Programmazione Algoritmica

Corso di problem solving & computational thinking

Tiziano Marzocchella

A.A. 2022/2023

Indice

L'obiettivo di questo corso è quello di imparare a risolvere problemi logici, acquisendo la capacità fondamentale di astrazione dei problemi.
L'esame è composto da:

- Una serie di 5 prove in itinere (compito in classe), ognuna con un punteggio massimo di 8pt, direttamente riportati al punteggio finale.
  Vengono prese in considerazione solo le migliori 4.
  Una prova è considerata **superata** se il punteggio totalizzato è di almeno 4pt.
  Sulla piattaforma di esercitazioni online è possibile trovare esercizi di preparazione.
  Se la somma dei punteggi delle suddette prove in itinere supera 18pt si viene direttamente ammessi all'esame orale.
- Un'esame orale

Materiale

- Google classroom
- Libro
- Checklist

1 Introduzione

1.1 Problem solving

Quando si parla di problem solving, si intende l'insieme di capacità che ci permettono di analizzare una situazione problematica e trovare una soluzione.
Per trovare soluzioni sono ottime le capacità di **astrazione** e **decomposizione**.

Fare astrazione di un problema significa trasformare la sua descrizione in linguaggio naturale, quindi ad "altissimo" livello, in una descrizione sistematica, nel nostro caso "comprensibile" da una macchina. Noi ci fermeremo al codice sorgente, ma il compilatore scende fino al più basso livello dei linguaggi informatici, il linguaggio macchina.

La **decomposizione** invece consiste nella suddivisione di un problema in sotto problemi, rendendo più facile il processo di soluzione del problema iniziale. Questo perché si può pensare alla soluzione di ogni sotto problema individualmente e unire tutte le "sotto-soluzioni" in un secondo momento.

Queste due tecniche sono ampiamente utilizzate nella realizzazione di **algoritmi**, ovvero un metodo per processare e trasformare informazioni.
Un algoritmo è così definito:

Definizione. Una sequenza finita di azioni da eseguire, esprimibili con una quantità finita di tempo e spazio, che trasforma uno o più valori come *input* in uno o più valori come *output*.

1.2 Ciclo di vita di un sistema software

Un qualsiasi sistema software ha un ciclo di vita formato da quattro fasi, ovvero *specifica*, *soluzione*, *codifica*, *esecuzione*.

1.2.1 Specifica

La fase di specifica serve a definire e capire il problema da risolvere. Dare una *specifica* di un problema significa definirlo in maniera più o meno formale, considerando ogni possibile caso. In generale, bisogna indicare lo stato iniziale del programma (**input**) e cosa ci si aspetta come risultato (**output**).

1.2.2 Soluzione

Per trovare una soluzione al problema, applichiamo le tecniche di *astrazione* e *decomposizione*. Dobbiamo quindi progettare una sequenza di azioni, ovvero tradurre il nostro ragionamento in un algoritmo. Può quindi tornare utile crearsi uno schema mentale della soluzione del problema. Un algoritmo va progettato pensando in alto livello. Poi, solo dopo aver fissato la soluzione più ottimale possibile, si può passare alla codifica. Un algoritmo ha le seguenti proprietà:

- Finitezza, sequenza di passi finita
- Non ambiguità, ogni singolo passo deve essere *non ambiguo*
- Eseguibilità, l'algoritmo deve poter essere eseguito da una macchina.

1.3 Limite Inferiore o Lower Bound

Il *lower bound* di un problema rappresenta il numero minimo di azioni necessarie per risolvere il problema. E.g.: Per sommare n numeri ho bisogno di $n-1$ somme, perché facendo $n-2$ o meno somme mi *perdo* uno o più numeri e la soluzione è certamente sbagliata.

2 Linguaggi di programmazione (LdP)

2.1 Nozioni di base

Un linguaggio di programmazione è uno strumento che utilizziamo per scrivere codice, che nel nostro caso sarà un codice più astratto del classico programma *C* o *Java*.
Ogni linguaggio di programmazione ha le due seguenti caratteristiche fondamentali:

- **Sintassi**, ovvero un insieme di regole che definiscono ciò che viene considerato codice corretto o *ben formato*, e che sono divise in due tipi:
  - **Grammatica** → è utile per definire come mettere insieme le parole.
    E.g. schema: <soggetto><verbo><complemento oggetto>
  - **Lessico** → è l'insieme delle parole che possono essere costruite a partire da un insieme di simboli atomici. Le parole *legali* sono il sottoinsieme del lessico che consideriamo accettate (E.g. il dizionario per la lingua italiana).
- **Semantica**, ovvero ciò che definisce se il codice scritto ha un significato.

