata dalla TM di indice y.

Per come è definita, sembrerebbe che la UTM sia un diverso tipo di automa rispetto alla TM: la prima lavora $\mathbb{N}^2$ mentre la seconda si $\mathbb{N}$. Sappiamo però che la cardinalità di $\mathbb{N}^2$ è uguale a quella di $\mathbb{N}$ in quanto esiste la biezione:

$$d(x, y) = \frac{(x + y)(x + y + 1)}{2} + x \tag{51}$$

Possiamo quindi esprimere $g(y, x)$ come $\hat{g}(n) = g(d^{-1}(n))$ con $n = d(y, x)$.
Una UTM che calcola $\hat{g}$ opererà in questo modo:

1. Dato n calcola $\langle y, x\rangle = d^{-1}(n)$;
2. Costruisce la funzione di transizione M_y calcolando E^{-1} e la memorizza sul nastro;
3. In un'altra porzione di nastro memorizza una codifica della configurazione di M_y;
4. Lascia sul nastro $f_y(x)$ se e solo se $M_y(x)$ termina la computazione.

7.5 Funzioni calcolabili e problemi definibili

È possibile calcolare tutte le funzioni da $\mathbb{N}$ a $\mathbb{N}$? In caso negativo, per la tesi di Church, esisterebbero problemi non risolvibili tramite algoritmi. Calcoliamo la cardinalità dell'insieme di funzioni naturali a variabile naturale. Consideriamo il sottoinsieme di $f : \mathbb{N} \to \{0, 1\}$. È possibile calcolare la cardinalità di questo insieme ed essa è pari a quella di $\wp(\mathbb{N})$ ossia $2^{\aleph_0}$ ossia la cardinalità del continuo. Visto che abbiamo considerato un sottoinsieme, la cardinalità che cerchiamo è sicuramente maggiore di quella del continuo. Poiché l'insieme di funzioni calcolabili dalle macchine di Turing è per definizione numerabile (di cardinalità $\aleph_0$) non abbiamo abbastanza TM per risolvere tutti i problemi possibili.

Precedentemente abbiamo detto che i problemi di dominio naturale sono sempre traducibili come la computazioni di una funzione di dominio naturale. Abbiamo però visto che non tutte le funzioni di questo tipo sono computabili. Per definire un problema, infatti, ci serve una frase, una stringa, di qualche linguaggio che li caratterizzi. Per definizione, ogni linguaggio è un insieme numerabile e ciò renderà anche l'insieme dei problemi definibili numerabile. Quindi esistono problemi che non sono nemmeno definibili.

Concentriamoci sui problemi che sono definibili. Essi sono sempre risolvibili? La risposta è negativa. Un famoso problema definibile ma non risolvibile è il "Halting Problem".

Proposizione 7.5 (Halting problem)**.** *Costruito un programma che, dati dei dati in ingresso, esegue una computazione che potrebbe terminare, è possibile determinare in anticipo se terminerà?*

Dimostrazione. Formalmente ci stiamo cheidendo se esiste una TM tale che data f_y, calcola $g(y, x)$ totale tale che:

$$g(y, x) = \begin{cases} 1 & \text{se } f_y(x) \neq \perp \\ 0 & \text{se } f_y(x) = \perp \end{cases}$$

Esistono diversi metodi di dimostrazione. Noi useremo quello per diagonalizzazione a quello usato da Cantor per dimostrare la cardinalità del continuo.

Enumeriamo tutte le funzione calcolabili da una macchina di Turing e i loro valori:

	1	2	...
f_1	$f_1(1)$	$f_1(1)$	...
f_2	$f_2(1)$	$f_2(1)$	...
f_3	$f_3(1)$	$f_3(1)$	...
$\vdots$	$\vdots$	$\vdots$	$\ddots$

Ipotizziamo allora per assurdo che la nostra funzione g sia calcolabile. Definiamo

$$h(x) = \begin{cases} 1 & \text{se } g(x,x) = 0 \\ \perp & \text{altrimenti} \end{cases}$$

La funzione h è ovviamente calcolabile se anche g lo sarà. Abbiamo quindi definito una funzione che si pone sulla diagonale della nostra tavola delle funzioni.

Se h è calcolabile, e lo è visto che per ipotesi anche g è calcolabile, esisterà una TM di indice i tale che $h = f_i$. Provando a calcolare $h(i)$ otteniamo:

1. Se $h(i) = f_i(i) = 1$ allora $g(i,i) = 0$, ossia $f_i(i) = \perp$ che è una contraddizione
2. Se $h(i) = f_i(i) = 0$ allora $g(i,i) = 1$, ossia $f_i(i) \neq \perp$ che è una contraddizione

Siamo arrivati così ad un assurdo e quindi g non è calcolabile. □

L'Halting Problem ci permette di affermare con fermezza che l'insieme dei problemi risolvibili è strettamente incluso in quello dei problemi descrivibili.

7.5.1 Specializzazioni e generalizzazioni

Consideriamo la funzione:

$$h'(x) = \begin{cases} 1 & \text{se } f_y(x) = \perp \\ 0 & \text{altrimenti} \end{cases} \tag{52}$$

Essa è un caso particolare del Halting Problem dove abbiamo imposto che in $g(y,x)$ sarà $y = x$. Si dimostra che h' non è computabile. Nota bene che h' non è un corollario e non deriva dal Halting problem. Infatti un caso specifico di un problema non risolvibile non è detto che sia anch'esso non risolvibile e, viceversa, la generalizzazione di un problema risolvibile non per forza mantiene la risolvibilità. Se un problema è già risolvibile, però, la sua specializzazione sarà anch'essa risolvibile e viceversa la generalizzazione di un problema non risolvibile sarà ancora non risolvibile.

7.5.2 Il problema delle funzioni totali

Consideriamo un'altra funzione:

$$k(y) = \begin{cases} 1 & \text{se } \forall x \in \mathbb{N}, f_y(x) \neq \perp \\ 0 & \text{altrimenti} \end{cases} \tag{53}$$

La funzione, in pratica, vale 1 se la funzione $f_y(x)$ è totale e 0 altrimenti. Da un punto di vista pratico, questa funzione è ancora più importante di quella derivata dal problema dell'arresto. In questo caso, infatti, ci chiediamo se il programma non termini per qualsiasi ingresso, invece di chiedercelo solo per uno. Dimostriamo che la funzione k non è computabile.

Proposizione 7.6. *La funzione 53 non è computabile.*

Dimostrazione. Ipotizziamo per assurdo che la funzione sia computabile. Definiamo $g(x) = w$ con w pari al numero di Gödel della x-esima TM che calcola una funzione totale. Se k è computabile, allora lo sarà ang g. Infatti possiamo trovare una procedura algoritmica che calcola $g(x)$:

1. Calcoliamo $k(x)$ al crescere di x. Trovato $x_0|_{k(x_0)=1}$ poniamo $g(0) = x_0$ e riprendiamo a calcolare k da $x_0 + 1$ in avanti;
2. Trovato $x_1|_{k(x_1)=1}$ poniamo $g(1) = x_1$ e iteriamo.

È facile vedere che g è strettamente monotona. Possiamo calcolare g^{-1} anch'essa strettamente monotona ma non totale. Definiamo

$$f_{g(x)}(x) + 1 = f_w(x) + 1$$

Sappiamo che $f_w(x)$ è calcolabile e totale per definizione di g e quindi anche h lo sarà. Esisterà allora un $\bar{w}|_{f_{\bar{w}}(\cdot)=h(\cdot)}$ Dato che h è totale, sicuramente anche $g^{-1} \neq \perp$. Poniamo allora $g^{-1}(\bar{w}) = \bar{x}$. Studiamo la forma di h: per definizione avremo

$$h(\bar{x}) = f_{g(\bar{x})}(\bar{x}) + 1 = f_{\bar{w}}(\bar{x}) + 1$$

Ma, siccome $h(\cdot) = f_{\bar{w}}(\cdot)$, abbiamo anche $h(\bar{x}) = f_{\bar{w}}(\bar{x})$. Cadiamo allora in un assurdo e quindi la funzione k non è calcolabile. □

Nota bene che la definizione di k ha una quantificazione universale. Potrebbe essere, quindi, che per qualche valore specifico di x possiamo stabilire il valore di $k(y)$. Non siamo però in grado di dire se possiamo farlo per ogni singolo ingresso possibile.

7.6 Decidibilità e semidecidibilità

Concentriamoci sui problemi con risposta binaria, detti anche di decisione. Questo tipo di problemi hanno due risposte possibili "vero" e "falso". Un problema di questo tipo può essere decidibile, semidecidibile o indecidibile. La prima e la ultima hanno significato ovvio, approfondiamo la seconda.

Definizione 7.3 (Problema semidecidibile)**.** Un problema si dice semidecidibile se esiste un algoritmo che ritorni "vero" se il problema ha risposta affermativa.

Nota. La definizione di semidecidibilità non ci impone limiti su quello che fa l'algoritmo in caso di esito negativo. Esso può ritornare il valore "falso" oppure anche non terminare.

Semidecidibilità e testing Un gran numero di problemi indecidibili si può dimostrare che sono semidecidibili. Uno di questi è il problema dell'arresto: l'algoritmo basta che guardi se la TM si ferma su uno stato di accettazione. L'utilità della proprietà di semidecidibilità è che ci permette di rilevare se c'è un errore invece di garantirne la sua assenza. Ciò ha importanti applicazioni pratiche nella verifica di programmi basata sul testing:

Lemma 7.1 (Asserzione di Dijkstra)**.** *Il testing può dimostrare la presenza di errori, non la loro assenza.*

Tutti i problemi di decisione possono essere riformulati come "dato un insieme S, $x \in S$". Alternativamente possiamo calcolare la funzione caratteristica dell'insieme S:

$$\mathbf{1}_S = \begin{cases} 1 & \text{se } x \in S \\ 0 & \text{altrimenti} \end{cases} \tag{54}$$

L'insieme S può essere ricorsivo (decidibile) oppure ricorsivamente numerabile (semidecidibile).

Definizione 7.4 (Insieme ricorsivo (decidibile))**.** Un insieme S si dice ricorsivo o decidibile se e solo se la sua funzione caratteristica è computabile.

Definizione 7.5 (Insieme ricorsivamente enumerabile (semidecidibile))**.** Un insieme S si dice ricorsivamente enumerabile (RE) o semidecidibile se e solo se:

- S è l'insieme vuoto;
- S è l'immagine di una funzione totale computabile:

$$S = I_{g_S} = \{x : g_S(y), y \in \mathbb{N}\} \tag{55}$$

Il termine di insieme semidecidibile deriva dal fatto che se $x \in S$ enumerando gli elementi di S prima o poi lo troverò, altrimenti sono mai certo di poter rispondere "falso" enumerando in quanto potrei non aver ancora trovato x.

Tra la decidibilità e la semidecidibilità ci sono diversi importanti legami:

Teorema 7.2 (Decidibilità implica semidecidibilità)**.** *Se un insieme S è ricorsivo (decidibile), esso è ricorsivamente enumerabile (semidecidibile).*

Dimostrazione. Studiamo i vari casi possibili nella definizione di semidecidibilità:

- Se S è vuoto, è RE per definizione;
- Se S non è vuoto, costruiamo una funzione totale e computabile di cui S è immagine. Poiché sappiamo che esisterà un $k \in S$ tale che $\mathbf{1}_S(k) = 1$, possiamo definire la funzione come

$$g_S(x) = \begin{cases} x & \text{se } \mathbf{1}_S(x) = 1 \\ k & \text{se } \mathbf{1}_S(x) = 0 \end{cases}$$

  Con g_S totale, computabile e $\mathbf{I}_{g_S} = S$. S sarà, quindi, RE.

□

Teorema 7.3 (semidecidibilità + semidecidibilità = decidibilità)**.** *Si un insieme S, esso sarà ricorsivo se e solo se sono ricorsivamente enumerabili sia S che $\bar{S} = \mathbb{N} \setminus S$.*

Dimostrazione. Dimostriamo che se S è ricorsivo allora S e $\bar{S}$ sono RE. Innanzitutto S è ricorsivo, per il teorema 7.2 esso è anche RE. Inoltre possiamo osservare che anche $\mathbf{1}_{\bar{S}}$ è calcolabile in quanto è pari a $\mathbf{1}_S$ con 1 e 0 scambiati, permettendoci di affermare che anche $\bar{S}$ è RE.

Dimostriamo ora l'implicazione inversa. Osserviamo che $S \cup \bar{S} = \mathbb{N}$ e $S \cap \bar{S} = \emptyset$ e quindi una qualunque $x \in \mathbb{N}$ appartiene a una e una sola delle due enumerazioni di S e $\bar{S}$:

$$(\forall x \in \mathbb{N}, \exists y : x = g_{\bar{S}}(y) \vee x = g_S(y)) \wedge (\neg \exists z : g_{\bar{S}}(z) = g_S(z))$$

Posso essere, quindi, certo di trovare qualsiasi x nell'enumerazione

$$\{g_S(0), g_{\bar{S}}(0), g_S(1), g_{\bar{S}}(1), \ldots\}$$

Nel momento in cui trovo x in posizione dispari concludo che $x \notin S$, altrimenti $x \in S$. So quindi calcolare $\mathbf{1}_S$, rendendo S ricorsivo. □

Da questi due teoremi possiamo trarre un po' di risvolti importanti. Innanzitutto possiamo affermare che gli insiemi decidibili sono chiusi rispetto al complemento. Possiamo usare la teoria degli insiemi ricorsivi per cercare di capire se possiamo enumerare tutte le funzioni calcolabili e totali.

Proposizione 7.7 (Definione delle funzioni calcolabili e totali)**.** *Dato un insieme S per cui:*

1. *Se $i \in S$ allora f_i e calcolabile e totale;*
2. *Se f e totale e computabile allora $\exists i \in S : f_i = f_i$.*

Esso non sarà ricorsivamente enumerabile.

Dimostrazione. Ipotizziamo per assurdo che esista una funzione caratteristica $\mathbf{1}_S(i)$ computabile. Definiamo:

$$h(x) = \begin{cases} f_x(x) + 1 & \text{se } \mathbf{1}_S(x) = 1 \\ 0 & \text{altrimenti} \end{cases}$$

Abbiamo che per ogni x $h(x) \neq f_{\mathbf{1}_S(x)}(x)$. Quindi $h(x)$ è calcolabile ma diversa da tutte le funzioni calcolabili, il che è un assurdo. □

Non è possibile quindi definire con un formalismo RE (automi, grammatiche e funzioni ricorsive) capace di definire l'insieme dei tutte e sole le funzioni calcolabili totali. Quindi non posso descrivere in nessun modo descrivere come è fatto l'insieme di tutti e soli i programmi che terminano sempre:

- Gli FSA e i PDA definiscono solo funzioni totali ma non tutte
- Le TM definiscono tutte le funzioni calcolabili, ma anche quelle non totali
- Il C mi permette di scrivere qualunque algoritmo, ma anche che non terminato

Riusciamo, però, ad aggirare il problema e considerare l'insieme di tutte le funzioni e rimuovere con un artificio le funzioni parziali? Arricchiamo $\mathbb{N}$ con un nuovo valore $\perp$ oppure assegnamo un calore convenzionale ad f quando non è definita. Matematicamente questa operazione non causa problemi, però si può dimostrare che per la funzione:

$$g(x) = \begin{cases} f_x(x) + 1 & \text{se } f_x(x) \neq \perp \\ \perp & \text{altrimenti} \end{cases} \tag{56}$$

Non è possibile costruire una funzione computabile e totale che la estenda. Inoltre possiamo dimostrare che esistono insiemi che sono semidecidibili senza essere decidibili.

Dimostrazione. Esistenza di insiemi semidecidibili non decidibili. Consideriamo

$$S = \{x : f_x(x) \neq \perp\}$$

Essa è il dominio della funzione $h(x) = f_x(x)$ che è computabile ma parziale. Poiché è possibile riscrivere questa affermazione in termini di immagine di una g totale computabile, S è RE. Sappiamo anche che la funzione caratteristica $\mathbf{1}_S = 1$ se $f_x(x) \neq \perp$, e 0 altrimenti non è computabile e quindi S non è decidibile. □

Possiamo costruire quindi un gerarchia di inclusioni strette tra vari tipi di insiemi:

$$\text{Ricorsivi} \subset \text{RE} \subset \wp(\mathbb{N}) \tag{57}$$

7.7 Stabilire decidibilità e semidecidibilità di un problema

Enunciamo il primo risultato importante che ci servirà per capire se e come possiamo stabilire la decidibilità (semidecidibilità) di un problema.