2.1.1 Compilazione del codice

Per passare da un linguaggio di programmazione, di un qualsiasi livello, ad un linguaggio "comprensibile" ed eseguibile dalla macchina, utilizziamo il *compilatore* o *interprete*. Il *compilatore* "traduce" **l'intero sorgente** in un eseguibile, mentre l'*interprete* "traduce" e esegue il sorgente **istruzione per istruzione**.

Questi due strumenti hanno un funzionamento ben definito, organizzato in diverse fasi. Queste fasi sono:

- Analisi lessicale (**scanner**), controlla che tutti i termini utilizzati siano previste dal LdP utilizzato.
- Analisi sintattica (**parser**), controlla che le parole siano combinate in modo corretto.
- Analisi semantica (**type check**), controllo sui tipi (?)
- Generazione codice oggetto, genera l'eseguibile
- Linking

2.1.2 Esecuzione del codice

- Architettura di Von-Neumann
- Ciclo Fetch-Execute

2.2 Paradigmi di programmazione

I linguaggi di programmazione possono essere suddivisi in due grandi famiglie, quella dei linguaggi *imperativi* (ad es. Java, C, JavaScript) oppure quella dei linguaggi *dichiarativi*. Le diverse tipologie di linguaggi contenuti in queste famiglie rappresentano diversi paradigmi di programmazione. Al giorno d'oggi però quasi tutti i moderni linguaggi di programmazione sono **multi-paradigma**, ovvero fanno uso di più paradigmi contemporaneamente. Noi vedremo linguaggi *procedurali*, *funzionali*, e *orientati ad oggetti*

3 Definizione di linguaggio

In questa sezione daremo una definizione formale di linguaggio di programmazione, partendo dalle basi.

3.1 Alfabeto

Alla base di questa definizione troviamo gli *alfabeti*, ovvero un insieme (matematico) composto da caratteri. I caratteri rappresentano la nostra unità di informazione nell'alfabeto. Avremo quindi bisogno di:

- Definire un *alfabeto*
- Definire l'operazione di *concatenazione* fra caratteri dell'alfabeto
- Definire una *stringa*

Cominciamo dall'alfabeto

Definizione. Alfabeto: Insieme **finito** e **non vuoto** di *simboli* (caratteri).

3.2 Stringa

Bene, avendo definito il nostro insieme di caratteri, possiamo passare ora alla definizione di concatenazione tra caratteri, che ci tornerà utile per definire una *stringa*.

Definizione. Concatenazione: Dati due caratteri $a, b \in A$, $\quad a \cdot b = ab$.

Adesso abbiamo tutte le conoscenze necessarie per definire formalmente una stringa.

Definizione. Stringa: Risultato della concatenazione di tutti gli elementi di un insieme finito e non vuoto, sottoinsieme di un certo alfabeto.

La stringa, ha alcune proprietà particolari:

- Lunghezza della stringa: $|s| =$ # di simboli nella stringa
- Concatenazione fra due stringhe x e y: $x \cdot y =$ tutti i simboli di x seguiti da tutti i simboli di y
- Esponenziale di una stringa

$$\begin{aligned} x^0 &= \epsilon \\ x^1 &= x \cdot x^0 = x \cdot \epsilon \\ &\vdots \\ x^n &= x \cdot x^{n-1} \end{aligned} \qquad \begin{aligned} x^0 &= \epsilon \\ x^1 &= ab \\ x^2 &= abab \\ x^3 &= ababab \end{aligned}$$

- Prefisso di una stringa: $pre(n, s) =$ i primi n caratteri della stringa s
- Suffisso di una stringa: $suf(n, s) =$ gli ultimi n caratteri della stringa s

3.3 Chiusura di Kleene

Per dare una definizione formale di linguaggio di programmazione abbiamo ancora bisogno di ulteriore livello di astrazione. Per prima cosa definiamo l'insieme di tutte le stringhe generabili a partire da un certo alfabeto A.

Definizione. Chiusura di Kleene: L'insieme che contiene tutte le possibili stringhe di qualsiasi lunghezza che si possono formare concatenando i simboli dell'alfabeto a cui viene applicata.

$$\text{Se } A = \{0, 1\} \text{ allora } A^* = \text{tutti i numeri binari.}$$

Esiste anche una variante della chiusura di Kleene, detta chiusura positiva, perché esclude la stringa vuota ϵ.

Definizione. Chiusura positiva di Kleene: Tutte le stringhe di A^* di lunghezza ≥ 1, ovvero tutte le stringhe di A^* tolta ϵ.

3.4 Linguaggio

Avendo definito tutti gli oggetti fondamentali alla base dei LdP, possiamo passare ad una definizione formale di linguaggio, abbastanza generica:

Definizione. Linguaggio: Insieme di stringhe *legali* formate a partire da un alfabeto secondo le regole di una **grammatica**.

Partendo da queste definizioni possiamo fare alcune osservazioni *interessanti*.
Il linguaggio vuoto è un insieme che non contiene alcun elemento.

$$\varnothing = \text{linguaggio vuoto}$$

Il linguaggio $\{\epsilon\}$ è il linguaggio contenente solo la stringa vuota.

$$\{\epsilon\} = \text{linguaggio contenente solo la stringa vuota}$$

Ogni linguaggio L è sottoinsieme della chiusura di Kleene su un alfabeto A.

$$L \subseteq A^*$$

4 Sintassi

4.1 Nozioni di base

Generalmente i linguaggi moderni sono sottoinsiemi di ASCII* e i programmi di tali linguaggi non sono altro che stringhe di caratteri ASCII. I LdP però non sono l'insieme completo ASCII*, ma solo un sotto insieme, che viene individuato secondo delle specifiche regole di sintassi.
Per definire quali stringhe appartengono al nostro linguaggio L, potremmo enumerarle una ad una, ma questo processo sarebbe impraticabile per insiemi infiniti, come il nostro L. Quindi, si può procedere in 3 modi diversi:

- Seguendo il metodo *generativo*, possiamo definire una *grammatica* in grado di poter generare l'insieme di stringhe legali
- Seguendo il metodo *riconoscitivo*, potremmo creare un *automa* in grado di riconoscere le stringhe legali
- Seguendo il metodo *algebrico*, potremmo definire l'insieme delle stringhe legali a partire dalle soluzioni di un sistema di equazioni algebriche

Noi utilizzeremo il metodo **generativo**.

4.2 Grammatica

Definizione. Grammatica: Una quadrupla $G = \langle N, \Sigma, P, S \rangle$ dove:

- $N \neq \varnothing$ è un insieme di simboli *non terminali*
- Σ è un alfabeto di simboli *terminali*
- $P \subseteq (N \cup \Sigma)^+ \times (N \cup \Sigma)^*$ è l'insieme delle produzioni, dove ogni produzione è una coppia
- $S \in N$ è il simbolo iniziale (starting symbol)

Esistono alcune convenzioni sulla scrittura delle grammatiche:

- $A, B, \ldots \in N$
- $a, b, \ldots \in \Sigma$
- $\alpha, \beta, \ldots \in (N \cup \Sigma)^*$
- $x, y, \ldots \in \Sigma^*$
- Per le produzioni: (α, β) oppure $\alpha ::= \beta$

4.2.1 Esempio di grammatica

Esempio. Grammatica delle espressioni aritmetiche
Considerati:

- gli operatori binari: + e *
- le parentesi ()
- i numeri naturali (operandi)

Possiamo definire la grammatica delle espressioni aritmetiche con le seguenti produzioni:

1. (E, N)
2. $(E, (E))$
3. $(E, E + E)$
4. $(E, E * E)$
5. $(N, 0)$
6. $(N, 1)$
7. $\vdots$
8. $(N, 9)$

La notazione (A, B) è una produzione che indica la sostituzione di A con B.
A prende il nome di *categoria sintattica* o *simbolo non terminale*.
Proviamo ad esprimere l'espressione $5 * 3$ utilizzando le produzioni che abbiamo appena definito:

$$\begin{aligned} E &= E * E \\ &= N * E \\ &= N * N \\ &= 5 * N \\ &= 5 * 3 \end{aligned}$$

4.3 Le derivazioni

Una derivazione è il processo passo-passo che identifica le "frasi" legali in una grammatica.
Quindi, data la grammatica: $G = \langle N, \Sigma, P, S \rangle$ e

- $\delta \in (N \cup \Sigma)^+$, una stringa di caratteri terminali e non terminali **non vuota**
- $\gamma \in (N \cup \Sigma)^*$, una stringa di caratteri terminali e non terminali

Definizione. Derivato immediato

$$\gamma \to \delta$$

δ è derivato immediato di γ se e solo se δ si ottiene da γ applicando una sola produzione.

Definizione. Derivato

$$\gamma \to^* \delta$$

δ è derivato di γ se δ si ottiene da γ applicando un numero qualsiasi di produzioni.