Teorema 7.4 (Teorema di Kleene del punto fisso)**.** *Sia una funzione $t(\cdot)$ totale e computabile. È sempre possibile trovare un $p \in \mathbb{N}$ tale che $f_p = f_{t(p)}$. La funzione f_p è detta punto fisso di $t(\cdot)$.*

Dimostrazione. Dato $u \in \mathbb{N}$ definiamo una TM che effettua il seguente calcolo sull'ingresso x: calcola $f_u(u) = z$ e quando il calcolo termina calcola $f_z(x)$. Possiamo costruire questa TM e cercare il suo numero di Gödel $g(u)$ per una qualsiasi u. Otteniamo che la funzione della TM sarà esprimibile come:

$$f_{g(u)}(x) = \begin{cases} f_{f_u(u)}(x) & \text{se } f_u(u) \neq \perp \\ \perp & \text{altrimenti} \end{cases}$$

Sappiamo che data $g(\cdot)$ totale e calcolabile e $t(\cdot)$ anch'essa totale e computabile, lo sarà anche $t(g(\cdot))$. Chiamiamo v il numero di Gödel di $t(g(\cdot))$, ottenendo $t(g(\cdot)) = f_v(\cdot)$. Ripetendo la costruzione che abbiamo fatto precedentemente otteniamo che:

$$f_{g(v)} = \begin{cases} f_{f_v(v)}(x) & \text{se } f_v(v) \neq \perp \\ \perp & \text{altrimenti} \end{cases}$$

Ricordando che $t(g(\cdot))$ è totale e computabile, otteniamo che $f_{g(v)}(x) = f_{f_v(v)}(x)$ per ogni x. Sostituendo nel secondo membro otteniamo:

$$f_{g(v)} = f_{t(g(v))}$$

Rendendo, quindi $f_{g(v)}$ il punto fisso di $t(\cdot)$. □

Possiamo ora enunciare una condizione per la decidibilità di un insieme.

Teorema 7.5 (Teorema di Rice)**.** *Siano F un'insieme di funzioni computabili e l'insieme S degli indici delle TM che calcolano le funzioni di F. L'insieme $S = \{x : f_x \in F\}$ è decidibile se e solo se:*

- *$F = \emptyset$;*
- *F è l'insieme di tutte le funzioni computabili.*

Dimostrazione. Supponiamo per assurdo che S sia decidibile con F non vuoto e diverso dall'insieme di tutte le funzioni computabili. Consideriamo la funzione caratteristica di S $\mathbf{1}_S(x)$, essa sarà calcolabile per ipotesi. Possiamo quindi calcolare:

1. Il più piccolo $i \in \mathbb{N}$ tale che $f_i \in F$
2. Il più piccolo $j \in \mathbb{N}$ tale che $f_j \notin F$

Per quanto detto, la funzione:

$$h_S(x) = \begin{cases} i & \text{se } f_x \notin F \\ j & \text{altrimenti} \end{cases}$$

Sarà anch'essa calcolabile e totale. Applicando il teorema di Kleene alla funzione appena definita otteniamo che esiste un punto fisso $f_{\bar{x}}$ tale per cui $f_{\bar{x}} = f_{h_s(\bar{x})}$. Siamo così arrivati ad una contraddizione in quanto:

- Se $h_S(\bar{x}) = i$: per definizione di h_S abbiamo che $f_{\bar{x}} \notin F$, ma da quanto detto per il teorema di Kleene $f_{\bar{x}} = f_{h_S(\bar{x})} = f_i$ da cui, per come definito i, $f_i \in F$.
- Se $h_S(\bar{x}) = j$: per definizione di h_S abbiamo che $f_{\bar{x}} \in F$, ma da quanto detto per il teorema di Kleene $f_{\bar{x}} = f_{h_S(\bar{x})} = f_j$ da cui, per come definito j, $f_j \notin F$.

□

Quindi in tutti i casi non banali l'insieme delle funzioni calcolabili con una data caratteristica desiderata non è decidibile! Quindi non sarà possibile rispondere a molti quesiti importanti:

- Il programma P è corretto? Risolve un dato problema?
- È possibile stabilire l'equivalenza tra due programmi?
- È possibile stabilire se un generico programma gode di una qualsiasi proprietà non banale riferita alla funzione che calcola?

Stabilire se un generico problema è decidibile (semidecidibile) o meno è indecidibile. Ragionando in modo pratico abbiamo tre possibili alternative per stabilire la decidibilità di un problema:

1. Se troviamo un algoritmo che termina sempre, allora il problema è decidibile;
2. Se troviamo un algoritmo che termina sempre se la risposta è "vero" ma può non terminare se la risposta è "falso", allora il problema è semidecidibile;
3. Se riusciamo a trovare una dimostrazione diagonale della indecidibilità del problema allora esso è indecidibile.

La terza opzione è molto laboriosa ma fattibile. Il teorema di Rice ci permette facilmente di stabilire se un problema non è decidibile, ma esso può lo stesso essere semidecidibile. Una tecnica alternativa, molto generale, è quella della riduzione dei problemi: ci permette di dimostrare in modo agevole l'indecidibilità di alcuni problemi.

7.7.1 La tecnica di riduzione di un problema

Consideriamo prima una visione operativa della tecnica di riduzione. Se abbiamo un algoritmo per risolvere un problema P, possiamo riusarlo modificandolo per risolvere problemi P' simili a P. In generale se trovo un algoritmo che, dato un esemplare di P' ne costruisce la soluzione usando un esemplare di P che so risolvere, ho ridotto P' a P. Ciò significa che affinché P' sia riducibile a P, si dovrà avere che:

1. P risolvibile;
2. C'è un algoritmo che per ogni istanza di P' determina una corrispondente istanza di P e costruisce algoritmicamente la soluzione dell'istanza di P' usando la soluzione dell'istanza di P.

Formalizziamo il tutto:

Proposizione 7.8 (Tecnica di riduzione)**.** *Consideriamo due problemi: $y \in S'$ e $x \in S$ di cui il secondo è quello che vogliamo risolvere. Se troviamo un a funzione t calcolabile e totale per cui:*

$$x \in S \iff t(x) \in S' \tag{58}$$

Il problema d'interesse è risolvibile. Inoltre, dato x, calcolare $\mathbf{1}_{S'}(t(x))$ equivale a calcolare $\mathbf{1}_S(x)$.

La proposizione funziona anche in direzione inversa: se so che $y \in S'$ non è risolvibile, se trovo la t che rispetta la precedente condizione allora anche $x \in S$ non sarà risolvibile.

Esempio 7.1. Dall'indecidibilità del problema dell'arresto della TM deduciamo l'indecidibilità del problema della terminazione del calcolo in generale. I passaggi operativi per dimostrare ciò che possiamo eseguire sono:

1. Costruiamo un programma P che simuli una TM M_i e memorizziamo il numero x in un file f.
2. Il programma P termina la computazione su f se e solo se $g(i, x) \neq \perp$.
3. Se riuscissimo a scrivere un programma che riesce a predire il contenuto di f avremmo risolto il problema dell'arresto, ma ciò non è possibile.

Esempio 7.2. È decidibile dire se, durante l'esecuzione di un generico programma P si accede ad una variabile non inizializzata?

Supponiamo per assurdo che questo problema sia decidibile. Riduciamolo al problema dell'arresto:

1. Dato un generico programma $Q(n)$, costruisco un programma P fatto in questo modo:

```
{
  int x, y;
  Q(n);
  y = x;
}
```

   Avendo cura di usare variabili non presenti in `Q`.

2. L'accesso `y = x` alla variabile non inizializzata x da parte di P è fatto se e solo se `Q` termina.

Se fossi in grado di decidere il problema dell'accesso a variabile non inizializzata, potrei decidere il problema della terminazione del calcolo, che è un assurdo.

Le proprietà dimostrate nei precedenti esempi non sono decidibili, ma sono semidecidibili. Abbiamo un metodo operativo per determinare la semidecidibilità di un problema?

Innanzitutto il problema $\exists z : f_x(z) \neq \perp$ è semidecidibile. Una idea di dimostrazione è simulare l'azione della funzione "in diagonale" eseguendo una sola mossa alla volta per ogni input finché non troviamo l'input per la quale la computazione non si arresta. Simulando abbastanza passi di f_x, eventualmente lo troverò.

7.8 Problemi notoriamente decidibili o indecidibili

Forniamo ora una lista di problemi notoriamente indecidibili o decidibili.

Problemi notoriamente indecidibili Sono notoriamente indecidibili i seguenti problemi relativi alle grammatiche/linguaggi liberi dal contesto:

1. "Emptyness of intersection", ossia stabilire se l'intersezione tra due linguaggi sia vuota;
2. "Grammar class membership", ossia determinare se una data grammatica $\mathcal{G}$ appartenga ad un insieme di grammatiche Γ;
3. "Language class membership", ossia determinare se il linguaggio di una data grammatica $\mathcal{G}$ appartenga ad un insieme di linguaggi $\mathcal{L}$.

Per le macchine di Turing abbiamo invece che i seguenti due problemi sono notoriamente indecidibili, oltre i vari problemi discussi in precedenza:

1. "L'alacre castoro", ossia determinare la TM in un insieme di TM con lo stesso numero di stati che effettua il maggior numero di transizioni terminando;
2. Verificare se una TM accetti una stringa vuota.

Problemi notoriamente decidibili I seguenti problemi sono notoriamente decidibili:

1. Stabilire se un linguaggio regolare è vuoto;
2. Stabilire l'uguaglianza tra due linguaggi data la chiusura rispetto a complemento e intersezione e la decidibilità dello stabilire se un linguaggio è vuoto;
3. Equivalenza di polinomi.

Riassumiamo infine la decidibilità dei problemi legati ai vari tipi di linguaggi nella seguente tabella,

	$x \in \mathcal{L}$	$\mathcal{L} = \emptyset$	$\mathcal{L}_1 = \mathcal{L}_2$	$\mathcal{L}_1 \cap \mathcal{L}_2 = \emptyset$
Non ristretti	I	I	I	I
Liberi dal contesto	D	D	I	I
Liberi dal contesto deterministici	D	D	D	I
Regolari	D	D	D	D

Tabella 2: Problemi relativi ai linguaggi e la loro decidibilità

8 Complessità del calcolo

Calcolare la complessità di una computazione significa calcolare l'efficienza e il costo in tempo di una computazione. Come possiamo misurare questi parametri? Dobbiamo prima definire degli strumenti per valutare la complessità e successivamente potremmo enunciare degli algoritmi e strutture dati notevoli. Il nostro obiettivo sarà sviluppare gli strumenti per saper progettare e combinare algoritmi e strutture dati che realizzano soluzioni efficienti.

Per quantificare l'efficienza di un algoritmo, faremo analisi qualitative di due metriche:

- Tempo di calcolo impiegato;
- Spazio occupato (registri, cache, RAM, disco o nastro).

Si possono fare analisi anche su altri aspetti come i costi di sviluppo, ma noi non li considereremo. Per la tesi Church, un problema è calcolabile indipendentemente dallo strumento usato, purché tale strumento sia Turing completo. Possiamo dire lo stesso per la complessità? Purtroppo no. Dobbiamo, quindi, costruire uno strumento che tralasci considerazioni superflue e sia utilizzabile per la maggioranza dei modelli di calcolo. Noi considereremo per convenzione quello della macchina di Turing deterministica.

Formalizziamo le nostre due metriche:

Definizione 8.1 (Complessità temporale)**.** Data la computazione $c_1 \vdash^\star c_r$ di una macchina di Turing $\mathcal{M}$ (a k nastri) deterministica la complessità temporale è $T_\mathcal{M}(x) = r$ se $\mathcal{M}$ termina in c_r, ∞ altrimenti.

Definizione 8.2 (Complessità spaziale)**.** Data la computazione $c_1 \vdash^\star c_r$ di $\mathcal{M}$ (a k nastri) deterministica, la complessità spaziale è:

$$S_\mathcal{M}(x) = \sum_{j=1}^{k} \max_{i \in \{0,\dots,r\}} (|\alpha_{ij}|) \tag{59}$$

Con α_{ij} il contenuto del j-esimo nastro alla i-esima mossa.

Nota. Complessità spaziale e temporale sono legate dalla seguente relazione:

$$\forall x \quad \frac{S_\mathcal{M}(x)}{k} \leq T_\mathcal{M}(x) \tag{60}$$

Ad eseguire l'analisi su una TM, dobbiamo gestire un po' troppi dettagli. Effettuiamo, allora, delle semplificazioni: esprimeremo la complessità in base alla "dimensione" n dei dati in ingresso. A causa di questa semplificazione dobbiamo considerare 3 casi per gestire la variazione di ingressi diversi ma di stessa lunghezza:

Caso pessimo

$$T_\mathcal{M}(n) = \max_{|x|=n} T_\mathcal{M}(x) \tag{61}$$

Caso ottimo

$$T_\mathcal{M}(n) = \min_{|x|=n} T_\mathcal{M}(x) \tag{62}$$

Caso medio

$$T_\mathcal{M}(n) = \frac{\sum_{|x|=n} T_\mathcal{M}(x)}{|I|^n} \tag{63}$$

Noi considereremo sempre il caso pessimo in quanto è il più rilevante. Inoltre l'analisi del caso medio risulta assai complessa in quanto dovrebbe tenere conto di ipotesi probabilistiche sulla distribuzione dei dati.

I valori esatti delle due complessità per un dato n non sono particolarmente utili. La prima (e più forte) semplificazione che facciamo è quella di considerare solo il comportamento asintotico, ossia $n \to \infty$. Per esprimere il comportamento asintotico abbiamo 3 notazioni:

- $\mathcal{O}$-grande: limite asintotico superiore;
- Ω-grande: limite asintotico inferiore;
- Θ-grande: limite asintotico sia superiore che inferiore.

Eseguire una approssimazione di questo genere causa due problemi pratici. Il primo è che il comportamento asintotico potrebbe essere un pessimo modello per valori piccoli di n; il secondo è che in qualche raro caso la complessità per n piccolo non corrisponde alla nostra intuizione, ossia una algoritmo con complessità asintotica minore può essere più lento per valori piccoli di n rispetto ad uno con complessità asintotica maggiore.

Definizione 8.3 ($\mathcal{O}$-grande)**.** Data una funzione $g(n)$, $\mathcal{O}(g(n))$ è l'insieme:

$$\mathcal{O}(g(n)) = \{f(n) : \exists c > 0, n_0 > 0 \ (\forall n > n_0 \ 0 \leq f(n) \leq cg(n))\} \tag{64}$$

Definizione 8.4 (Ω-grande)**.** Data una funzione $g(n)$, $\Omega(g(n))$ è l'insieme:

$$\Omega(g(n)) = \{f(n) : \exists c > 0, n_0 > 0 \ (\forall n > n_0 \ 0 \leq cg(n) \leq f(n))\} \tag{65}$$

Definizione 8.5 (Θ-grande)**.** Data una funzione $g(n)$, $\Theta(g(n))$ è l'insieme:

$$\Theta(g(n)) = \{f(n) : \exists c_1 > 0, c_2 > 0 n_0 > 0 \ (\forall n > n_0 \ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n))\} \tag{66}$$

Proposizione 8.1 (Proprietà delle notazioni asintotiche)**.** *1.*

$$f(n) \in \Theta(g(n)) \iff f(n) \in \mathcal{O}(g(n)) \wedge f(n) \in \Omega(g(n))$$

2. Le tre notazioni godono della proprietà transitiva;

3. Le tre notazioni godono della proprietà riflessiva;

4. $f(n) \in \Theta(g(n)) \iff g(n) \in \Theta(f(n))$*;*

5. $f(n) \in \mathcal{O}(g(n)) \iff g(n) \in \Omega(f(n))$.

Nota. Per le proprietà enunciate sopra, la Θ è una relazione di equivalenza.

Noi abbiamo definito le notazioni asintotiche in termini insiemistici, ma possiamo anche definirle come limiti:

Definizione 8.6 (Θ-grande (come limite))**.** Diciamo che $f(n) \in \Theta(g(n))$ se e solo se:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c, c \neq 0, c \neq \infty \tag{67}$$

Definizione 8.7 ($\mathcal{O}$-grande (come limite))**.** Diciamo che $f(n) \in \mathcal{O}(g(n))$ se e solo se:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \tag{68}$$

Possiamo anche scrivere $\Theta(f(n)) < \Theta(g(n))$

8.1 Complessità degli altri modelli visti fino ad ora

Analizziamo velocemente la complessità degli altri modelli di calcolo introdotti fino ad ora: gli FSA, i PDA e le TM a nastro singolo.

FSA Gli FSA avranno sempre complessità spaziale $S_{FSA}(n) \in \Theta(1)$ (complessità costante), in quanto privi di memoria, e complessità temporale $T_{FSA}(n) \in \Theta(n)$ poiché leggono l'input in sequenza, un carattere alla volta.