Esistono due tipi di derivazioni canoniche:

- Canonica destra, secondo la quale si espande sempre il simbolo non terminale più a destra
- Canonica sinistra, secondo la quale si espande sempre il simbolo non terminale più a sinistra

4.4 Forma normale di Backus-Naur (BNF)

Definizione. Una notazione standard più semplificata.

Esempio. Un esempio di grammatica in BNF

- $E ::= E + E \mid E - E \mid E * E \mid -E \mid (E) \mid V$
- $V ::= x \mid y \mid z$
- $I ::= 0 \mid \ldots \mid 9$

4.5 Classificazione delle grammatiche

Le grammatiche possono essere classificate in base alla loro espressività (Gerarchia di Chomsky). I LdP moderni sono grammatiche *libere dal contesto*, ovvero una delle definizioni date da Chomsky, più restrittiva della definizione generale.

Definizione. Grammatiche generali: $(\alpha, \beta), \alpha \in (N \cup \Sigma)^+, \beta \in (N \cup \Sigma)^*$

Definizione. Grammatiche libere dal contesto: $(a, \beta), \beta \in (N \cup \Sigma)^*$

4.6 Parse tree (Alberi di derivazione)

Le derivazioni si possono rappresentare attraverso una struttura matematica ad albero discreta dove:

- il nodo radice è etichettato con S
- i nodi sono etichettati con simboli non terminali
- le foglie sono i simboli terminali

La frontiera di un albero di derivazione, ovvero la sequenza delle sue foglie, rappresenta la sequenza di derivazioni utilizzate.

4.7 Grammatiche ambigue

Una grammatica è ambigua se esiste almeno una stringa del linguaggio che può essere derivata attraverso due sequenze di derivazioni diverse. Per eliminare le ambiguità sarà necessario rendere la grammatica più complessa, in modo da creare una sorta di precedenza tra le derivazioni.

4.8 Analisi sintattica

L'analizzatore sintattico (*parser*) utilizza le grammatiche non ambigue per determinare correttamente la precedenza tra operatori.

Quindi a partire da una grammatica genera una struttura ad albero che identifica le sequenze di produzioni possibili e definisce una precedenza tra queste ultime.

4.9 Alberi di sintassi astratta

Sono una semplificazione degli alberi di derivazione che omettono le informazioni ridondanti delle sequenza di simboli non terminali.

4.10 Albero di sintassi

La struttura ad albero viene utilizzata anche per rappresentare in modo formale le stringhe del linguaggio che sono sintatticamente corrette.

4.11 Definizione formale di linguaggio utilizzando le grammatiche

Un linguaggio L generato a partire da una grammatica G è definito come segue.

Definizione. Linguaggio: L'insieme delle stringhe di caratteri terminali ottenuti applicando un numero qualsiasi di produzioni a partire dal simbolo iniziale.

$$L(G) = \{w \mid w \in \Sigma^* \wedge S \rightarrow^* w\}$$

Due grammatiche sono equivalenti se generano lo stesso linguaggio.

$$L(G) = L(G')$$

5 Semantica

La *semantica* ha il compito di determinare il significato dei programmi sintatticamente corretti, ed è definita come una funzione che associa alle stringhe del linguaggio un significato.

$$f : L \to \mathbb{N}$$

- L è il dominio sintattico, ovvero l'insieme delle frasi legali del linguaggio.
- $\mathbb{N}$ è il dominio semantico, ovvero l'interpretazione delle stringhe.

Le stringhe del dominio sintattico non hanno significato prima di applicare la f.

Come si può determinare questa funzione quindi?

Se L fosse un insieme finito, basterebbe una sorta di tabella di corrispondenza. Ma nel nostro caso l'insieme delle stringhe generabili da una grammatica è infinito.

Utilizzeremo quindi il concetto di *composizionalità*, che in parole povere da un significato agli elementi atomici e delle regole per comporre questi elementi, che daremo nel dominio semantico. Assegnando un significato ai simboli del linguaggio e definiamo un modo per comporre i significati si riesce a definire la semantica attraverso elementi finiti, che però sono in grado di interpretare le infinite frasi del linguaggio.

5.1 Astrazione

Alla base dell'analisi semantica troviamo una serie di definizioni che ci premettono di astrarre i concetti di programmazione e di esecuzione del codice.

Definizione. Identificatori: Sono stringhe che danno un nome significativo al dato che rappresentano. Seguono alcune convenzioni e non possono cominciare con un numero.

Definizione. Variabili: Sono identificatori che individuano delle *locazioni di memoria* il cui contenuto può essere variato nel corso del programma.

Definizione. Costanti: Sono identificatori che individuano dei valori che non cambiano per tutto il corso del programma.