PDA A causa di come funziona la stack, la complessità spaziale sarà $S_{PDA}(n) \in \Theta(n)$, mentre quella temporale sarà, come nel caso degli FSA, $T_{PDA}(n) \in \Theta(n)$.

TM a nastro singolo La complessità spaziale non potrà mai scendere sotto la linearità ($S_{\mathcal{M}}(n) \in \Theta(n)$) mentre si dimostra che le TM a nastro singolo non possono accettare $\mathcal{L} = \{wcw^R | w \in \{a, b\}^\star\}$ con $\Theta(T_{\mathcal{M}}(n)) < \Theta(n^2)$. Le TM a nastro singolo sono, quindi, più potenti dei PDA ma anche più lente.

8.2 Teoremi di accelerazione lineare

Enunciamo ora dei teoremi che ci permettono di migliorare con facilità, sebbene al massimo linearmente la complessità dei nostri algoritmi.

Teorema 8.1 (Primo teorema di accelerazione lineare)**.** *Se L è il linguaggio accettato da una TM $\mathcal{M}$ a k nastri in $S_{\mathcal{M}}(n)$, per ogni $c \in \mathbb{R}^+$ posso costruire una TM $\mathcal{M}'$ a k nastri che accetta L con:*

$$S_{\mathcal{M}'}(n) < c \cdot S_{\mathcal{M}}(n) \tag{69}$$

Il trucco usato da questo teorema, e anche degli altri teoremi a seguire, sta nel aumentare la dimensione dell'alfabeto in favore della complessità. Scegliamo infatti un "fattore di compressione" r tale che $rc > 2$. Per ogni alfabeto Γ_i dell'i-esimo nastro di $\mathcal{M}$ costruisco Γ_i' di $\mathcal{M}'$ assegnando un elemento per ogni $s \in \Gamma_i^r$. Ci basterà infine costruire l'organo di controllo di $\mathcal{M}'$ in modo tale per cui:

1. Calcoli con i nuovi simboli sui nastri emulando le mosse di $\mathcal{M}$ spostando le testine sui nastri ogni r movimenti di $\mathcal{M}$;
2. Memorizzi la posizione della testina arricchendo ulteriormente gli alfabeti Γ_i, oppure arricchendo l'insieme degli stati.

Teorema 8.2 (Secondo teorema di accelerazione lineare)**.** *Se L è il linguaggio accettato da una TM $\mathcal{M}$ a k nastri in $S_{\mathcal{M}}(n)$, posso costruire una TM $\mathcal{M}'$ a 1 nastro che accetta L con:*

$$S_{\mathcal{M}'}(n) = c \cdot S_{\mathcal{M}}(n) \tag{70}$$

Teorema 8.3 (Terzo teorema di accelerazione lineare)**.** *Se L è il linguaggio accettato da una TM $\mathcal{M}$ a k nastri in $S_{\mathcal{M}}(n)$, per ogni $c \in \mathbb{R}^+$ posso costruire una TM $\mathcal{M}'$ a 1 nastro che accetta L con:*

$$S_{\mathcal{M}'}(n) < c \cdot S_{\mathcal{M}}(n) \tag{71}$$

Teorema 8.4 (Quarto teorema di accelerazione lineare)**.** *Se L è il linguaggio accettato da una TM $\mathcal{M}$ a k nastri in $T_{\mathcal{M}}(n)$ sovralineare, per ogni $c \in \mathbb{R}^+$ posso costruire una TM $\mathcal{M}'$ a $k+1$ nastri che accetta L con:*

$$T_{\mathcal{M}'} = \max(n+1, c \cdot T_{\mathcal{M}}) \tag{72}$$

Come anche il primo teorema di accelerazione lineare, il trucco sta nel codificare in modo compresso i simboli dell'alfabeto di $\mathcal{M}$. Dobbiamo considerare che la compressione avviene a runtime, quindi $T_{\mathcal{M}'}$ sarà al massimo lineare. Comprimendo r simboli in uno, nel caso pessimo, possono servirmi 3 mosse di $\mathcal{M}'$ per emulare $r+1$ di $\mathcal{M}$.

I teoremi sopra sono validi anche per le macchine di Von Neumann. Infatti equivalgono ad aumentare la dimensione della parola di memoria della macchina o, equivalentemente, ad usare operazioni vettoriali. Possiamo avere speedup lineari arbitrariamente grandi aumentando il parallelismo fisico (stando ai limiti della fisica ovviamente). Miglioramenti più che lineari sono possibili solo cambiando algoritmo. Quindi concepire algoritmi più efficienti è di gran lunga più efficace della forza bruta.

8.3 Un nuovo modello di calcolo: la macchina RAM

Le macchine di Turing e i calcolatori hanno solo differenze marginali: un calcolatore è capace di fare operazioni aritmetiche su tipi a dimensione finita in tempo costante, la TM invece deve eseguire l'aritmetica bit per bit. Ciò può essere "emulato" considerando una TM con un alfabeto molto vasto: $|I| = 2^w$ se volgiamo operare in binario con una parola di w bit. Inoltre un calcolatore può accedere direttamente ad una cella di memoria, mentre la TM deve spendere $\Theta(n)$ (n distanza tra la cella e la testina) tempo per scorrere il nastro. È giunta ora di aggiornare il nostro modello di calcolo per modellare un po' più da vicino i calcolatori reali: la macchina RAM.

La macchina RAM è dotata di un nastro di lettura `In` e uno di scrittura `Out` come anche una TM. Il programma che esegue lo assumiamo essere cablato nell'organo di controllo e composto da diverse istruzioni che essa esegue. Anche la logica per la gestione del program counter assumiamo essere già cablata. La RAM è dotata di una memoria ad indirizzamento diretto e randomico (da cui prende il nome) $\texttt{M}[n]$, $n \in \mathbb{N}$. Tutte le istruzioni del programma useranno la prima cella di memoria $\texttt{M}[0]$, detta accumulatore. Ogni cella di memoria inoltre conterrà un intero.

Enunciamo l'instruction set e la corrispettiva semantica.

Istruzione	Semantica
`LOAD X`	$\texttt{M}[0] \leftarrow \texttt{M}[X]$
`LOAD= X`	$\texttt{M}[0] \leftarrow X$
`LOAD* X`	$\texttt{M}[0] \leftarrow \texttt{M}[\texttt{M}[X]]$
`STORE X`	$\texttt{M}[X] \leftarrow \texttt{M}[0]$
`STORE* X`	$\texttt{M}[\texttt{M}[X]] \leftarrow M[0]$
`ADD X`	$\texttt{M}[0] \leftarrow \texttt{M}[0] + \texttt{M}[X]$
`SUB X`	$\texttt{M}[0] \leftarrow \texttt{M}[0] - \texttt{M}[X]$
`MUL X`	$\texttt{M}[0] \leftarrow \texttt{M}[0] * \texttt{M}[X]$
`DIV X`	$\texttt{M}[0] \leftarrow \texttt{M}[0] / \texttt{M}[X]$
`ADD= X`	$\texttt{M}[0] \leftarrow \texttt{M}[0] + X$
`SUB= X`	$\texttt{M}[0] \leftarrow \texttt{M}[0] - X$
`MUL= X`	$\texttt{M}[0] \leftarrow \texttt{M}[0] * X$
`DIV= X`	$\texttt{M}[0] \leftarrow \texttt{M}[0] / X$
`HALT`	$-$
`READ X`	$\texttt{M}[X] \leftarrow In$
`READ* X`	$\texttt{M}[\texttt{M}[X]] \leftarrow In$
`WRITE X`	$Out \leftarrow \texttt{M}[X]$
`WRITE= X`	$Out \leftarrow X$
`WRITE* X`	$Out \leftarrow \texttt{M}[\texttt{M}[X]]$
`JUMP l`	$PC \leftarrow l$
`JZ l`	$PC \leftarrow l$ se $\texttt{M}[0] = 0$
`JGZ l`	$PC \leftarrow l$ se $\texttt{M}[0] > 0$
`JLZ l`	$PC \leftarrow l$ se $\texttt{M}[0] < 0$

Tabella 3: Instruction set (non completo) della macchina RAM.

Si può notare dalla tabella 3 che sono ammesse 3 modalità di indirizzamento: diretto, immediato e indiretto. Le tre modalità funzionano come in un normale linguaggio assembly.

8.4 Limiti del criterio di costo

Consideriamo il caso del calcolo di 2^{2^n} con una una macchina RAM. Uno schema di implementazione può essere:

```
1  read(n);
2  x = 2;
3  for (int i = 0; i < n; i++) x = x * x;
4  write(x);
```

La complessità temporale dell'implementazione sopra è $T_{RAM} = \Theta(n)$. Qualcosa non quadra: mi servono 2^n bit solo per scrivere il risultato! Il criterio di costo fino ad ora usato considera un intero arbitrario di dimensione costante. L'approssimazione regge fin quando una singola parola della macchina reale contiene gli interi che maneggiano. Se questo non accade, dobbiamo tener conto del numero di cifre necessarie per rappresentare un intero, perdendo il costo costante del salvataggio di interi e delle operazioni elementari su di essi. In questi casi eseguire operazioni su un intero i costa tanto quanto il suo numero di cifre in base b.

Forniamo allora una criterio di costo più realistico: il costo logaritmico. Una operazione su un intero i costerà tanto più sono le sue cifre in una base $b \geq 2$, ossia $\log_b(i) = \Theta(\log(i))$. Possiamo omettere la base del logaritmo poiché i logaritmi in diverse basi differiscono tra di loro solo per una costante (vedasi la formula di cambiamento di base del logaritmo).

Applichiamo il nuovo criterio di costo alle principali operazioni aritmetiche. Consideriamo $d = \log_2(i)$:

- Le addizioni e sottrazioni sono operazioni effettuate bit per bit, quindi avranno costo $\Theta(d)$.
- La moltiplicazione eseguita con il metodo scolastico richiede di iterare sulle cifre 2 volte, quindi avrà costo $\Theta(d^2)$. Studi sull'aritmetica ci dicono che la complessità è migliorabile, sebbene con un compromesso nella facilità di calcolo:
  - Primo miglioramento: $\Theta(d^{\log_2(3)}) \approx \Theta(d^{1.58})$
  - Secondo miglioramento: $\Theta(d \log(d) \log(\log(d)))$
  - Miglior costo attualmente scoperto: $\Theta(d \log(d))$
- Le divisioni con il metodo scolastico, analogamente alle moltiplicazioni, hanno costo $\Theta(d^2)$. Si è dimostrato che la complessità può essere migliorata a $\Theta(\log^2(d)) \cdot \mathit{costo\_mul}$ dove $\mathit{costo\_mul}$ è il costo dell'algoritmo di moltiplicazione scelto.

Le operazioni di `JUMP` e `HALT` hanno costo costante, così come anche le istruzioni di salto condizionale.

Per scegliere tra i due criteri basta che controlliamo l'accuratezza dell'approssimazione a costo costante: se la dimensione di ogni singolo elemento in ingresso non varia in modo significativo durante l'esecuzione usiamo il costo cotante e altrimenti quello logaritmico.

8.5 Legami tra le complessità dei vari modelli di calcolo

Quanto effetto ha sulla complessità cambiare modello di calcolo? Sotto ragionevoli ipotesi di criteri di costo, se un problema è risolvibile da una TM $\mathcal{M}$ in $T_{\mathcal{M}}(n)$, allora è risolvibile da un qualsiasi altro modello Turing-completo in $\pi(T_{\mathcal{M}})$ dove $\pi(\cdot)$ è un opportuno polinomio. Questa affermazione è detta "tesi di correlazione lineare". Noi la dimostreremo solo tra i due modelli più importanti, la macchina RAM e la macchina di Turing.

Prima di passare alla tesi di correlazione lineare, dimostriamo un risultato intermedio che useremo.

Lemma 8.1 (Occupazione sul nastro principale)**.** *Sia una macchina di Turing e una macchina RAM. Lo spazio occupato sul nastro principale è $\mathcal{O}(T_{RAM}(n))$.*

Dimostrazione. Ogni cella della RAM occupa $\log(i_j) + \log(\texttt{M}[i_j])$ spazio e viene materializzata se e solo se la RAM effettua un operazione di `STORE`. L'istruzione di `STORE` costa alla RAM $\log(i_j) + \log(\texttt{M}[i_j])$, quindi per riempire r celle la RAM impiegherà $\sum_{j=1}^{r} \log(i_j) + \log(\texttt{M}[i_j])$. Questa quantità di tempo è identica allo spazio che r celle di memoria occuperanno sul nastro della TM. □

Teorema 8.5 (Correlazione temporale tra TM a k nastri e RAM)**.** *Se un problema è risolvibile da una TM a k nastri $\mathcal{M}$ in $T_{\mathcal{M}}(n)$ e da una macchina RAM in $T_{RAM}(n)$ secondo costo logaritmico, allora si avrà:*

$$T_{\mathcal{M}}(n) = \pi(T_{RAM}(n)) \tag{73}$$

Dove $\pi(\cdot)$ è un opportuno polinomio.

Dimostrazione. Come primo passo dimostriamo che possiamo emulare un TM con una RAM in tempo polinomiale. Mappiamo innanzitutto le varie parti di una TM su una RAM:

- Lo stato della TM corrisponderà all'accumulatore della RAM;

- Una cella della RAM corrisponderà ad una cella di nastro;
- La RAM è suddivisa in blocchi da k celle.

I blocchi saranno riempiti secondo questa strategia: nel blocco 0 saranno salvate in ogni cella le posizioni delle k testine; nei rimanenti $n > 0$ blocchi sarà salvato lo n-esimo blocco di ognuno dei k nastri. La RAM emulerà la lettura di un carattere sotto la testina con un accesso indiretto, usando l'indice contenuto nel blocco 0. Studiamo ora il costo dell'emulazione delle due operazioni che può effettuare una macchina di Turing:

Lettura La lettura del blocco 0 e dello stato avviene in $\Theta(k)$ mosse. La lettura dei valori sui nastri in corrispondenza delle testine avverrà in $\Theta(k)$ accessi indiretti.

Scrittura La scrittura dello stato avviene in una mossa ($\Theta(1)$). Le scritture sui nastri e nel blocco 0, come anche le letture, impiegano rispettivamente $\Theta(k)$ accessi indiretti e $\Theta(k)$ mosse.

La RAM è quindi capace di emulare una mossa della TM con k mosse:

$$T_{RAM}(n) = \Theta(T_{\mathcal{M}}(n))[= \Theta(T_{\mathcal{M}}(n)\log(T_{\mathcal{M}}(n))) \text{ costo logaritmico}]$$

Studiamo ora l'emulazione da parte di una MT di una macchina RAM. Per semplicità ometteremo l'emulazione delle operazione di `MUL` e `DIV` in quanto non ledono alla generalità della dimostrazione. Organizziamo il nastro della TM come in figura 12.

Figura 12: Configurazione del nastro della TM.

Il nastro è inizialmente vuoto, salveremo solo le celle in cui è avvenuta una `STORE`. Usiamo anche un ulteriore nastro per contenere il valore di `M`[0] in binario. Useremo anche un ultimo nastro come stoccaggio temporaneo per quando serve salvare per la prima volta `M`$[i_j]$ ma `M`$[i_k]$ e `M`$[i_l]$ con $i_k < i_j < i_l$ sono già state salvate. Con questa configurazione, studiamo l'emulazione delle varie istruzioni della RAM (nel dettaglio vedremo solo `LOAD`, `STORE` e `ADD`):

`LOAD x` Devo effettuare una ricerca di `x` sul nastro principale e poi copiare la porzione di nastro accanto nella zona dati di `M`[0] usando il nastro di supporto.

`STORE x` Devo anche qui effettuare una ricerca di `x` sul nastro principale, se lo trovo vi salvo nello spazio adiacente il valore di `M`[0], altrimenti creo dello spazio usando il nastro di servizio se necessario e lo salvo.

`ADD x` Effettuo ancora una ricerca di `x`, copio `M`$[x]$ sul nastro di supporto, ne calcolo la somma scrivendo il risultato in `M`[0].

In generale possiamo dire grazie al lemma 8.1 che simulare una mossa della RAM richiede alle TM un numero di mosse minore o uguale ad una costante per la lunghezza del nastro principale. Quindi la TM impiega al più $\Theta(T_{RAM}(n))$ per simulare una mossa della RAM. Poiché ogni mossa della RAM ha costo unitario, possiamo dire che essa esegue $T_{RAM}(n)$ mosse. Quindi la simulazione completa della RAM da parte della TM costa al più

$$\Theta(T_{RAM}(n) \cdot T_{RAM}(n)) = \Theta((T_{RAM}(n))^2)$$

Quindi il legame tra $T_{RAM}(n)$ e $T_{\mathcal{M}}(n)$ è polinomiale. □

Il teorema sopra può essere esteso ad ogni modello computazionale affermando che con opportune ipotesi, complessità di diversi modelli computazionali sono correlate polinomialmente. Per quanto riguarda la relazione tra RAM e TM, possiamo raffinare il precedente teorema e ottenere i seguenti teoremi.