Definizione. Assegnamento: Un operatore che serve per scrivere un valore all'interno di un locazione di memoria. Prima valuta la parte destra (*rvalue*) e poi scrive il risultato della valutazione nella parte sinistra (*lvalue*).

Definizione. Macchina astratta: Per analizzare al meglio la **bontà** di una soluzione definiamo una *macchina astratta*, ovvero l'astrazione di un'architettura generica. Il modello a cui faremo riferimento è quello di Von-Neumann che descrive: *memoria*, *controllo di flusso* e *CPU*. Definiamo inoltre un meccanismo di esecuzione chiamato ciclo *fetch-execute*.

5.2 Semantica statica

La semantica statica si occupa di analizzare i tipi all'interno del programma da eseguire. L'analizzatore semantico riceve in input un albero di sintassi astratta, ovvero un programma corretto per le regole della grammatica utilizzata.

In analisi statica ho bisogno quindi di distinguere variabili, costanti e tipi di dato, per farlo utilizzo la funzione **ambiente statico**

$$\Delta : Id \cup Val \to T \cup TLoc$$

Nelle espressioni le stringhe di cui tratteremo vengono considerate come una sequenza di *letterali*, ovvero costanti o valori, composti mediante operatori unari o binari.

Per arrivare alla conclusione che una stringa del programma è semanticamente ben formata serve un meccanismo finito in grado di considerare le infinite espressioni generabili secondo le regole di sintassi. Sfruttiamo il principio di composizionalità e definiamo la semantica delle produzioni della grammatica che generano le nostre espressioni. In questo modo possiamo comporre la semantica per tutte le possibili espressioni generabili.

5.2.1 Espressioni

Un'espressione E è ben formata se partendo dall'ambiente statico Δ posso associare ad E il tipo del valore che rappresenta.

Assiomi

- (A1): $\varnothing \vdash_e i : Int$
- (A2): $\varnothing \vdash_e d : Double$
- (A3): $\varnothing \vdash_e b : Bool$
- (A4): $\varnothing \vdash_e s : String$

Regole di inferenza

- (R1)
$$\frac{\Delta(Id) = \tau \vee \Delta(Id) = \tau Loc}{\Delta \vdash_e Id : \tau}$$
- (R2)
$$\frac{\Delta \vdash_e E_1 : \tau,\ uop : \tau_1 \to \tau}{\Delta \vdash_e uop\, E_1 \ : \tau}$$
- (R3)
$$\frac{\Delta \vdash_e E_1 : \tau,\ \Delta \vdash_e E_2 : \tau,\ bop : \tau_1 \times \tau_2 \to \tau}{\Delta \vdash_e E_1\, bop\, E_2 \ : \tau}$$
- (R4)
$$\frac{\Delta \vdash_e E : \tau}{\Delta \vdash_e (E) : \tau}$$

5.2.2 Comandi

Definizione. Comando ben formato: Un comando C è ben formato se in un assegnamento riesco ad associare un tipo all'identificatore e all'espressione sulla destra

Assiomi

- (A5): $\varnothing \vdash_c nil$

Regole di inferenza

- (R5)
$$\frac{\Delta \vdash_e E : \tau,\ \Delta(Id) = \tau Loc}{\Delta \vdash_c Id = E}$$
- (R6)
$$\frac{\Delta \vdash_c C_1,\ \Delta \vdash_c C_2}{\Delta \vdash_c C; C}$$
- (R7)
$$\frac{\Delta \vdash_e E : Bool,\ \Delta \vdash_c C}{\Delta \vdash_c \ \text{if}(E)\{C\}}$$
- (R8)
$$\frac{\Delta \vdash_e E : Bool,\ \Delta \vdash_c C}{\Delta \vdash_c \ \text{while}(E)\{C\}}$$
- (R9)
$$\frac{\Delta \vdash_d D : \Delta',\ \Delta[\Delta'] \vdash_c C}{\Delta \vdash_c D; C}$$

5.2.3 Dichiarazioni

Una dichiarazione D è ben formata se rispetta i tipi e mi permette di produrre un nuovo ambiente statico Δ'

Assiomi

- (A6)
$$\varnothing \vdash_d nil : \varnothing$$

Regole di inferenza

- (R10)
$$\frac{\Delta \vdash_e E : \tau,\ T == \tau}{\Delta \vdash_d \text{const } Id : T = E}$$
- (R11)
$$\frac{\Delta \vdash_e E : \tau,\ T == \tau}{\Delta \vdash_d \text{var } Id : T = E\ :\ [(Id, \tau Loc)]}$$
- (R12)
$$\frac{\Delta \vdash_d D_1 : \Delta_1,\ \Delta[\Delta_1] \vdash_d D_2 : \Delta_2}{\Delta \vdash_d D_1; D_2 : \Delta[\Delta_1[\Delta_2]]}$$

5.3 Semantica dinamica

La semantica dinamica definisce una sequenza di transizioni tra stati della macchina astratta. La semantica dinamica simula l'esecuzione di un programma ignorando i tipi perché sono già verificati dalla semantica statica, lavorando quindi su valori.