Teorema 8.6 (Simulazione di TM da parte di una RAM)**.** *Una TM multinastro con complessità temporale $T_{\mathcal{M}}$ può essere simulata da una macchina RAM con complessità temporale $T_{RAM} = \Theta(T_{\mathcal{M}})$, secondo il criterio di costo uniforme, o $T_{RAM} = \Theta(T_{\mathcal{M}} \cdot \log(T_{\mathcal{M}}))$, secondo quello logaritmico.*

Teorema 8.7 (Correlazione tra RAM e TM)**.** *Sia $\mathcal{L}$ un linguaggio accettato da una RAM con complessità temporale T_{RAM} secondo il criterio di costo logaritmico. Se il programma RAM non usa le istruzioni `MULT` e `DIV`, allora $\mathcal{L}$ può essere riconosciuto da una opportuna TM multinastro $\mathcal{M}$ in un tempo limitato da una funzione $T_{\mathcal{M}} = \Theta(T_{RAM}^2)$.*

L'esistenza della correlazione polinomiale tra macchina RAM e macchina di Turing ci permette di definire una classe di problemi risolvibili in tempo polinomiale (classe P). Questo risultato ha portato ad una "testi di trattabilità": i problemi risolvibili in P sono quelli trattabili. In questo caso per trattabili intendiamo risolvibili con complessità con esponente ragionevole. Infatti la classe P comprende anche polinomi come n^{30}, ma empiricamente si è visto che la maggioranza dei problemi polinomiali di interesse pratico ha un grado polinomiale accettabile di solito inferiore a 4.

9 Studio della complessità di algoritmi

Dato un certo problema, un buon flusso di lavoro è il seguente:

1. Concepire un algoritmo che lo risolve;
2. Valutare la complessità dell'algoritmo trovato;
3. Se la complessità calcolata è soddisfacente, implementare l'algoritmo nel linguaggio scelto.

Per controllare la correttezza di un algoritmo, come abbiamo già visto, non esiste una procedura generale. Ciò però non nega il fatto che per alcuni particolari casi possiamo costruire degli algoritmi di controllo.

Il modello che useremo per valutare la complessità è, ovviamente, la macchina RAM. Ci concentreremo, inoltre, sulla complessità temporale calcolata con il criterio di costo costante in quanto gli algoritmi trattati non hanno espansioni significative della dimensione dei singoli dati. Nei rari casi in cui ciò accade, possiamo rappresentare questi numeri molto grandi come vettori di numeri più piccoli. Il linguaggio che useremo per definire gli algoritmi è lo "pseudocodice". Esso non è altro che un'astrazione delle caratteristiche dei linguaggi di programmazione più comuni (C, Java, ecc...). Una assunzione fondamentale sullo pseudocodice sarà che ogni statement di esso può essere tradotto in un numero costante di istruzioni RAM.

9.1 La sintassi dello pseudocodice

Innanzitutto ogni algoritmo sarà definito come una procedura, ossia una funzione che prende un input e non ritorna nulla. Forniamo ora la sintassi base dello pseudocodice che useremo.

Operatori Sono definiti i soliti operatori matematici. Useremo $\leftarrow$ come assegnamento e riserveremo `=` per uguaglianza. Per i confronti useremo i soliti `<`, $\leq$, `>` e $\geq$.

Commenti Sono legali solo commenti monoriga delimitati da ▷ oppure usando `//`.

Costrutti di controllo del flusso Sono presenti tutti i costrutti di controllo del flusso che ci si aspetterebbe in un normale linguaggio C-like: `if`, `else`, `else if` e i cicli `for...`, `while` e `do...while`.

Strutture dati e tipi aggregati Sono definiti gli array e sono accessibili con una notazione C-like:

- `A[i]` per indirizzare lo $i-1$-esimo elemento (indicizzazione a partire da 0);
- `A[i..j]` per estrarre un sotto-array (slice).

È anche possibile creare tipi aggregati, i vari campi di questi sono indirizzabili tramite '`.`'. I tipi aggregati sono di default identificati da un puntatore alla struttura, quindi la notazione '`.`' equivale al `->` del C/C++. Un puntatore indefinito ha valore `NIL`.

Chiamata di funzioni Le chiamate a funzione seguono le consuete regole di chiamata e ritorno di un valore. I parametri di tipo base sono passati per valore, mentre i tipi aggregati per riferimento (convenzione Java-like).

Esempio 9.1. Consideriamo la rimozione di un elemento rispettivamente da un vettore e da una lista.

```
1  CancellaElV(v, len, e)
2    i ← 0
3    while v[i] ≠ e
4      i ← i + 1
5    while i < len -1
6      i ← v[i + 1]
7      i ← i + 1
8    v[len - 1] ← ⊥
9
10 CancellaElL(l, e)
11   p ← l
12   while p.next ≠ NIL and p.next.value ≠ e
13     p ← p.next
14   if p.next.value = e
15     p.next ← p.next.next
```

Sono entrambi, nel caso pessimo, $\Theta(n)$ dove n è il numero degli elementi del vettore/lista.

Esempio 9.2. Consideriamo il seguente algoritmo di moltiplicazione di matrici:

```
1  A ← Matrix(n, m)
2  B ← Matrix(m, o)
3
4  MatrixMultiply(A, B)
5    C ← Matrix(n, o)
6    for i ← 0 to A.n - 1
7      for j ← 0 to B.o - 1
8        C[i][j] ← 0
9        for k ← 0 to A.m - 1
10         C[i][j] ← C[i][j] + A[i][k] * B[k][j]
11   return C
```

La riga 5 viene eseguita $n * o$ volte, la 10 $n * m * o$ volte quindi l'algoritmo avrà complessità $\Theta(n * m * o)$ sia in generale che nel caso pessimo. Se le due matrici sono quadrate avremo $\Theta(n^3)$.

9.2 Studio di algoritmi ricorsivi

È possibile incontrare algoritmi la cui complessità non è immediatamente esprimibile in forma chiusa. Il caso tipico sono algoritmi che seguono la strategia "divide et impera". Questa strategia consiste nel suddividere il problema in sottoproblemi con input $\frac{1}{b}$ dell'originale; quando il sottoproblema ha ingresso di dimensioni n piccole a sufficienza può essere risolto a tempo costante. Chiamiamo $D(n)$

il costo del suddividere il problema e con $C(n)$ il costo di combinare le soluzioni. Possiamo quindi esprimere il costo totale $T(n)$ con la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n < c \\ D(n) + aT\left(\frac{n}{b}\right) + C(n) & \text{altrimenti} \end{cases} \tag{74}$$

Dove le costanti a, b e c indicano rispettivamente il numero di chiamate ricorsive, il numero di suddivisioni dell'input e il numero sotto al quale il sottoproblema ha costo costante.

Per risolvere un problema in questa forma possiamo utilizzare 3 tecniche: sostituzione, studio dell'albero di ricorsione e il teorema dell'esperto ("master theorem").

9.2.1 Metodo di sostituzione

Il metodo di sostituzione è sostanzialmente una dimostrazione per induzione del fatto che una soluzione "intuita" è effettivamente una soluzione. Esso si articola in 3 fasi:

1. Intuire una soluzione (ovviamente);
2. Sostituire la presunta soluzione nella ricorrenza;
3. Dimostrare per induzione che la presunta soluzione è tale per l'equazione/disequazione alle ricorrenze.

Usiamo come caso di studio l'algoritmo di ricerca binaria in quanto semplice e dall'implementazione intuitiva. Possiamo esprimere la complessità della ricerca binaria come:

$$T(n) = \Theta(1) + T\left(\frac{n}{2}\right) + \Theta(1) \tag{75}$$

"Intuiamo" una soluzione $T(n) = \mathcal{O}(\log(n))$, ossia $T(n) \leq c\log(n)$. Dobbiamo quindi dimostrare che

$$T(n) = \Theta(1) + T\left(\frac{n}{2}\right) + \Theta(1) \leq c\log(n) \tag{76}$$

Consideriamo vero per l'ipotesi di induzione $T(\frac{n}{2}) \leq c\log(\frac{n}{2})$ in quanto $\frac{n}{2} < n$ e sostituiamo ottenendo:

$$T(n) \leq c\log\left(\frac{n}{2}\right) + \Theta(k) = c\log(n) - c\log(2) + \Theta(k) \leq c\log(n) \tag{77}$$

Esempio 9.3. Determiniamo un limite superiore per $T(n) = 2T(\frac{n}{2}) + n$. Intuiamo $\mathcal{O}(n\log(n))$, dimostriamo quindi che $T(n) \leq cn\log(n)$. Supponiamo vero per induzione che $T(n/2) \leq c(n/2\log(n/2))$ e sostituiamo:

$$\begin{aligned} T(n) &\leq 2c(n/2\log(n/2)) + n \leq cnlog(n/2) + n = \\ &= cn\log(n) - cn\log(2) + n \leq \\ &\leq cn\log(n) + (1 - c\log(2))n < \\ &< cn\log(n) \end{aligned}$$

Per dimostrare il caso base possiamo trovare un n_0 per il quale vale la nostra ipotesi, in questo caso $n_0 = 3$.

Esempio 9.4. Troviamo un limite superiore per $T(n) = 2T(n/2) + 1$. Tentiamo di provare che $\mathcal{O}(n)$, ossia $T(n) = cn$. Supponiamo vero, sempre per induzione, $T(n/2) = cn/2$. Sostituiamo ottenendo che $T(n) \leq 2cn/2 + 1 = cn + 1$. Non possiamo trovare un valore che faccia rispettare l'ipotesi: $cn + 1 \geq cn$ sempre. In questo caso non siamo riusciti a dimostrare il limite tramite sostituzione. Attenzione: ciò non implica che $T(n) \neq \mathcal{O}(n)$! Infatti se prendiamo come ipotesi $T(n) \leq cn - b$ con b costante consente di dimostrare che $T(n) = \mathcal{O}(n)$.

9.2.2 Studio dell'albero di ricorsione

Lo studio dell'albero di ricorsione ci fornisce un aiuto per trovare una congettura da verificare con il metodo di sostituzione. L'albero di ricorsione è una rappresentazione delle chiamate ricorsive, indicando per ognuna la complessità. Ogni chiamata costituisce un nodo dell'albero, i chiamati appaiono come figli del chiamante. Rappresentiamo l'albero di:

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n \tag{78}$$

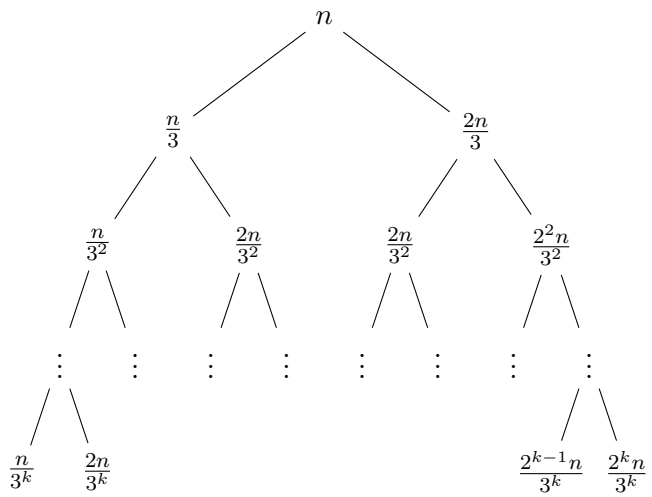

Figura 13: L'albero di ricorsione di 78.

L'albero ha la ramificazione a profondità massima posta sull'estrema destra della figura 13. Sappiamo che essa ha profondità k pari a:

$$\frac{2^k}{3^k}n = 1 \implies k = \log_3(2^k n) = \cdots = c\log_3(n) \tag{79}$$

Il costo pessimo per il contributo di un dato livello è la n della radice dell'albero, congetturiamo allora che $T(n) = \Theta(n\log(n))$. Possiamo quindi proseguire la dimostrazione con i metodi visti in 9.2.1.

9.2.3 Teorema dell'esperto

Il teorema dell'esperto è uno strumento per risolvere buona parte delle equazioni alle ricorrenze. Affinché sia applicabile, la ricorrenza deve avere la seguente forma:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{80}$$

Con $a \geq 1$ e $b > 1$. L'idea di fondo è quella di confrontare $n^{\log_b(a)}$ (effetto delle chiamate ricorsive) con quello di $f(n)$ (costo di ogni singola chiamata). Le ipotesi del teorema sono le seguenti:

1. a deve essere costante e maggiore di 1;
2. $f(n)$ deve essere sommata, non sottratta o altro a $aT(\frac{n}{b})$;
3. ~~Il legame tra $n^{\log_b(a)}$ e $f(n)$ deve essere polinomiale~~.

Se queste ipotesi sono valide, è possibile ricavare informazioni sulla complessità a seconda del caso in cui ci si trova:

Caso 1 Nel primo caso abbiamo che $f(n) = \mathcal{O}(n^{\log_b(a)-\epsilon})$ per un $\epsilon > 0$. La complessità risultante sarà $T(n) = \Theta(n^{\log_b(a)})$. Intuitivamente si può intendere questa situazione come il caso nel quale il costo della ricorsione domina quello della chiamata.

Caso 2 Nel secondo caso abbiamo che $f(n) = \mathcal{O}(n^{\log_b(a)}(\log(n))^k)$. La complessità risultante sarà $T(n) = \Theta(n^{\log_b(a)}(\log(n))^{k+1})$. In questo caso, invece, possiamo intuire che il contributo della ricorsione e quello della singola chiamata differiscono per meno di un termine polinomiale.

Caso 3 Nell'ultimo caso abbiamo che $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ per un $\epsilon > 0$. Se questo è vero, deve anche valere $af(\frac{n}{b}) < cf(n)$ per un qualche valore di $c < 1$. Se le ipotesi sono rispettate abbiamo che $T(n) = \Theta(f(n))$. Qui, invece, domina il costo della singola chiamata.

Variante per funzioni decrescenti Si può enunciare anche un'altra variante del teorema dell'esperto, valido per ricorrenze del tipo $T(n) = aT(n-b) + f(n)$ con $a > 0, b > 0, f(n) = O(n^k)$:

1. Se $a = 1$ si ha che la ricorrenza sarà $T(n) = O(nf(n))$;
2. Se $a > 1$ si ha che la ricorrenza sarà $T(n) = O(f(n) \cdot a^{\frac{n}{b}})$;
3. Se $a < 1$ si ha che la ricorrenza sarà $T(n) = O(f(n))$;

9.3 Algoritmi di ordinamento

Tra i problemi che capita più spesso di dover risolvere, l'ordinamento di una collezione di oggetti è un classico. Un punto chiave dell'utilità dell'ordinamento è consentire utilizzare una ricerca più efficiente, come ad esempio quella binaria, sulla nostra collezione. Analizzeremo soluzioni diverse considerando la loro complessità temporale, spaziale e relative peculiarità.

Definizione 9.1 (Proprietà di stabilità di un ordinamento)**.** Diciamo che un ordinamento gode della proprietà di stabilità se non modifica l'ordine di elementi duplicati.

9.3.1 Insertion sort

L'insertion sort è il primo e il tra i più naturali algoritmi di ordinamento che vedremo. L'algoritmo modella come un uomo ordinerebbe, ad esempio, un mazzo di carte: selezioniamo un elemento della slice non ordinata e reinseriamolo al giusto posto della slice ordinata.

```
1 InsertionSort(A)
2   for i ← 1 to A.length - 1
3     tmp ← A[i]
4     j ← i - 1
5     while j ≥ 0 and A[j] > tmp
6       A[j + 1] ← A[j]
7       j ← j - 1
8     A[j + 1] ← tmp
```

Snippet 1: Insertion Sort

Nel caso ottimo abbiamo una complessità di $\Theta(n)$, in quello pessimo invece $\Theta(n^2)$, in generale invece $\mathcal{O}(n^2)$. La complessità spaziale è invece costante in quanto tutte le operazioni avvengono in-place. Inoltre otterremo un algoritmo stabile usando `A[j] > tmp`, mentre uno instabile usando `A[j]` $\geq$ `tmp`.