In semantica dinamica un programma è considerato come una serie di istruzioni atomiche che modificano gli stati, ovvero una rappresentazione di un istante di tempo durante l'esecuzione. Uno stato è l'insieme delle istruzioni ancora da eseguire associato ad una istanza dell'ambiente e della memoria.

Definizione. Ambiente: $\rho : Id \to Loc \cup Val$

Definizione. Memoria: $\sigma : Loc \to Val$

L'analisi dinamica segue un processo ben preciso che partendo da uno stato iniziale "calcola" il risultato finale dell'esecuzione di un certo codice. Ogni esecuzione è articolata in 3 fasi:

- L'esecuzione comincia dallo *stato iniziale* dove tutte le istruzioni del programma sono ancora da eseguire.
- Per passare da uno stato all'altro si applicano le regole di semantica statica definite per **comandi** (C), **espressioni** (E) e **dichiarazioni** (D).
- Alla fine della *simulazione* il risultato del programma si troverà nello stato finale.

5.3.1 Espressioni

La semantica dinamica delle espressioni simula la valutazione delle espressioni all'interno del programma.

$$\langle E, \rho, \sigma\rangle \longrightarrow_e \langle E', \rho', \sigma'\rangle$$

$$Eval(E, \rho, \sigma) = v \in Val \iff \langle E, \rho, \sigma\rangle \longrightarrow_e^* v$$

Regole di inferenza

- (id1)
$$\frac{\rho(Id) = v \vee (\rho(Id) = L \in Loc \wedge \sigma(L) = v)}{\langle Id, \rho, \sigma\rangle \longrightarrow_e v}$$
- (uop1)
$$\frac{\langle E, \rho, \sigma\rangle \longrightarrow_e \langle E', \rho, \sigma\rangle}{\langle uop\, E, \rho, \sigma\rangle \longrightarrow_e \langle uop\, E', \rho, \sigma\rangle}$$
- (uop2)
$$\langle E, \rho, \sigma\rangle \longrightarrow_e v' = \text{uop } v$$
- (bop1)
$$\frac{\langle E_1, \rho, \sigma\rangle \longrightarrow_e \langle E_1', \rho, \sigma\rangle}{\langle E_1\, bop\, E_2, \rho, \sigma\rangle \longrightarrow_e \langle E_1'\, bop\, E_2, \rho, \sigma\rangle}$$
- (bop2)
$$\frac{\langle E_2, \rho, \sigma\rangle \longrightarrow_e \langle E_2', \rho, \sigma\rangle}{\langle v_1\, bop\, E_2, \rho, \sigma\rangle \longrightarrow_e \langle v_1\, bop\, E_2', \rho, \sigma\rangle}$$
- (bop3)
$$\langle v_1\, bop\, v_2, \rho, \sigma\rangle \longrightarrow_e v = v_1 \text{ bop } v_2$$

5.3.2 Comandi

$$\langle C, \rho, \sigma\rangle \longrightarrow_c \langle C', \rho', \sigma'\rangle,\ Exec(C, \rho, \sigma) = \sigma' \iff \langle C, \rho, \sigma\rangle \longrightarrow_c^* \sigma'$$