9.3.2 Merge sort

Possiamo sicuramente trovare algoritmi più efficienti. Intuitivamente si può vedere che ogni algoritmo di ordinamento sarà $\Omega(n)$ e $\mathcal{O}(n^2)$ (si possono trovare anche limiti superiori maggiori, ma noi consideriamo solo ordinamenti "intelligenti"). Proviamo ad avvicinarci al limite inferiore teoretico. Astraiamo dalla specifica strategia di ordinamento e studiamo il numero di confronti e scambi. Consideriamo per semplicità un array di 3 elementi v. Come si può vedere dalla figura 14, otteniamo un albero quasi binario con $n!$ foglie, dove n è il numero di elementi. Assumiamo che non ci siano confronti ridondanti. La lunghezza del più lungo percorso radice-foglia è il numero massimo di confronti che devo fare per ordinare un vettore. L'altezza dell'albero sarà:

$$\log_2(n!) \approx n\log_2(n) - n\log_2(e) + \mathcal{O}(\log(n)) \tag{81}$$

Dove abbiamo usato l'approssimazione di Sterling del fattoriale per ottenere che la migliore complessità ottenibile con questo metodo sarà $\mathcal{O}(n\log(n))$.

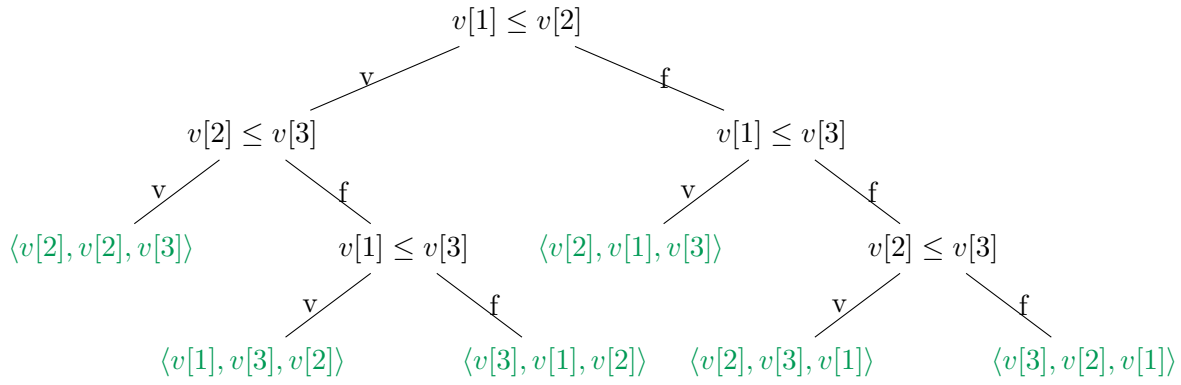

Figura 14: Albero dei confronti per l'ordinamento di v

Il primo algoritmo che vediamo che raggiunge questa complessità sarà il merge sort. Il merge sort è un ordinamento "divide et impera" con complessità $\Theta(n\log(n))$ in tutti i casi. Nonostante l'eccezionale complessità in caso pessimo, non è l'ordinamento perfetto perché è caratterizzato da molte operazioni di memoria e una non trascurabile complessità spaziale.

Il merge sort prima suddivide il vettore di elementi da ordinare in porzioni più piccole, fin quando le partizioni non sono ordinabili in $\Theta(1)$, dopodiché riassembla le varie slice. È importante che il riassemblamento dei risultati non abbia complessità eccessiva. Poiché la procedura di unione delle varie slice è la parte principale della complessità, studiamo prima quella.

Consideriamo due slice `A[p..q]` e `A[q + 1..r]`, è facile dimostrare che la complessità della ricombinazione di questi sia lineare:

```
1  Merge(A, p, q, r)
2    len1 ← q - p + 1
3    len2 ← r - q
4    Alloc(L[1..len1 + 1])
5    Alloc(R[1..len2 + 1])
6    for i ← 1 to len1 // copia della prima meta'
7      L[i] ← A[p + i - 1]
8    for i ← 1 to len2 // copia della seconda meta'
9      R[i] ← A[q + i]
10   L[len1 + 1] ← ∞ // sentinella
11   R[len2 + 1] ← ∞ // sentinella
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
```

Snippet 2: Funzione di merge dei vettori

Vediamo ora il merge sort:

```
1 MergeSort(A, p, r)
2   if p < r - 1
3     q ← ⌊(p+r)/2⌋
4     MergeSort(A, p, q)
5     MergeSort(A, q + 1, r)
6   else // caso base della ricorsione: solo 2 elementi
7     if A[p] < A[r]
8       tmp ← A[r]
9       A[r] ← A[p]
10      A[p] ← tmp
```

Snippet 3: Merge Sort

Il costo del merge sort sarà dato dalla ricorrenza $T(n) = 2T(\frac{n}{2}) + \Theta(n)$. Usando il secondo caso del master theorem otteniamo $T(n) = \Theta(n \log(n))$.

La stabilità del merge sort dipende dall'algoritmo d'unione: se consideriamo $\leq$ allora sarà stabile, instabile con $<$.

9.3.3 Quicksort

Quicksort sarà il secondo algoritmo che vedremo che ha complessità vicina al minimo teoretico. Anche esso è, come merge sort, un algoritmo di tipo "divide et impera". La sua peculiarità è che l'ordinamento avviene senza uso di spazio ausiliario.

Il quicksort applica la strategia "divide et impera" ad una slice `A[lo..hi]`:

Dividi Scelgo un elemento `A[p]`, detto pivot, come punto di suddivisione della slice di partenza e sposta tutti gli elementi di essa in modo che tutti quelli di `A[lo..p - 1]` siano minori del pivot.

Impera Ordina ricorsivamente `A[lo..p - 1]` e `A[p + 1..hi]` con quicksort.

Combina Non c'è nulla da ricombinare in quanto tutto è eseguito in place.

L'algoritmo di quicksort è molto semplice:

```
1 Quicksort(A, lo, hi)
2   if lo < hi
3     p ← Partition(A, lo, hi)
4     Quicksort(A, lo, p - 1)
5     Quicksort(A, p + 1, hi)
```

Snippet 4: Quicksort

La parte più importante dell'algoritmo è `Partition()`. Studiamo due implementazioni di questa funzione: una è attribuita a Lomuto (`PartitionLomuto()`) mentre l'altra è attribuita al creatore del quicksort, Hoare (`PartitionHoare()`).

```
1 PartionLomuto(A, lo, hi)
2   pivot ← A[hi]
3   i ← lo - 1
4   for j ← lo to hi
5     if A[j] ≤ pivot
6       i ← i + 1
7       Scambia(A, i, j)
8   Scambia(A, i + 1, hi)
```

Snippet 5: Partizione secondo Lomuto

```
1 PartitionHoare(A, lo, hi)
2   pivot ← A[lo]
3   i ← lo - 1
4   g ← hi + 1
5   while true
6     do
7       i ← i + 1
8     while A[i] < pivot
9     do
10      j ← j - 1
11    while A[1] > pivot
12    if i < j
13      Scambia(A, i, j)
14    else
15      return j
```

Snippet 6: Partizione secondo Hoare

La complessità della partizione secondo Lomuto è $\Theta(n)$, come anche quella di Hoare ma la seconda effettua (in media) 1/3 degli scambi rispetto alla prima. La partizione di Hoare ha anche un altro vantaggio: se il vettore è composto da solo elementi uguali non eseguiamo nessuno scambio.

La complessità del quicksort sarà:

$$T(n) = T(n/a) + T(n - n/a) + \Theta(n) \tag{82}$$

Dove a dipende da quanto bene `Partition()` ha suddiviso il vettore. Il caso pessimo del quicksort è un vettore diviso in porzioni lunghe $n-1$ e 1. La ricorrenza diventerebbe:

$$T(n) = T(n-1) + T(1) + \Theta(n) \tag{83}$$

Che si dimostra facilmente essere $\Theta(n^2)$. È un caso molto specifico e molto poco probabile nella vita reale in quanto i vettori sono solitamente ordinati in modo casuale. Il caso ottimo è un vettore diviso in due porzioni lunghe $n/2$. La ricorrenza diventa:

$$T(n) = 2T(n/2) + \Theta(n) \tag{84}$$

La stessa del merge sort, che è $\Theta(n \log(n))$. La costante nascosta dalla notazione asintotica si può calcolare ed è pari a 1.39 rendendo il quicksort solo 39% più lento del minimo teorico.

9.3.4 Counting sort

Sappiamo che non è possibile essere più veloci di $\Theta(n \log(n))$ usando algoritmi di ordinamento per confronto. Le cose cambiano se però eliminiamo la necessità di effettuare confronti. Consideriamo un set di dati di cui sappiamo che il dominio ha dimensioni contenute. Un metodo per per ordinare gli elementi senza confrontarli è calcolare prima l'istogramma delle frequenze e poi restituirne gli elementi in ordine. Questa è l'idea dietro al counting sort.

```
1 CountingSort(A)
2   Is[0..k] ← 0 // operazione dal costo Θ(k)
3   for i ← 0 to A.length
4     Is[A[i]] ← Is[A[i]] + 1
5   idxA ← 0
6   for i ← 0 to k
7     while Is[i] > 0
8       A[idxA] ← i
9       idxA ← idxA + 1
```

```
10        Is[i] ← Is[i] - 1
```

Snippet 7: Counting sort

La complessità temporale è dominata dalle righe 6–10: $\Theta(n + k)$, dove k è la dimensione del nostro dominio. Se $k \gg n$ la complessità sfugge dal controllo.

Nella versione sopra, il counting sort non è stabile. È possibile, però, escogitare una strategia per renderlo stabile. Il counting sort stabile parte come la variante instabile con il calcolare il numero delle occorrenze di ogni elemento. L'istogramma costruito, però, viene trasformato nel vettore contenente il conteggio degli elementi con valori maggiori o uguali dell'indice del vettore. Calcolato ciò, piazza un elemento calcolando la sua posizione come il valore corrente dell'informazione nel vettore, decrementandola.

```
1  CountingSort(A) // variante out-of-place
2    B[0..A.length - 1] ← 0
3    Is[0..k] ← 0
4    for i ← 0 to A.length
5      Is[A[i]] ← Is[A[i]] + 1
6    sum ← 0
7    for i ← 0 to k // calcola il numero di elementi ≤ i
8      sum ← sum + Is[i]
9    for i ← A.length - 1 to 0
10     idx ← Is[A[i]]
11     B[idx - 1] ← A[i]
12     Is[A[i]] ← Is[A[i]] - 1
13   return B
```

Snippet 8: Counting sort (versione stabile)

10 Strutture dati

Fino ad ora non ci siamo mai preoccupati del modo in cui abbiamo salvato i nostri dati in memoria. L'unico modo, e il più semplice, che abbiamo usato è stato quello di salvare tutti gli elementi in sequenza. Questa modalità di salvataggio, o struttura dati, viene detta vettore. Analizziamo quindi come è possibile distribuire in memoria collezioni di elementi in modo ottimizzato e intelligente.

Una struttura dati usa delle etichette opache (chiavi) per identificare gli oggetti al suo interno. Esse, inoltre, supportano diverse operazioni sui loro contenuti:

- `Search(S, k)`: ricerca in `S` dell'elemento con chiave `k`;
- `Minimum(S)`: ricerca dell'elemento più piccolo in `S`;
- `Maximum(S)`: ricerca dell'elemento più grande in `S`;
- `Successor(S, x.k)`: restituisce il successore di `x` (se ordinati) o della sua chiave `k` in `S`;
- `Predecessor(S, x.k)`: restituisce il predecessore di `x` (se ordinati) o della sua chiave `k` in `S`;
- `Insert(S, x)`: inserisce `x` in `S`;
- `Delete(S, x)`: rimuove `x` da `S`.

10.1 Vettori

Analizziamo ora la complessità dei vettori. Essi sono una struttura compatta in memoria in cui si accede direttamente ad ogni elemento data la sua posizione. L'indice del vettore agisce come chiave a tutti gli effetti.

Se il vettore di lunghezza `n` non è ordinato le operazioni sopra sono tutte lineari. Se il vettore è, invece, ordinato:

- Minimo e massimo sono costanti, mentre ricerca e successore $\Theta(\log(n))$.
- Inserimento e cancellazione rimangono $\mathcal{O}(n)$.

Inoltre gli inserimenti in un vettore pieno possono o essere rifiutati, quindi $\Theta(1)$, oppure causare una riallocazione, quindi $\mathcal{O}(n)$.

10.2 Liste semplicemente connesse

Una lista semplice stocca gli elementi in modo sparso: ogni elemento ha un riferimento al successivo (puntatore). Se la lista di lunghezza `n` non è ordinata abbiamo che, come il vettore, tutte le operazioni sono lineari tranne l'inserimento che è costante (sempre in testa) e la cancellazione se viene fornito un riferimento all'elemento. Se la lista è, invece, ordinata:

- Uno dei due tra minimo e massimo è $\Theta(1)$, l'altro $\Theta(n)$. Se aggiungiamo un puntatore accessorio diventano entrambi costanti.
- Ricerca e successore sono $O(n)$.
- Inserimento e cancellazione diventano entrambi $O(n)$.

10.3 Pile o stack

Una pila è una struttura dati LIFO che supporta solo le seguenti operazioni: `Push(S, e)`, `Pop(S, e)` e `Empty(S)`. Una stack può essere implementata sia usando una lista che un vettore.

Se usiamo l'implementazione basata sulla lista, le 3 operazioni sono tutte costanti. Se utilizziamo quella basata sul vettore, invece, dobbiamo utilizzare una variabile ausiliaria (`ToS`, `Top Of Stack`) per mantenere la complessità costante. Inoltre `Push()` può avere costo lineare in caso di riallocazione.

Nella pratica, avere dati sparsi in memoria può penalizzare la performance a causa delle cache-misses. Può, quindi, valere la pena di usare una stack basata su un vettore usando uno schema di riallocazione intelligente.

10.4 Code o queues

Una coda è una struttura dati FIFO che supporta solo le seguenti operazioni: `Enqueue(Q, e)`, `Dequeue(Q)` e `Empty(Q)`. Come nel caso delle pile, è possibile realizzare una coda sia con una lista che con un vettore.

Vettore Se realizziamo una coda con vettore, lo stoccaggio dei dati sarà effettuato in un vettore `A` lungo `l` con indice del primo elemento `0`. Teniamo traccia della posizione dove va inserito un nuovo elemento e di quella dell'elemento più vecchio con due indici `tail` e `head` e del numero di elementi contenuti `n`. Gli indici verranno incrementati modulo `l` (vettore circolare). Le operazioni saranno implementate così:

- `Enqueue(Q, e)`: se `n < l`, inserisci l'elemento in `A[tail]` e incrementa `n` e `tail`, altrimenti segnala l'errore. Abbiamo una complessità costante.
- `Dequeue(Q, e)`: se `n > 0`, restituisci `A[head]` corrente, decrementa `n` e incrementa `head`. Abbiamo ancora complessità costante.
- `Empty(Q)`: restituisci `n == 0`. Ovviamente costante.

Per ampliare lo stoccaggio, possiamo creare una nuova coda di dimensione maggiori e spostare gli elementi uno a uno usando `Enqueue()` e `Dequeue()`.

Lista Se usiamo una lista teniamo traccia dell'ultimo elemento della lista (oltre al primo) con un puntatore `tail`, mantenendo invariati gli algoritmi delle varie operazioni. Possiamo, quindi, mantenere la complessità costante.

10.5 Mazzo (deque)

È una struttura dati che si comporta come un mazzo di carte. Possiamo vederla anche come una pila in cui è possibile aggiungere sia in testa che in coda. Le operazioni supportate sono: `PushFront(Q,e)`, `PushBack(Q,e)`, `PopFront(Q)`, `PopBack(Q)` e `Empty(Q)`. Come le pile e le code, il mazzo può essere implementato sia con un vettore, sia con una lista.

Vettore Se usiamo un vettore, adottiamo una strategia analoga a quella usata per le code implementate con vettore:

- `PushBack()` e `PopFront()` si comportano rispettivamente come `Enqueue()` e `Dequeue()`.
- La `PopBack()` restituisce `A[tail]`, decrementa `n` e `tail` se `n > 0`. Complessità costante.
- La `PushFront()` decrementa `head`, inserisce l'elemento in `A[head]` e incrementa `n` se `n < l`. Complessità costante.
- La `Empty()` restituisce `n = 0`. Complessità costante.

Anche per ampliare lo stoccaggio useremo la stessa strategia che abbiamo usato per le code.

Lista Per implementare un mazzo con una lista non basta più una lista semplice, singolarmente concatenata, ma serve una doppiamente concatenata. Questo ci permette di aggiungere e rimuovere in testa e in coda in tempo costante.

- `PushBack()` e `PopFront()` si comportano come `Enqueue()` e `Dequeue()` di una coda realizzata con una lista.
- La `PopBack()` restituisce `tail.prev` corrente se diverso da `head`, rimuovendolo dalla lista. Costo costante.
- La `PushFront()` aggiunge un elemento in testa, aggiornando `head` e il suo successore. Costo costante.
- la `Empty()` restituisce `head.next = tail.prev`. Costo costante.

10.6 Dizionari

Il dizionario è una struttura dati astratta che contiene elementi accessibili direttamente data la loro chiave. Assumiamo che le chiavi siano numeri naturali, nel caso che non lo siano è sufficiente considerare la loro rappresentazione binaria e il corrispondente numero. Le operazioni supportate sono `Insert()`, `Delete()` e `Search()`. Un dizionario può essere implementato con diverse strutture dati concrete.

10.6.1 L'approccio ingenuo

Un primo approccio ingenuo è quello di considerare un insieme di chiavi limitato e implementare un dizionario come un semplice array di puntatori indicizzato dalle chiavi stesse (direct-addressing). Le operazioni sarebbero:

- `Insert()`: `D[e.key]` $\leftarrow$ `e`
- `Delete()`: `D[e.key]` $\leftarrow$ `NIL`
- `Search()`: `return D[e.key]`

La complessità computazionale è costante per tutte le operazioni. La complessità spaziale, invece, sarà $\mathcal{O}(|D|)$, con D il dominio delle chiavi. Quindi anche per chiavi abbastanza semplici, questa implementazioni sarebbe assai onerosa in termini di spazio.

10.6.2 Tabelle Hash

Una tabella hash implementa un dizionario con una complessità di memoria pari al numero di chiavi per cui è effettivamente presente un valore. Quindi il domino delle chiavi può essere arbitrariamente grande, persino infinito.

Il tipico approccio per una implementazione è preallocare spazio per `m` chiavi e riallocare solo quando ci saranno `n > m` chiavi. Come indice di questo vettore useremo il risultato di una speciale funzione $h(k) : D \to \{0, \dots, m-1\}$ detta funzione di hash. Se il calcolo di $h(k)$ è $\mathcal{O}(k)$, la tabella di hash ha la stessa efficienza temporale del dizionario implementato nel precedente modo naif.

Gestione delle collisioni Idealmente, la funzione di hash dovrebbe mappare ogni chiave su di un distinto elemento del suo codominio. Ma ciò è impossibile poiché, per come li abbiamo definiti, $|D| \gg m$. Diremo che è avvenuta una collisione ogniqualvolta dati k_1, k_2 tali che $k_1 \neq k_2$ si ha che $h(k_1) = h(k_2)$. Vedremo 2 strategie per la gestione efficiente dei conflitti: l'indirizzamento chiuso, o open hashing, e l'indirizzamento aperto, o closed hashing.

Indirizzamento chiuso Ogni riga, detta bucket, della tabella contiene la testa di una lista al posto del puntatore ad un singolo elemento. Nel caso di collisione, l'elemento nuovo viene aggiunto in testa alla lista. Per cercare o cancellare un elemento di chiave k, è necessario cercare nell'intera lista del bucket $h(k)$.

Indirizzamento aperto In caso di collisione si seleziona secondo una regola deterministica un altro bucket in una procedura detta di ispezione (probing). Nel caso non si trovino bucket vuoti si può fallire, con costo $\Theta(m)$, oppure riallocare una tabella più grande e reinserire tutti gli elementi della vecchia nella nuova (re-hashing) con costo $\Theta(n)$. La funzione `Search()`, se l'elemento non viene trovato nel suo bucket, effettua la stessa ispezione svolta durante l'inserimento. La cancellazione è effettuata inserendo un opportuno valore, detto tombstone che non corrisponde ad alcuna chiave. Analizziamo diverse possibili procedure di probing e i rispettivi vantaggi e svantaggi.

Ispezione lineare È il metodo più semplice. Dato $h(k, 0) = a$ il bucket dove avviene la collisione al primo tentativo ($i = 0$), si sceglie $h(k, i) = a + ci \mod m$ come bucket candidato per l'i-esimo tentativo. Il problema principale di questo metodo è il caso in cui avvengono molte collisioni su un dato bucket. In questo caso, detto di clustering primario delle collisioni, la probabilità di collisione nei bucket adiacenti peggiorerà con ogni nuova ispezione. Per alcune scelte di $h(k)$, è possibile avere un forte peggioramento delle prestazioni dovuto al clustering. È possibile avere un clustering di dimensione logaritmica nella dimensione della tabella a costo di effettuare il re-hashing molto prima che la tabella sia piena (circa a metà).

Ispezione quadratica Per mitigare il fenomeno del clustering, usiamo una funzione di probing quadratica: $h(k, i) = a + c_1 i + c_2 i^2 \mod m$. Insorge, però, un altro problema: la funzione di probing non garantisce a priori la visita di tutte le celle. Esiste una sequenza particolare, però, che visita tutte le celle nel caso di tabelle di dimensione 2^m, come dimostrato dal lemma 10.1, Il problema del clustering però non è ancora del tutto risolto in quanto chiavi con la stessa posizione iniziale hanno la stessa sequenza di ispezione (clustering secondario). Sarà, anche in questo caso, necessario fare re-hashing a tabella non piena.

Lemma 10.1 (Sequenza di probing ideale per tabelle di dimensione 2^w)**.** *La sequenza:*

$$h(k, i) = a + \frac{1}{2}i + \frac{1}{2}i^2 \tag{85}$$

genera tutti i valori in $[0; m-1]$ *con* $m = 2^w$.

Dimostrazione. Effettuiamo una dimostrazione per assurdo. Supponiamo che esistano $0 < p < q < m-1$ tali che $\frac{1}{2}(p+p^2) = \frac{1}{2}(q+q^2) \mod m$. Ciò implica che $p+p^2 = q+q^2 \mod 2m$. Fattorizzando

l'eguaglianza otteniamo che $(q-p)(p+q+1) = 0 \mod 2m$. Analizziamo i 3 casi in cui si avvererebbe questa disuguaglianza:

1. Se $q - p = 0 \mod 2m$ significa che $q = p$; contro ipotesi e quindi impossibile.
2. $(p + q + 1) \neq 0 \mod 2m$ perché per la scelta di p e q la somma rientra nel range $[1; 2m - 2]$.
3. Poiché stiamo lavorando in modulo $2m$ e il modulo introduce divisori dello 0, è possibile che $(q-p)(p+q+1) = 0 \mod 2m$ anche con le considerazioni dei precedenti punti. Poiché $(q-p)-(p+q+1) = 2p+1$, significa che almeno uno dei due addendi è dispari. Essendo $m = 2^w$, il fattore pari è per forza multiplo di $2m = 2^{w+1}$ ma abbiamo che $(q - p) \leq m - 1$ e $(p + q + 1) \leq 2m - 2$ per come abbiamo scelto p e q. Assurdo!

È quindi impossibile che $(q-p)(p+q+1) = 0 \mod 2m$, quindi l'ipotesi iniziale $\frac{1}{2}(p+p^2) = \frac{1}{2}(q+q^2) \mod m$ è falsa. □

Doppio hashing Definiamo $h(k, i) = h_1(k) + h_2(k)i \mod m$ facendo sì che l'ispezione dipenda dalla chiave. Per garantire che la sequenza di probing sia in grado di visitare tutte le caselle dobbiamo fare sì che $h_2(k)$ sia coprimo con m. Un modo semplice per garantire ciò è:

- Per $m = 2^w$ fare sì che h_2 generi solo numeri dispari.
- Se m è primo, fare sì che h_2 generi numeri minori di m.

È importante che h_2 non restituisca mai zero, sennò la sequenza di probing degenererebbe.

Generazione di una funzione di hash Vediamo ora dei metodi per produrre delle funzioni di hash efficaci.

Metodo della divisione Il metodo più semplice è porre $h(k) = k \bmod m$. La distribuzione dei risultati non è uniforme in $\{0, \ldots, m-1\}$ e va evitato se $m = 2^i$ in quanto $h(k)$ dipenderebbe solo dai bit meno significativi. Non funziona però malissimo se m è un primo vicino ad una potenza di 2.

Metodo della moltiplicazione Un altro metodo semplice è porre $h(k) = \lfloor n(\alpha k - \lfloor \alpha k \rfloor) \rfloor$, con $\alpha \in \mathbb{R}$ costante. In questo caso, la dimensione della tabella m non è critica. Perciò spesso si prende $m = 2^w$ in modo da effettuare le moltiplicazioni con semplici shift. Una scelta possibile per α è la sezione aurea $\frac{\sqrt{5}+1}{2}$: si è visto empiricamente che fornisce una buona distribuzione.

Hashing universale Finora abbiamo ipotizzato che le collisioni fossero accidentali. Esse però possono essere anche indotte sfruttando le debolezze della funzione di hashing. Un modo per evitare questo problema è non scegliere una sola funzione di hashing, ma una intera famiglia di buone funzioni. Si può dimostrare che $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$ con $p > m$ primo distribuisce uniformemente le chiavi della tabella per ogni $a, b \in \mathbb{Z}_0$. Sarà quindi sufficiente scegliere casualmente a e b per ogni istanza della tabella.

Costo delle operazioni Nel caso pessimo, ossia quello in cui tutti gli elementi collidono, la struttura degenera in un array/lista di lunghezza `n`: $\mathcal{O}(n)$. Studiamo separatamente le due modalità di indirizzamento nel caso medio.

Ipotesi di hashing uniforme semplice Aggiungiamo prima una ipotesi semplificatoria che empiricamente possiamo ritenere soddisfatta per una buona funzione di hash: ogni chiave ha la stessa probabilità $\frac{1}{m}$ di finire in una qualsiasi delle m celle della tabella. Chiamiamo questa ipotesi di hashing uniforme semplice.

Indirizzamento chiuso Definiamo fattore di carico $\alpha = \frac{n}{m}$ con n il numero di chiavi inserite e m la lunghezza della tabella. Sotto hashing uniforme semplice, abbiamo che la lunghezza media delle liste sarà il fattore di carico. Quindi il tempo medio per cercare una chiave non presente sarà $\Theta(\alpha + 1)$, come anche quello per la ricerca di una chiave presente nella tabella. Quindi se α non è eccessivo, in media tutte le operazioni avranno complessità $\Theta(1)$.

Indirizzamento aperto Il tempo impiegato per trovare un elemento dipenderà dalla sequenza di ispezione. Generalizziamo l'ipotesi di hashing uniforme semplice dicendo che tutte le sequenze di ispezione sono equiprobabili (hashing uniforme). Considerando il fattore di carico definito precedentemente, sia X la variabile aleatoria che modella il numero di passi di ispezione fatti senza trovare il valore desiderato, abbiamo che $P(X > i) = \alpha^{i+1}$. Il numero medio di passi di ispezione sarà:

$$E(X) = \frac{1}{1-\alpha} \tag{86}$$

Il numero medio di tentativi si ricava essere:

$$\frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right) \tag{87}$$

10.7 Alberi binari

Vediamo ora una importante struttura dati: l'albero. Abbiamo già fatto uso degli alberi in maniera informale. Gli alberi con adeguate proprietà sono una rappresentazione efficiente per insiemi di dati ordinati.

Definizione 10.1 (Albero)**.** Un albero è una coppia (V, E) dove V è un insieme di nodi e E un insieme di archi, ossia di coppie di nodi ordinate. Ogni nodo può apparire un'unica volta come destinazione di un arco (secondo elemento della coppia). Non sono possibili cicli.

Chiameremo radice l'unico nodo dell'albero privo di un arco entrante, mentre foglia quelli senza archi uscenti. Il padre di un nodo sarà il nodo da cui l'arco entrante in esso ha origine; viceversa il figlio di un nodo è il nodo in cui degli archi uscenti da esso termina. Chiameremo livello la distanza in numero di archi tra la radice ed un nodo e, infine, diremo che un albero è completo se non è possibile aggiungere un nodo con livello minore o uguale a quello dei nodi già presenti.

Noi studieremo gli alberi binari, ossia alberi in cui un nodo ha al massimo 2 figli. È utile dare una definizione ricorsiva di albero binario:

Definizione 10.2 (Albero binario)**.** Un albero è formato da un nodo radice a cui sono collegati due alberi, quello destro e quello sinistro. Un albero può essere vuoto.

Le azioni sull'albero indicizzeranno i nodi con una chiave intera, in modo simile a come facevamo per le tabelle di hash. Definiamo un nodo `N` la struttura:

```
1 struct Node:
2   Node left;    // figlio sinistro
3   Node right;   // figlio destro
4   Node parent; // padre (NIL per la radice)
5   int   key;    // chiave
6   Data  data;
```

Ogni albero `A` è un riferimento alla radice.

Un albero può essere materializzato in memoria con una struttura basata su puntatori oppure tramite un array: la radice sarà l'elemento con indice 0; dato un nodo contenuto in posizione i il suo figlio sinistro sarà in posizione $2i + 1$ e quello destro in $2i + 2$; dato un nodo in posizione i il padre sarà salvato in posizione $\lfloor \frac{i-1}{2} \rfloor$.

Visita di un albero L'operazione naturale di un albero, oltre ai consueti inserimento, ricerca e cancellazione, è l'attraversamento (o visita). La visita di un albero consiste nell'enumerare le chiavi contenute secondo un particolare ordine. Ovviamente le visite avranno tutte complessità $\Theta(n)$ con n il numero di nodi dell'albero. Vediamo alcune delle strategie più usate.

```
1 InOrder(T)
2   if T = NIL
3     return
4   InOrder(T.left)
5   Print(T.key)
6   InOrder(T.right)
```

Snippet 9: Visita in ordine

```
1 PreOrder(T)
2   if T = NIL
3     return
4   Print(T.key)
5   PreOrder(T.left)
6   PreOrder(T.right)
```

Snippet 10: Visita anticipata

```
1 PostOrder(T)
2   if T = NIL
3     return
4   PostOrder(T.left)
5   PostOrder(T.right)
6   Print(T.key)
```

Snippet 11: Visita posticipata

10.7.1 Alberi binari di ricerca

Uno degli usi più comuni degli alberi binari è utilizzare quelli per cui è valida una data relazione tra le chiavi. Vediamo uno particolare di questi: l'albero di ricerca binario o BST.

Definizione 10.3 (Albero di ricerca binario)**.** Un albero binario è detto albero di ricerca binario (BST) se per un qualunque suo nodo valgono:

1. Se y è contenuto nel sotto-albero sinistro di x vale $y.key \leq x.key$
2. Se y è contenuto nel sotto-albero destro di x vale $y.key \geq x.key$

È ovvio che inserimenti e cancellazioni devono conservare questo invariante. Una visita in ordine (`InOrder()`) del BST stampa le chiavi in ordine. Se nella definizione 10.3 rendiamo le disuguaglianze strette otteniamo un BST che non accetta elementi duplicati. Vediamo ora le principali operazioni eseguibili su un BST.

Ricerca La struttura dei BST li rende candidati naturali per una ricerca efficiente. L'algoritmo 12 ha complessità $\mathcal{O}(h)$ dove h è l'altezza dell'albero. Nel caso ottimo, ossia di albero ben bilanciato, la complessità migliora a $\mathcal{O}(\log(n))$, mentre nel caso pessimo, ossia di albero degenerato a lista, peggiora a $\mathcal{O}(n)$.

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

Snippet 12: Ricerca in un BST

Minimo e massimo Cerchiamo l'elemento con chiave minima (massima), ossia quello più a sinistra (destra) del BST. Come prima, la complessità è $\mathcal{O}(h)$.

```
1  Min(T)
2    cur ← T
3    while cur.left ≠ NIL
4      cur ← cur.left
5    return cur
```

```
1  Min(T)
2    cur ← T
3    while cur.right ≠ NIL
4      cur ← cur.right
5    return cur
```

Successore Il successore di un elemento x di un elemento è l'elemento y con la più piccola chiave maggiore di di quella di x nel BST. Abbiamo due casi possibili di ricerca:

- Il sotto-albero destro di x non è vuoto, significando che il successore di x sarà il minimo di quel sotto-albero.
- Il sotto-albero destro di x è vuoto, significando che il successore di x sarà il progenitore più prossimo a x per cui x appare nel suo sotto-albero sinistro.