Regole di inferenza

- (id2)
$$\frac{\langle E, \rho, \sigma\rangle \longrightarrow_c^* v}{\langle Id = E, \rho, \sigma\rangle \longrightarrow_c \langle Id = v, \rho, \sigma\rangle}$$
- (id3)
$$\langle Id = E, \rho, \sigma\rangle \longrightarrow_c \sigma[\rho(Id) = v]$$
- (seq1)
$$\frac{\langle C_1, \rho, \sigma\rangle \longrightarrow_c \langle C_1', \rho, \sigma\rangle}{\langle C_1; C_2, \rho, \sigma\rangle \longrightarrow_c \langle C_1'; C_2, \rho, \sigma'\rangle}$$
- (seq2)
$$\frac{\langle C_1, \rho, \sigma\rangle \longrightarrow_c \sigma'}{\langle C_1; C_2, \rho, \sigma\rangle \longrightarrow_c \langle C_2, \rho, \sigma'\rangle}$$
- (if1)
$$\frac{\langle E, \rho, \sigma\rangle \longrightarrow_e^* true}{\langle \text{if}(E)\{C_1\} \text{ else } \{C_2\}, \rho, \sigma\rangle \longrightarrow_c \langle C_1, \rho, \sigma\rangle}$$
- (if2)
$$\frac{\langle E, \rho, \sigma\rangle \longrightarrow_e^* false}{\langle \text{if}(E)\{C_1\} \text{ else } \{C_2\}, \rho, \sigma\rangle \longrightarrow_c \langle C_2, \rho, \sigma\rangle}$$
- (rep1)
$$\frac{\langle E, \rho, \sigma\rangle \longrightarrow_e^* true}{\langle \text{while(E)}\{\text{C}\}, \rho, \sigma\rangle \longrightarrow_c \langle C; \text{while}(E)\{C\}, \rho, \sigma\rangle}$$
- ($rep2$)
$$\frac{\langle E, \rho, \sigma\rangle \longrightarrow_e^* false}{\langle \text{while(E)}\{\text{C}\}, \rho, \sigma\rangle \longrightarrow_c \sigma'}$$

5.3.3 Dichiarazioni

$$\langle D, \rho, \sigma\rangle \longrightarrow_e \langle D', \rho', \sigma'\rangle,\ Elab(D, \rho, \sigma) = \langle \rho', \sigma'\rangle \iff \langle D, \rho, \sigma\rangle \longrightarrow_d^* (\rho', \sigma')$$

Regole di inferenza

- (const1)
$$\frac{\langle E, \rho, \sigma\rangle \longrightarrow_e^* v}{\langle \text{const}Id : T = E, \rho, \sigma\rangle \longrightarrow_d \langle [(Id, v)], \sigma\rangle}$$
- (var1)
$$\frac{\langle E, \rho, \sigma\rangle \longrightarrow_e^* v}{\langle \text{var}Id : T = E, \rho, \sigma\rangle \longrightarrow_d \langle [(Id, \text{new}L)], [(L, v)]\rangle}$$
- (dd1)
$$\frac{\langle D_1, \rho, \sigma\rangle \longrightarrow_d^* \langle D_1', \rho', \sigma'\rangle}{\langle D_1; D_2, \rho, \sigma\rangle \longrightarrow_d \langle D_1'; D_2, \rho', \sigma'\rangle}$$

- (dd2)

$$\frac{\langle D_2, \rho[\rho_1], \sigma\rangle \longrightarrow_d^* \langle D_2', \rho[\rho_1]', \sigma'\rangle}{\langle \rho_1; D_2, \rho[\rho_1], \sigma\rangle \longrightarrow_d \langle \rho_1'; D_2, \rho[\rho_1]', \sigma'\rangle}$$

- (dd3)

$$\langle \rho_1, \rho_2, \rho, \sigma\rangle \longrightarrow_d \langle \rho_1[\rho_2], \sigma\rangle$$

6 Calcolo della complessità

6.1 Limiti asintotici

Usiamo la notazione asintotica per dare un limite ad una funzione $f(n)$ a meno di un fattore costante c.

Definizione. Limite superiore asintotico
Se esistono $c > 0, n_0 \geq 0$ t.c. $f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$,
$g(n)$ è detto limite superiore asintotico di $f(n)$.

$$f(n) = O\left(g(n)\right)$$

Diciamo che $f(n)$ è **O-grande** di $g(n)$.

Possiamo anche cercare il limite non stretto,
Per ogni $c > 0$ esiste un n_0 tale che

$$f(n) < c \cdot g(n) \quad \forall n \geq n_0$$

Scriviamo $f(n) = o\left(g(n)\right)$ che si legge f è **o-piccolo** di g.

Definizione. Limite inferiore asintotico
Se esistono $c > 0, n_0 \geq 0$ t.c. $f(n) \geq c \cdot g(n) \quad \forall n \geq n_0$,
$g(n)$ è detto limite inferiore asintotico di $f(n)$.

$$f(n) = \omega\left(g(n)\right)$$

Diciamo che $f(n)$ è **Omega-grande** di $g(n)$.

Possiamo anche cercare il limite non stretto,
Per ogni $c > 0$ esiste un n_0 tale che

$$f(n) > c \cdot g(n) \quad \forall n \geq n_0$$

Scriviamo $f(n) = \omega\left(g(n)\right)$ che si legge f è **omega-piccolo** di g.

Definizione. Se $f(n) = O\left(g(n)\right)$ e $f(n) = \Omega\left(g(n)\right)$,
$g(n)$ è detto limite asintotico stretto di $f(n)$.

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

Diciamo che $f(n)$ è **Theta** di $g(n)$.

Osservazione. $f(n) = O\left(g(n)\right)$ e $f(n) = \Omega\left(g(n)\right)$ implica che,
Esistono $c_1 > 0, c_2 > 0$ e $n_0 > 0$ tali che

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \forall n \geq n_0$$

Teoremi

- $f(n) = O\left(g(n)\right)$ se e solo se $g(n) = \Omega\left(f(n)\right)$
- Se $f_1(n) = O\left(f_2(n)\right)$ e $f_2(n) = O\left(f_3(n)\right)$ allora $f_1(n) = O\left(f_3(n)\right)$

- Se $f_1(n) = \Omega(f_2(n))$ e $f_2(n) = \Omega(f_3(n))$ allora $f_1(n) = \Omega(f_3(n))$
- Se $f_1(n) = \Theta(f_2(n))$ e $f_2(n) = \Theta(f_3(n))$ allora $f_1(n) = \Theta(f_3(n))$
- Se $f_1(n) = O(g_1(n))$ e $f_2(n) = O(g_2(n))$ allora

$$O(f_1(n) + f_2(n)) = O(max\{g_1(n), g_2(n)\})$$

- Se $f(n)$ è un polinomio di grado d allora

$$f(n) = \Theta\left(n^d\right)$$

Nell analizzare un algorimo utilizzeremo il limite inferiore asintotico per dare un limite di tempo minimo per un qualunque input di dimensione N.

Viceversa, utilizzeremo il limite superiore asintotico per dare un limite di tempo massimo per un qualunque input di dimensione N.

6.2 Costo di funzioni ricorsive

Per calcolare il costo di funzioni ricorsive utilizziamo il concetto di **relazione di ricorrenza**.

Facciamo sempre riferimento al concetto di induzione e definiamo:

- Costo base: $h(1) = 1$
- Costo per un N generico: $h(N) = 2h(N-1) + 1, \quad N > 1$

Se sviluppiamo la formula per il costo generico

$$\begin{aligned} h(N) &= 2h(N-1) + 1 = 2(2h(N-2) + 1) + 1 \\ &= 4h(N-2) + 3 = 4(2h(N-3) + 1) + 3 \\ &= 8h(N-3) + 7 \\ &= 2^{N-1}h(N - (N-1)) + 2^{N-1} - 1 \\ &= 2^{N-1}h(1) + 2^{N-1} - 1 \\ &= 2^N - 1 \end{aligned}$$

riusciamo ad esprimere la complessità eliminando la definizione ricorsiva.

6.3 Divide et impera

$$T(n) = \begin{cases} 0 & true \\ 0 & false \end{cases} \tag{1}$$

6.4 Master theorem

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

Caso 1. Se $f(n)$ cresce polinomialmente più lentamente di $n^{\log_b a}$

$$f(n) = O\left(n^{\log_b a - \epsilon}\right)$$

allora la soluzione è

$$T(n) = \Theta\left(n^{\log_b a}\right)$$

Caso 2. Se $f(n)$ è Θ-grande di $n^{\log_b a} \lg^k n$ per qualche costante $k \geq 0$

$$f(n) = \Theta\left(n^{\log_b a} \lg^k n\right)$$

allora la soluzione è

$$T(n) = \Theta\left(n^{\log_b a} \lg^{k+1} n\right)$$

Caso 3. Se $f(n)$ cresce polinomialmente più velocemente di $n^{\log_b a}$

$$f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$$

e

$$af(\frac{n}{b}) < cf(n) \quad \text{per qualche costante } c < 1 \text{ e per tutti gli } n \geq n_0$$

allora la soluzione è

$$T(n) = \Theta\left(f(n)\right)$$

7 Algoritmi

7.1 Merge-sort

L'argorimo *merge-sort* è di tipo **divide-et-impera**, che quindi si basa sull'idea di dividere a metà l'array da ordinare. Quindi lo step *divide* seziona l'array di partenza in 2 parti uguali, poi lo step *impera* ordina entrambe le parti, seguendo sempre l'argorimo *merge-sort*, e infine i due array ordianti vengono *fusi* insieme.

8 Alberi

Un albero è un grafo connesso aciclico. I nodi di grado 1 sono detti foglie, i nodi di grado ¿ 1 sono detti nodi interni. Un albero *radicato* ha 1 nodo **radice**. I nodi interni hanno 1 o più filgi, che consideriamo ordinati e numerati da 0 a k.

Nel caso $k = 2$ l'albero radicato è detto **binario**. In questo caso i filgi si chiamano sinistro (sx) e destro (dx).

9