Lo pseudocodice della ricerca del successore è l'algoritmo 13. Anche in questo caso la complessità è $\mathcal{O}(h)$, sia nel caso ottimo che nel pessimo.

```
1 Successore(x)
2   if x.right ≠ NIL
3     return Min(x.right)
4   y ← x.p
5   while y ≠ NIL and y.right = x
6     x ← y
7     y ← y.p
8   return y
```

Snippet 13: Ricerca del successore in un BST

Inserimento L'inserimento di un nuovo elemento deve per forza rispettare la proprietà fondamentale del BST. Considereremo il caso di BST che non ammette duplicati. L'idea di fondo è eseguire una ricerca (che ovviamente fallirà) e inserire il nuovo nodo nel posto `NIL` trovato. La complessità sarà ancora $\mathcal{O}(h)$.

```
1 Inserisci(T, x)
2   pre ← NIL
3   cur ← T.root
4   while cur ≠ NIL      // ciclo di ricerca
5     pre ← cur
6     if x.key < cur.key
7       cur ← cur.key
8     else
9       cur ← cur.right
10  x.p ← pre            // inserimento
11  if pre = NIL
12    T.root ← x
13  else if x.key < pre.key
14    pre.left ← x
15  else
16    pre.right ← x
```

Snippet 14: Inserimento di un elemento in un BST

Cancellazione La strategia di cancellazione dipende dal numero di figli dell'elemento da rimuovere:

0. L'elemento non ha figli quindi è sufficiente eliminarlo ripulendo la memoria e i puntatori del padre.
1. L'elemento può essere sostituito dal suo unico figlio nel suo ruolo.
2. Sovrascriviamo il valore dell'elemento da rimuovere con quello del suo successore, eliminandolo. Il successore s di un elemento con due figli x infatti non ha mai figlio sinistro f: se ciò fosse vero si avrebbe `s.key < f.key < x.key`, ma ciò è impossibile per la definizione stessa di successore.

Anche in questo caso otteniamo complessità $\mathcal{O}(h)$. Un algoritmo che implementa il ragionamento sopra è il seguente.

```
1  Cancella(T, x)
2    // Individuiamo il nodo da cancellare
3    if x.left = NIL or x.right = NIL
4      da_canc ← = x
5    else
6      da_canc ← Successore(x)
7
8    // Individuiamo il sotto-albero da spostare
9    if da_canc.left ≠ NIL
10     sottoa ← da_canc.left
11   else
12     sottoa ← da_canc.right
13   if sottoa ≠ NIL
14     sottoa.p ← da_canc.p
15
16   // Correggiamo il valore del padre
17   if da_canc.p = NIL
18     T.root ← sottoa
19   else if da_canc = da_canc.p.left
20     da_canc.p.left ← sottoa
21   else
22     da_canc.p.right ← sottoa
23
24   // Copiamo il valore della chiave
25   if da_canc ≠ x
26     x.key ← da_canc.key
27   Free(da_canc)
```

Snippet 15: Rimozione di un elemento in un BST

10.7.2 Alberi RB

Abbiamo visto nella sezione 10.7.1 che tutte la complessità di tutte le operazioni dipende dall'altezza h dell'albero. Nel migliore dei casi $h = \log(n)$, nel peggiore $h = n$. È critico quindi riuscire a garantire che un BST non degeneri a seguito di inserimenti e cancellazioni. Si può dimostrare che l'altezza attesa di un BST è $\mathcal{O}(\log(n))$ se le chiavi hanno una distribuzione uniforme. A noi ci serve, però, un approccio deterministico e necessitiamo del concetto di albero bilanciato.

Intuitivamente, diciamo che un albero è bilanciato quando la distanza delle foglie dalla radice è limitata superiormente per tutte le foglie. Una definizione operativa analoga è stata data da Adelson-Velskii e Landis nel 1962:

Definizione 10.4 (Albero bilanciato)**.** Un albero si dice bilanciato se per ogni nodo le altezze dei due sotto-alberi differiscono al più di 1.

Adelson-Velskii e Landis proposero, insieme alla definizione, anche una modifica ai BST e ai metodi di accesso che garantisse il bilanciamento (alberi AVL). Gli alberi RB (RBT o alberi rosso-neri) sono una ottimizzazione della versione proposta da Adelson-Velskii e Landis che sacrifica parte del bilanciamento a favore di migliori prestazioni di inserimento e rimozione.

Definizione 10.5 (Albero RB)**.** Definiamo albero RB un BST i cui nodi sono dotati di un attributo aggiuntivo (colore) che può assumere due valori: rosso e nero. I nodi dello RBT soddisfano le seguenti proprietà:

1. Ogni nodo è o rosso o nero;
2. La radice è nera;
3. Le foglie sono nere;
4. I figli di un nodo rosso sono entrambi neri;
5. Per ogni nodo dell'albero, tutti i cammini dai suoi discendenti alle foglie contenute nei suoi sotto-alberi hanno lo stesso numero di nodi neri.

Chiamiamo, per comodità, altezza nera (black height) di un nodo x il valore $bh(x)$ pari al numero di nodi neri, escluso x se è il caso, nel percorso che ca da x alle foglie.

Per convenzione, consideriamo che i dati sono mantenuti unicamente nei nodi interni e le foglie sono tutte nulle rappresentate da `T.nil`, per definizione nero. Anche il padre della radice punta a `T.nil`.

Tutte le operazioni che non vanno a modificare la struttura dell'albero (ricerca, minimo, massimo ecc...) operano come se l'albero RB fosse un BST. Le operazioni di inserimento e cancellazione, invece, necessitano di "riparare" l'albero, ossia ridistribuire gli elementi in modo da non violare gli invarianti RB. Per avere una complessità accettabile è necessario che l'operazione di riparazione, o ribilanciamento, possa avvenire con modifiche solamente locali.

Teorema 10.1 (Buon bilanciamento)**.** *Un albero RB con n nodi interni ha altezza massima:*

$$h_{max} = 2\log(n+1) \tag{88}$$

Dimostrazione. Dimostriamo che un sotto-albero con radice x ha almeno $2^{bh(x)} - 1$ nodi per induzione sull'altezza del sotto-albero.

Caso base x è un foglia, il sotto-albero contiene almeno $2^{bh(x)} - 1 = 2^0 - 1 = 0$ nodi.

Passo Dato x, entrambi i suoi figli hanno altezza nera $bh(x)$ o $bh(x) - 1$. Dato che l'altezza dei figli è minore di quella di x per ipotesi induttiva i loro sotto-alberi hanno almeno $2^{bh(x)-1} - 1$ nodi interni. L'albero radicato in x contiene quindi almeno $2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 2 = 2^{bh(x)}$ nodi. Per la proprietà 4, almeno metà dei nodi su un qualunque percorso radice-foglia sono neri. L'altezza nera della radice è almeno $\frac{h}{2}$, per quanto detto prima il sotto-albero radicato in essa contiene almeno $2^{\frac{h}{2}} - 1$ nodi. Otteniamo quindi $n \geq 2^{\frac{h}{2}} - 1 \to h \leq 2\log(n+1)$.

□

Conseguenza del teorema sopra è che le operazioni di "query" saranno $\mathcal{O}(2\log(n+1))$. Se riesco a riparare le violazioni degli invarianti in maniera efficiente avrò anche che le modifiche all'albero saranno $\mathcal{O}(2\log(n+1))$.

Rotazione Definiamo prima un'operazione di vitale importanza per il ribilanciamento: la rotazione. Essa è un'operazione locale a due nodi di un BST che cambia il livello a cui sono situati due nodi senza violare le proprietà del BST.

```
1  LeftRotate(T, x)
2    y ← x.right
3    x.right ← y.left
4    if y.left ≠ T.nil
5      y.left.p ← x
6    y.p ← x.p
7    if x.p = T.nil
8      T.root ← y
9    else if x = x.p.left
10     x.p.left ← y
11   else
12     x.p.right ← y
13   y.left ← x
14   x.p ← y
```

Snippet 16: Algoritmo della rotazione verso sinistra

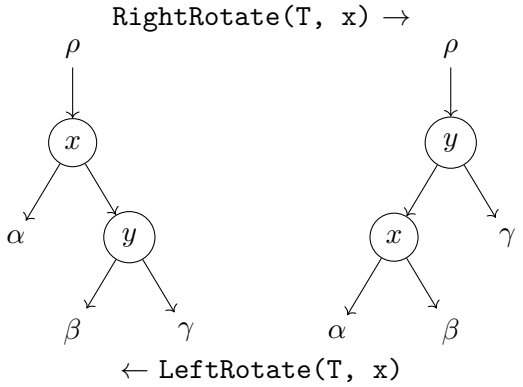

Figura 15: Schema di funzionamento delle rotazioni di un albero

Inserimento L'inserimento procede ad inserire il nuovo elemento come se l'albero fosse un semplice BST salvo assegnare `T.nil` al posto di `NIL` e colorare il nodo appena inserito di rosso. L'inserimento di un nuovo nodo rosso può causare la violazione delle proprietà 2 e 4 degli alberi RB. È compito di `RiparaRBInserisci(z)`, dato il nodo `z` inserito, riparare l'albero. Possono esserci diversi casi:

- Con `z.p` figlio sinistro del "nonno" di `z` possono esserci 3 casi a seconda del colore del "zio" e della posizione di `z` e del suo padre.
- Simmetricamente ci saranno 3 casi per quando il padre di `z` è figlio destro del "nonno".

Consideriamo solo i 3 casi del lato sinistro.

1. Lo zio `y` è rosso. Poiché `z.p.p` è nero, possiamo colorare di nero sia `z.p` sia `z` e `z.p.p` di rosso senza rompere altre proprietà. Ripetiamo il ciclo facendo sì che `z` ← `z.p.p`, risalendo due livelli.
2. Lo zio `y` è nero e `z` è figlio destro del padre `x`. In questo caso eseguiamo una `LeftRotate(T, x)` spostando la riparazione su `z` con `z.left = x` (caso 3).
3. Lo zio `y` è nero e `z` è figlio sinistro del padre `x`. Ci basterà scambiare i colori di `z.p.p = x.p` e `x` ed eseguire una `RightRotate(T, x.p)`.

```
1 RiparaRBInserisci(T, z)
2   while z.p.color = red
3     if z.p = z.p.p.left
4       y ← z.p.p.right
5       if y.color = red            // caso 1
6         z.p.color ← black
7         y.color ← black
8         z.p.p.color ← red
9         z ← z.p.p
10      else
11        if z = z.p.right          // caso 2
12          z ← z.p
13          LeftRotate(z)
14        z.p.color ← black         // caso 3
15        z.p.p.color ← red
16        RightRotate(z.p.p)
17    else
18      y ← z.p.p.left
19      if y.color = red            // caso 1
20        z.p.color ← black
21        y.color ← black
22        z.p.p.color ← red
23        z ← z.p.p
24      else
25        if z = z.p.left           // caso 2
26          z ← z.p
27          RightRotate(z)
28        z.p.color ← black         // caso 3
29        z.p.p.color ← red
30        LeftRotate(z.p.p)
31  T.root.color ← black
```

Snippet 17: Algoritmo di riparazione a seguito di un inserimento

Nei casi 2 e 3 la procedura `RiparaRBInserisci()` deve solo effettuare un cambio di colori locale e 2 o 1 rotazioni/e. Tutte queste operazioni sono $\mathcal{O}(k)$. Nel primo caso la routine continua analizzando il nodo padre del nodo corrente. Nel caso pessimo il ciclo itera un numero di volte pari a meta dell'altezza dell'albero. Ciò significa che l'intero inserimento prenderà al più $\mathcal{O}(\log(n))$.

Cancellazione La cancellazione procede a cancellare il nuovo elemento come se l'albero fosse un semplice BST salvo l'uso di `T.nil` al posto di `NIL` e invocare la procedura di ribilanciamento. Nel caso sia eliminato un nodo rosso è impossibile violare le proprietà e quindi non sarà necessario nessun cambiamento. Se il nodo eliminato invece è nero, la funzione `RiparaRBCancella(x)`, dato il nodo `x` presente al posto di quello cancellato `z` ribilancia l'albero. La funzione di riparazione gestisce 5 casi (come nell'inserimento, omettiamo i simmetrici 5 casi per quando `x` è figlio destro):

0. Caso banale: `x` è rosso. Semplicemente ricoloriamo `x` di nero.
1. `x` è nero con fratello `w` rosso. Scambiamo i colori di `w` e `w.p` e invochiamo una `LeftRotate(x.p)`. Ci riconduciamo ai casi 2,3 o 4.
2. `x` è nero con fratello `w` nero e nipoti entrambi neri. Coloriamo `w` di rosso e invochiamo `RiparaRBCancella(x.p)`.
3. `x` è nero con fratello `w` nero e nipote destro nero. Scambiamo di colore `w` e `w.left` ed invochiamo `RightRotate(w)` riconducendoci al caso 4.

4. `x` è nero con fratello `w` nero e nipote destro rosso. Assegnamo a `w` il colore di `w.p` e rendiamo `w.right` nero. Invochiamo quindi `LeftRotate(w.p)`.

Nei casi 0,1,3 e 4 della funzione `RiparaRBCancella()` vengono effettuati un numero costante di rotazioni e scambi di colore, rendendo il tutto $\mathcal{O}(k)$. L'unica chiamata ricorsiva avviene nel caso 2. Poiché ad ogni chiamata, nel caso pessimo, si risale di un livello verso la radice, verranno effettuate al massimo $\mathcal{O}(\log(n))$ chiamate. Detto ciò, possiamo affermare che la procedura di cancellazione sarà $\mathcal{O}(\log(n))$ come le altre operazioni di modifica su alberi RB.

10.7.3 Mucchi (heap)

Il mucchio, o heap, è una struttura dati ad albero nella quale la chiave del nodo padre è sempre maggiore (max-heap) di quella dei figli. Nessuna relazione sussiste tra le chiavi di due fratelli. È possibile definire anche la variante dove il nodo padre è minore rispetto ai figli, creando una min-heap. Poiché parliamo lavoriamo con alberi binari, anche la nostra heap sarà detta binaria.

Manteniamo le heaps come alberi binari quasi completi. In memoria li vedremo come array di elementi ordinati per livello. Terremo anche due attributi ausiliari: `heapsize`, il numero di elementi nello heap, e `length` che indica la lunghezza dell'array di supporto.

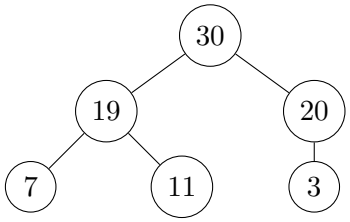

Figura 16: Esempio di heap salvata nel vettore $H = [30, 19, 20, 7, 11, 3]$

Quindi il primo elemento sarà la radice, i 2 elementi successivi saranno il primo livello, i 4 elementi il secondo e così via. Le foglie mancanti saranno, quindi, sempre quelle a occupare gli ultimi posti dell'array.

Gli heap, in particolare gli heap binary che affronteremo, trovano impiego per l'implementazione di code con priorità e l'ordinamento di vettori. Le operazioni su un max-heap che vedremo sono: texttttMax(), `Inserisci()`, `Cancella_Max()`, `Costruisci_Max_Heap()` e `Max_Heapify()`.

Operazioni Per implementare le primitive necessarie necessitiamo di una procedura di supporto: `Max_Heapify()`. Per semplicità, nel trattare queste operazioni considereremo gli indici come 1-based.

`Max_Heapify()` La funzione `Max_Heapify(A, n)` riceve come input un array è una posizione in esso. Si assume che i due sotto-alberi con radice in `Left(n) = 2n` e `Right(n) = 2n + 1` siano dei max-heap. La routine modificherà, quindi, `A` affinché l'albero radicato in `n` sia un max-heap. La procedura causerà la discesa del nuovo valore verso le foglie finché sarà maggiore dei figli. Nel caso pessimo la complessità sarà $\mathcal{O}(\log(n))$ in un heap con n elementi.

```
1  Max_Heapify(A, n)
2    l ← Left(n)
3    r ← Right(n)
4    if l ≤ A.heapsize and A[l] > A[n]
5      posmax ← l
6    else
7      posmax ← n
8    if r ≤ A.heapsize and A[r] > A[posmax]
9      posmax ← r
10   if posmax ≠ n
11     Swap(A[n], A[posmax])
```

```
12      Max_Heapify(A, posmax)
```

Snippet 18: Psudocodice di `Max_Heapify()`

`Costruisci_Max_Heap()` La routine trasforma un array in una heap chiamando iterativamente `Max_Heapify()`. Poiché uno heap binario è sempre alto $\lfloor \log(n) \rfloor$ per come l'abbiamo definito, il numero di nodi con altezza h sarà:

$$h \leq \frac{2^{\lfloor \log(n) \rfloor}}{2^h} \leq \frac{n}{2^h} \tag{89}$$

Calcolare `Max_Heapify()` per un nodo richiede al più $\mathcal{O}(h)$ spostamenti verso il basso. Possiamo quindi calcolare il costo complessivo sommando, per ogni livello, il costo di "mucchificare" ogni elemento di esso ottenendo:

$$\sum_{h=0}^{\lfloor \log(n) \rfloor} \frac{n}{2^h} \mathcal{O}(h) = n\mathcal{O}(\sum_{h=0}^{\lfloor \log(n) \rfloor} \frac{h}{2^h}) = n * k \tag{90}$$

Poiché $\sum_{h=0}^{\lfloor \log(n) \rfloor} \frac{h}{2^h}$ converge. Avremo quindi che `Costruisci_Max_Heap()` avrà costo $\mathcal{O}(n)$.

```
1 Costruisci_Max_Heap(A)
2   A.heapsize ← A.legnth
3   for i ← ⌊A.length/2⌋ downto 1
4     Max_Heapify(A, i)
```

Snippet 19: Pseudocodice di `Max_Heapify()`

`Max()` Operazione banale eseguita in tempo costante: `return A[1]`.

`Cancella_Max()` Estrarre l'elemento con la massima chiave costa $\mathcal{O}(\log(n))$. In media, però, il costo è inferiore. Si nota come la rimozione sia più efficiente rispetto a quella in un vettore ordinato, dove è $\mathcal{O}(n)$.

```
1 Cancella_Max(A)
2   if A.heapsize < 1
3     return ⊥
4   max ← A[1]
5   A[1] ← A[A.heapsize]
6   Max_Heapify(A, 1)
7   return max
```

Snippet 20: Pseudocodice di `Cancella_Max()`

`Inserisci()` Inserisce un elemento con una data chiave `key`. La complessità sarà $\mathcal{O}(\log(n))$.

```
1 Inserisci(A, key)
2   A.heapsize ← A.heapsize + 1
3   A[A.heapsize] ← key
4   i ← A.heapsize
5   while i > 1 and A[Parent(i)] < A[i]
6     Swap(A[Parent(i)], A[i])
7     i ← Parent(i)
```

Snippet 21: Pseudocodice di `Inserisci()`

Code con priorità Una coda con priorità è una struttura dati a coda in cui è possibile dare una priorità numerica agli elementi all'interno. Gli elementi con priorità maggiore verranno estratti sempre prima di elementi con priorità minore indipendentemente dall'ordine di inserimento. L'implementazione più comune di una coda con priorità è, come già detto prima, un max-heap nel quale la priorità di un elemento sarà la sua chiave. L'accodamento di nuovi elementi verrà effettuato tramite `Inserisci()` e la rimozione dell'elemento con massima priorità tramite `Cancella_Max()`. Quindi una coda con priorità così implementata ha costo $\mathcal{O}(\log(n))$ sia per enqueue che per dequeue. La costruzione di una coda a partire da un array tramite `Costruisci_Max_Heap()`, invece, ha costo lineare.

Ordinamento Ordinare un array in ordine crescente può essere fatto trovando il massimo tra i suoi elementi e posizionandolo sempre alla fine della slice già ordinata. Questo è il procedimento del `SelectionSort()`. Un ordinamento così strutturato ha complessità $\mathcal{O}(n^2)$. Se però prima di applicare rendiamo l'array un max-heap, riusciamo a rendere la complessità $\mathcal{O}(n \log(n))$ nel caso pessimo. Questo è il cosiddetto heapsort.

```
1 HeapSort(A)
2   Costruisci_Max_Heap(A)
3   for i ← A.length downto 2
4     Swap(A[1], A[i])
5     A.heapsize ← A.heapsize -1
6     Max_Heapify(A, 1)
```

Snippet 22: Pseudocodice di che implementa lo heapsort

Come detto prima, lo heapsort ha complessità $\mathcal{O}(n \log(n))$ in caso pessimo, il minimo teorico. Inoltre ha complessità spaziale costante, rendendolo più vantaggioso rispetto al mergesort. Nonostante ciò nel caso medio rimane più lento del quicksort. Come mergesort (la versione in-place) e quicksort, anche heapsort non è stabile.

10.8 Grafi

Il grafo è la struttura dati più naturale per rappresentare un insieme di oggetti legati da una generica relazione. Questa relazione è rappresentata da un insieme di coppie di oggetti, ordinate o meno.

Definizione 10.6 (Grafo)**.** Un grafo è una coppia $\mathcal{G} = (\mathbf{V}, \mathbf{E})$ con $\mathbf{V}$ un insieme di nodi (o vertici) e $\mathbf{E}$ un insieme di archi (detti anche lati).

Se un grafo ha $|\mathbf{V}|$ nodi, esso avrà al più $|\mathbf{V}|^2$ archi. Forniamo della nomenclatura utile che useremo:

- Due nodi collegati da un arco si dicono adiacenti.
- Un cammino tra due nodi v_1 e v_2 è un insieme di archi di cui il primo ha origine in v_1, l'ultimo termina in v_2 e ogni nodo compare almeno una volta sia come destinazione di un arco che come sorgente.
- Un grafo è detto orientato se la coppia di nodi che costituisce un arco è ordinata.
- Un grafo è connesso se esiste un percorso per coppia di nodi.
- Un grafo è completo (completamente connesso) se esiste un arco tra ogni coppia di nodi.
- Un percorso è un ciclo se il nodo di inizio e di fine coincidono. Il ciclo si dice orientato se segue la direzione degli archi.
- Un grafo privo di cicli è detto aciclico.

Sono possibili due strategie per rappresentare un grafo in memoria: liste o matrici di adiacenza. Le liste di adiacenza sono un vettore di liste lungo $|\mathbf{V}|$, indicizzato dai nomi dei nodi. Ogni lista contiene i nodi adiacenti all'indice della sua testa. La matrice di adiacenza è invece una matrice $|\mathbf{V}| \times |\mathbf{V}|$

di valori booleani con righe e colonne indicizzate dai nomi dei nodi. La cella alla riga i, colonna j contiene 1 se l'arco (v_i, v_j) è presente nel grafo e 0 altrimenti. Le liste sono una rappresentazione più compatta se il grafo è sparso ($|\mathbf{E}| \leq |\mathbf{V}|^2$) in quanto sono $\Theta(|\mathbf{V}| + |\mathbf{E}|)$ contro i $\Theta(|\mathbf{V}|^2)$ di una matrice di adiacenza. Le matrici di adiacenza possono eseguire il test di appartenenza in tempo costante. Le liste, invece, possono contare il numero di archi uscenti da un nodo linearmente rispetto al numero di questi, invece delle matrici che sono lineari rispetto al numero totale di nodi.

Per rappresentare grafi non orientati posso ottimizzare lo spazio sfruttandone le proprietà:

- La matrice di adiacenza di un grafo non orientato è simmetrica rispetto alla diagonale principale: posso stoccarne solo metà.
- Nelle liste di adiacenza possono stoccare solo uno dei due archi, raddoppiando però il tempo di ricerca per un nodo adiacente.

Operazioni sui grafi Le operazioni sui grafi sono tipicamente di ispezione, ossia visita in ampiezza o in profondità, oppure vanno a determinare determinate proprietà del grafo, come ad esempio trovare le componenti connesse, stabilire l'ordinamento topologico, trovare il percorso più breve tra due nodi e individuare cicli. Vediamone un po'.

Visita in ampiezza Chiamata anche Breadth First Search (BFS) è una strategia di visita che visita tutti i nodi di un grafo $\mathcal{G}$ a partire da una sorgente s visitando prima tutti i nodi con un cammino tra loro ed s lungo n prima di visitare quelli con un cammino lungo $n+1$. Per evitare di iterare all'infinito nel caso di cicli, useremo dei "colori" per segnare i nodi:

- I nodi bianchi devono essere ancora visitati.
- I nodi grigi sono stati visitati ma devono essere visitati quelli adiacenti.
- I nodi neri sono stati visitati e sono stati anche visitati gli adiacenti.

Memorizzeremo in una coda, inizializzata con la sola sorgente, i nodi ancora da visitare. Estrarremo un nodo dalla coda e ne visiteremo i vicini bianchi, colorandoli di grigio calcolandone la distanza e accodandoli affinché siano a loro volta visitati. Marchiamo quindi il nodo estratto di nero e iteriamo estraendo il successivo finché la coda non è vuota. La complessità totale di questo algoritmo sarà $\mathcal{O}(|\mathbf{V}| + |\mathbf{E}|)$.

```
1  VisitaAmpiezza(G, s)
2    for each n ∈ V \ {s}
3      n.color ← white
4      n.dist ← ∞
5    s.color ← grey
6    s.dist ← 0
7    Q ← ∅
8    Enqueue(Q, s)
9    while ¬IsEmpty(Q)
10     curr ← Dequeue(Q)
11     for each v ∈ curr.adiacenti
12       if v.color = white
13         v.color ← gray
14         v.dist ← curr.dist + 1
15         Enqueue(Q, v)
16       curr.color ← black
```

Snippet 23: Pseudocodice che implementa un BFS

La visita in ampiezza si può trasformare in un algoritmo di ricerca: basta inserire un controllo appena si sta per accodare un nuovo elemento.

Visita in profondità A differenza della visita in ampiezza, nella visita in profondità (Depth First Search o DFS) visitiamo prima i nodi adiacenti a quello dato, poi il nodo stesso seguendo i cammini fino in fondo. Lo pseudocodice di una DFS è uguale a quello di una BFS in qui però si sostituisce una la coda con una pila. La complessità è la stessa.

Componenti connesse È detta componente connessa di un grafo $\mathcal{G}$ un insieme **S** di nodi tali per cui esiste un cammino tra ogni coppia di essi, ma nessuno di essi è connesso a nodi non appartenenti a **S**. Individuare le componenti connesse in un grafo equivale ad etichettare i nodi con lo stesso valore se appartengono alla stessa componente. La complessità sarà $\mathcal{O}(|\mathbf{V}|)$.

```
1 for each v ∈ V
2   v.etichetta ← -1
3 eti ← 1
4 for each v ∈ V
5   if v.etichetta = -1
6     VisitaEdEtichetta(G, v, eti)
7     eti ← eti + 1
```

Snippet 24: Psudocodice per l'etichettatura delle componenti connesse

La funzione ausiliaria `VisitaEdEtichetta()` usata in 24 si comporta come una BFS/DFS che imposta a `eti` il campo `etichetta` del nodo visitato.

Ordinamento topologico L'ordinamento topologico è un valore utile da calcolare per un grafo orientato aciclico. Esso è una sequenza di nodi tale per cui nessun nodo compare prima di un suo predecessore. Il predecessore di un nodo è così definito:

Definizione 10.7 (Predecessore)**.** Dato un grafo orientato, il predecessore di un nodo v è un nodo u tale per cui esiste un cammino da u a v.

L'ordinamento topologico non è unico! Inoltre se un grafo non è connesso, le componenti non connesse possono essere ordinate in qualunque modo l'una rispetto all'altra.

```
1 // Come una DFS ma dopo aver colorato di nero un nodo lo
2 // inseriamo in testa alla lista L
3 VisitaProfOT(G, s, L)
4   s.color ← gray
5   for each v ∈ curr.adiacenti
6     if v.color = white
7       VisitaProfOT(G, v, L)
8   s.color ← black
9   PushFront(L, s)
10
11 OrdinamentoTopologico(G)
12   for each v ∈ V
13     v.color ← white
14   for each v ∈ V
15     if v.color = white
16       VisitaProfOT(G, s, L)
17   return L
```

Snippet 25: Pseudocodice per calcolare l'ordinamento topologico

Percorso più breve Per l'individuazione del percorso più breve vedremo l'algoritmo di Dijkstra. Esso trova, dato un grafo orientato e un suo nodo s, i percorsi più brevi da un nodo a qualunque altro. L'algoritmo funziona sia su di un grafo classico che su un grafo pesato. Il principio di funzionamento è il seguente:

1. Inserisco ogni $v \in \mathbf{V} \setminus \{s\}$ in un insieme $\mathbf{Q}$ dopo aver impostato l'attributo distanza a ∞ e il precedente a `NIL`;
2. Inserisco s in $\mathbf{Q}$ dopo aver impostato la distanza di s a 0 e il predecessore a `NIL`;
3. Fin quando $\mathbf{Q}$ non è vuoto, estraggo il nodo c con distanza minima e controllo per ogni adiacente a se hanno distanza maggiore di `c.dist + Peso(c, a)`. Se ciò accade, impostiamo `a.pred` $\leftarrow$ `c` e `a.dist` $\leftarrow$ `c.dist + Peso(c, a)`.

Rappresenteremo l'insieme $\mathbf{Q}$ come una coda con priorità basata su una min-heap dove la priorità è la distanza in modo da ottenere estrazione del minore in tempo costante e un cambio di priorità in $\mathcal{O}(\log(|\mathbf{V}|))$. La complessità totale risulterà $\mathcal{O}((|\mathbf{E}| + |\mathbf{V}|)\log(|\mathbf{V}|))$.

```
1  DijkstraQueue(G, s)
2  Q ← ∅
3  s.dist ← 0
4  for each v ∈ V
5    if v ≠ s
6      v.dist ← ∞
7      v.pred ← NIL
8      AccodaPriorita(Q, v, v.dist)
9  while Q ≠ ∅
10   u ← CancellaMin(Q)
11   for each v ∈ u.succ
12     ndis ← u.dist + Peso(u, v)
13     if v.dist > ndis
14       v.dist ← ndis
15       v.prev ← u
16       DecrementaPriorita(Q, v, ndis)
```

Snippet 26: Pseudocodice dell'algoritmo di Dijkstra

Individuazione di cicli Il nostro problema è il seguente: "Dato un grafo orientato per cui ogni nodo ha un solo successore, determinare, dato un nodo di partenza, se il cammino che parte da esso ha cicli". L'algoritmo migliore per risolvere questo problema è l'algoritmo di Floyd, o della lepre e della tartaruga. Illustriamo il funzionamento. Usiamo due riferimenti t ed l che spostiamo per ogni iterazione: nel caso di t lo spostiamo al successore mentre nel caso di l lo spostiamo al successore del successore. Entrambi partiranno dal nodo iniziale. Chiamiamo C la lunghezza del ciclo e T quella della coda. Quando t ha effettuato $T = qC + r$ passi, l sarà sicuramente all'interno del ciclo in posizione $qC + r \equiv_C r$ all'interno del ciclo. Dopo altre $C - r$ mosse, t si si troverà $C - r$ posizioni nel ciclo mentre l si troverà $r + 2(C - r) \equiv_C C - r$ posizioni nel ciclo. I due puntatori si sono incontrati! Il numero di mosse totale prima dell'incontro è $T + C - r = (q + 1)C$, un multiplo della lunghezza del ciclo. Facendo ripartire da capo t e muovendo l a partire dall'incontro un passo alla volta i due puntatori si rincontreranno all'inizio del ciclo (facilmente dimostrabile). La complessità temporale di questo algoritmo è $\Theta(2(T + C) - r)$, mentre quella spaziale è costante.

```
1  FloydLT(G, x)
2  t ← x.succ
3  l ← x.succ.succ
4  while l ≠ t
5    t ← t.succ
6    l ← l.succ.succ
7  T ← 0
8  t ← x
9  while l ≠ t
10   t ← t.succ
```

```
11    l ← l.succ
12    T ← T + 1
13 l ← t.succ
14 C ← 0
15 while l ≠ t
16    l ← l.succ
17    C ← C + 1
18 return T, C
```

Snippet 27: Paseudocodice dell'algoritmo di Floyd

A Costo delle istruzioni RAM in criterio di costo logaritmico

Istruzione	Costo
`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`	vedi `ADD`
`READ x`	$l(In) + l(x)$
`READ* x`	$l(In) + l(x) + l(M[x])$
`WRITE= x`	$l(x)$
`WRITE x`	$l(x) + l(M[x])$
`WRITE* x`	$l(x) + l(M[x]) + l(M[M[x]])$
`JUMP l`	1
`JGZ l`	$l(M[0])$
`JLZ l`	$l(M[0])$
`HALT`	1

B Riassunto sulle proprietà di chiusura dei linguaggi

	$\cup$	$\cap$	$\neg$	$\setminus$	$\star$	.
Star-Free	✓	✓	✓	✓	×	✓
Regolari	✓	✓	✓	✓	✓	✓
Non contestuali deterministici	×	×	✓	×	×	×
Non contestuali	✓	×	×	×	✓	✓
Dipendenti dal contesto	✓	✓	✓	✓	✓	✓
Ricorsivi	✓	✓	✓	✓	✓	✓
Ricorsivamente enumerabili	✓	✓	×	×	✓	✓

C Somme e approssimazioni notevoli

- Somme/serie notevoli:

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$
$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$
$$\sum_{i=0}^{n} z^i = \frac{1-z^{n+1}}{1-z}$$
$$\sum_{i=0}^{\infty} \frac{z^i}{i!} = e^z$$
$$\sum_{i=0}^{\infty} i\frac{z^i}{i!} = ze^z$$

- Approssimazioni di Stirling:

$$\ln(n!) \approx n\ln(n) - n \text{ per } n \to \infty$$
$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \text{ per } n \to \infty$$